



Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης  
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών  
Εργαστήριο Επεξεργασίας Πληροφορίας και Υπολογισμών

# ΕΞΟΥΥΞΗ ΔΕΔΟΜΕΝΩΝ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΕΠΑΝΑΧΡΗΣΙΜΟΠΟΙΗΣΗ ΛΟΓΙΣΜΙΚΟΥ

Θεμιστοκλής  
Διαμαντόπουλος

Διατριβή που υποβλήθηκε για τη μερική  
ικανοποίηση των απαιτήσεων για την  
απόκτηση Διδακτορικού Τίτλου Σπουδών

Επιβλέπων:  
Ανδρέας Συμεωνίδης  
*Επίκουρος Καθηγητής*





Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης  
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών  
Εργαστήριο Επεξεργασίας Πληροφορίας και Υπολογισμών

## Εξόρυξη Δεδομένων Τεχνολογίας Λογισμικού για Επαναχρησιμοποίηση Λογισμικού

Διδακτορική Διατριβή

που υποβλήθηκε για τη μερική ικανοποίηση των απαιτήσεων  
για την απόκτηση Διδακτορικού Τίτλου Σπουδών του

Θεμιστοκλή Διαμαντόπουλου του Γεωργίου

Επιβλέπων

Ανδρέας Συμεωνίδης  
*Επίκουρος Καθηγητής*

Συμβουλευτική Επιτροπή

Περικλής Μήτκας  
*Καθηγητής*

Αναστάσιος Ντελόπουλος  
*Αναπληρωτής Καθηγητής*

Επιπλέον Μέλη Εξεταστικής Επιτροπής

Ιωάννης Θεοχάρης  
*Καθηγητής*

Γεώργιος Πάγκαλος  
*Καθηγητής*

Διομήδης Σπινέλλης  
*Καθηγητής*

Αλέξανδρος Χατζηγεωργίου  
*Καθηγητής*

Θεσσαλονίκη, 2018



μηδὲν εἶναι μήτε τέχνην  
ἄνευ μελέτης μήτε  
μελέτην ἄνευ τεχνης

Πρωταγόρας (481 – 411 π.Χ)



# Πρόλογος

Στις επόμενες σελίδες, θα βρείτε τη διδακτορική διατριβή μου, μια εργασία σχετική με τις ευρύτερες περιοχές της ανάλυσης δεδομένων και της κατασκευής λογισμικού. Όταν ξεκίνησα να δουλεύω πάνω σε αυτήν την εργασία, δεν είχα φανταστεί τις προκλήσεις που θα αντιμετώπιζα, την απογοήτευση που με κατέβαλλε τις στιγμές που τίποτα δεν πήγαινε όπως έπρεπε, καθώς και τον ενθουσιασμό μου, όταν οι αμέτρητες ώρες εργασίας που είχα ξοδέψει έδειχναν να έχουν θετικό αποτέλεσμα. Κατά τη διάρκεια των τελευταίων ετών έμαθα πολλά, γνώρισα ανθρώπους και επαναπροσδιόρισα την οπτική μου, όταν ήρθα σε επαφή με διαφορετικούς τρόπους σκέψης. Έτσι, το διδακτορικό αυτό αποτελεί ίσως ένα μικρό απόσπασμα του τι άλλαξε μέσα σε αυτά τα χρόνια στο δικό μου τρόπο σκέψης. Κι αυτό βέβαια δεν περιγράφεται μέσα σε λίγες σελίδες. Κατάλαβα ότι για να πετύχεις αυτό που θέλεις δεν αρκεί το ταλέντο, αλλά απαιτούνται και σκληρή δουλειά, τύχη, κατάλληλες συνθήκες, και φυσικά η στήριξη από συνεργάτες και φίλους. Δοθείσης της ευκαιρίας, θα ήθελα να αναφερθώ σε όλα τα παραπάνω.

Για μένα, οι κατάλληλες συνθήκες παρουσιάστηκαν όταν έφτασα στο Εργαστήριο Επεξεργασίας Πληροφορίας και Υπολογισμών του Αριστοτελείου Πανεπιστημίου Θεσσαλονίκης. Εκεί γνώρισα ανθρώπους με τους οποίους μοιραζόμασταν το ίδιο όραμα και καθένας εκ των οποίων είχε τη δική του πορεία. Το ίδιο το εργαστήριο μου έδωσε τα απαραίτητα εφόδια για να μπορέσω να ολοκληρώσω αυτή τη διατριβή, και, το πιο σημαντικό, να μπορέσω να αποκτήσω τη δική μου θέση στην ερευνητική και ακαδημαϊκή κοινότητα, προσφέροντας με τη σειρά μου αυτά που μπορώ και ευχόμενος να προσφέρω περισσότερα στο μέλλον.

Πρωτίστως, θα ήθελα να ευχαριστήσω τον επίκουρο καθηγητή Ανδρέα Συμεωνίδη, ο οποίος ήταν δίπλα μου σε όλο αυτό το διάστημα όχι μόνο ως επιβλέπων αλλά και ως συνοδοιπόρος στις επιτυχίες και στις δυσκολίες που κατά καιρούς προέκυπταν. Οι συμβουλές του, τόσο στο πλαίσιο της διατριβής, όσο και εκτός αυτού του πλαισίου, με ώθησαν να γίνω καλύτερος επιστήμονας και καλύτερος άνθρωπος.

Επιπλέον, θα ήθελα να ευχαριστήσω τον καθηγητή και πρότανη Περικλή Μήτκα και τον αναπληρωτή καθηγητή Αναστάσιο Ντελόπουλο, καθηγητές-μέλη του εργαστηρίου (και μέλη της τριμελούς επιτροπής αυτής της διατριβής), για τις χρήσιμες συμβουλές τους σε συζητήσεις που είχαμε, καθώς και για το ότι αποτέλεσαν για μένα πρότυπα.

Δε θα μπορούσα να παραλείψω τους συναδέλφους μου από το εργαστήριο με τους οποίους συνεργάστηκα στενά και πολλούς εξ αυτών τους θεωρώ πλέον φίλους μου. Θέλω να εκφράσω ένα μεγάλο ευχαριστώ στο Μιχάλη, στο Χριστόφορο και στον Κυριάκο για την καλή συνεργασία που είχαμε αλλά και για την σημαντική υποστήριξή τους. Ευχαριστώ επίσης τα παλιά και νέα μέλη του εργαστηρίου και ιδιαιτέρως τον Αντώνη, το Μανώλη, το Δημήτρη και τη Μαριάννα για τη στήριξή τους σε διάφορα επίπεδα. Επιθυμώ τέλος να εκφράσω ευ-

χαριστίες και προς τα μέλη της έτερης ομάδας του εργαστηρίου, το Χρήστο, το Γιάννη, και όλους τους άλλους συναδέλφους που βρίσκονταν δίπλα μου όλο αυτόν τον καιρό.

Καθώς η έρευνα είναι πάντα συλλογική διαδικασία, θα ήθελα ακόμα να ευχαριστήσω τους άλλους συνεργάτες μου, τους αποφοίτους Νίκο, Αντώνη, Γιώργο, Χρήστο, Θέμη, Αλεξάνδρα, Κλέαρχο, Μαρίνα, Νίνα και Βάσω, όπως επίσης και τη Βάλια, τον Αλέξανδρο, το Φίλιππο και τον Ηλία, για την καλή συνεργασία που είχαμε.

Ευχαριστώ επίσης τους φίλους μου, που ήταν πάντοτε δίπλα μου μέσα σε όλα αυτά τα χρόνια ακόμα και αν κάποιοι εξ αυτών είναι σε άλλα μέρη. Εκτιμώ τη στήριξή τους σε όλες τις δύσκολες στιγμές και την επιμονή τους στο να μου πουν κάθε φορά αυτό που πραγματικά πιστεύουν και όχι αυτό που θέλω να ακούσω.

Θέλω επίσης να πω ένα μεγάλο ευχαριστώ στην οικογένεια μου που με στήριξε όλα αυτά τα χρόνια, και ήταν εκεί για να συζητήσουν, να συμφωνήσουν και να διαφωνήσουν μαζί μου, για όλες τις επιλογές της ζωής μου, πάντα με γνώμονα το δικό μου καλό.

Τέλος, το μέγιστο ευχαριστώ θα πρέπει να πάει στην κοπέλα μου τη Ρία, γιατί μέσα σε όλο αυτό το διάστημα υπέμεινε το άγχος και την κούρασή μου και ήταν πάντα εκεί για να μου χαρίσει ένα χαμόγελο και να μου φτιάξει τη μέρα. Η στήριξή της κατά τη συγγραφή αυτής της διατριβής ήταν καθοριστική για την επιτυχή ολοκλήρωσή της.

Θεμιστοκλής Διαμαντόπουλος  
Θεσσαλονίκη, Καλοκαίρι 2018



# Περίληψη

Η ανάγκη για την αποτελεσματική ανάπτυξη και συντήρηση λογισμικού έχει εντοπιστεί εδώ και αρκετό καιρό στο σχετικό κλάδο της Τεχνολογίας Λογισμικού. Σήμερα, ωστόσο, με την εισαγωγή νέων πρακτικών ανάπτυξης λογισμικού και πρωτοβουλιών λογισμικού ανοικτού κώδικα, τα δεδομένα λογισμικού που μπορεί να βρει κανείς στο διαδίκτυο είναι άφθονα, επομένως η πρόκληση που προκύπτει είναι η αποτελεσματική αξιοποίησή τους για την παραγωγή καλύτερων προϊόντων λογισμικού. Και η πρόκληση αυτή αποτελεί στην πραγματικότητα ένα πρόβλημα επαναχρησιμοποίησης. Στο πλαίσιο αυτής της διατριβής προτείνουμε μια ενιαία προσέγγιση που περιλαμβάνει την εφαρμογή τεχνικών εξόρυξης δεδομένων σε δεδομένα τεχνολογίας λογισμικού για τη διευκόλυνση της επαναχρησιμοποίησης σε διάφορες φάσεις του κύκλου ζωής του λογισμικού. Η μεθοδολογία μας προτείνει λύσεις για τη φάση του καθορισμού των απαιτήσεων και της εξαγωγής προδιαγραφών, τις φάσεις της σχεδίασης και ανάπτυξης λογισμικού, ενώ συμβάλλει επίσης στην αξιολόγηση της ποιότητας και τον έλεγχο του λογισμικού.

Αρχικά εστιάζουμε στη φάση του καθορισμού των απαιτήσεων και της εξαγωγής προδιαγραφών, όπου κατασκευάζουμε κατάλληλες οντολογίες για την αποθήκευση απαιτήσεων λογισμικού. Χρησιμοποιώντας ένα σύνολο από εργαλεία που σχεδιάσαμε, ο μηχανικός απαιτήσεων μπορεί να εισάγει στο μοντέλο μας λειτουργικές απαιτήσεις γραμμένες σε φυσική γλώσσα, γραφικά σενάρια και διαγράμματα UML. Η μοντελοποίηση που προτείνουμε επιτρέπει την επικύρωση των απαιτήσεων, την εξαγωγή προδιαγραφών, καθώς επίσης και την εφαρμογή τεχνικών εξόρυξης δεδομένων με σκοπό την επαναχρησιμοποίηση απαιτήσεων. Επιπρόσθετα, παρουσιάζουμε μεθοδολογίες για την επαναχρησιμοποίηση απαιτήσεων μέσω τεχνικών κανόνων συσχετίσεων και τεχνικών αντιστοίχισης.

Όσον αφορά τη φάση της ανάπτυξης λογισμικού, εφαρμόζουμε τεχνικές για την επαναχρησιμοποίηση κώδικα σε διαφορετικά επίπεδα. Αρχικά, παρουσιάζουμε τη σχεδίαση μιας μηχανής αναζήτησης κώδικα που είναι προσανατολισμένη στην επαναχρησιμοποίηση. Η μηχανή μας βασίζεται σε ένα ευρετήριο κώδικα που επιτρέπει την αναζήτηση σε επίπεδο τμημάτων κώδικα (components), σε επίπεδο snippets και σε επίπεδο έργων λογισμικού (π.χ. για αναζήτηση αρχιτεκτονικών προτύπων). Στη συνέχεια, προτείνουμε ένα σύστημα προτάσεων κώδικα που αφορά την επαναχρησιμοποίηση οδηγούμενη από ελέγχους (test-driven reuse). Το σύστημά μας χρησιμοποιεί τεχνικές ανάκτησης πληροφοριών, ενώ επιπλέον λαμβάνεται υπόψη η σύνταξη του κώδικα για την επιστροφή τμημάτων που είναι σχετικά με το ερώτημα και εφαρμόζονται κατάλληλοι μετασχηματισμοί για την ενσωμάτωσή τους στον κώδικα του προγραμματιστή. Τα αποτελέσματα εξετάζονται επίσης χρησιμοποιώντας ελέγχους προκειμένου να διασφαλιστεί ότι καλύπτεται η επιθυμητή λειτουργικότητα. Η επόμενη πρόκληση που αντιμετωπίζουμε είναι η σύνδεση των τμημάτων κώδικα, που πραγματοποιείται συνήθως με τη βοήθεια παραδειγμάτων (snippets). Σε αυτόν τον άξονα,

προτείνεται ένα σύστημα που δέχεται ερωτήματα σε φυσική γλώσσα και κατεβάζει παραδείγματα κώδικα, που στη συνέχεια ομαδοποιούνται με βάση τις βιβλιοθήκες που χρησιμοποιούν. Οι βιβλιοθήκες και τα παραδείγματα κατατάσσονται με βάση την προτίμησή τους από την προγραμματιστική κοινότητα. Τέλος, για την περαιτέρω βελτίωση των τμημάτων κώδικα που ανακτήθηκαν (ή γενικά που αναπτύχθηκαν από τον προγραμματιστή), προτείνουμε μια μεθοδολογία επικύρωσης κώδικα με βάση πληροφορίες από υπηρεσίες ερωταπαντήσεων. Αναλύεται το πρόβλημα της εύρεσης χρήσιμων αναρτήσεων σε τέτοιες υπηρεσίες, και συγκεκριμένα προτείνεται η χρήση κώδικα για την εύρεση σχετικών snippets.

Ο τρίτος άξονας αυτής της διατριβής είναι η αξιολόγηση της ποιότητας των ανακτηθέντων τμημάτων λογισμικού. Προς το σκοπό αυτό, παρουσιάζεται αρχικά ένα σύστημα προτάσεων κώδικα που αξιολογεί τα τμήματα λογισμικού τόσο από λειτουργική σκοπιά, όσο και για τη δυνατότητα επαναχρησιμοποίησής τους. Το σύστημα περιλαμβάνει έναν μηχανισμό αντιστοίχισης ερωτημάτων σε τμήματα κώδικα με βάση τη σύνταξη, καθώς και ένα μοντέλο μετρικών στατικής ανάλυσης που αξιολογεί την επαναχρησιμοποιησιμότητα κάθε τμήματος. Στη συνέχεια, σχεδιάζουμε ένα πιο λεπτομερές μοντέλο αξιολόγησης της επαναχρησιμοποιησιμότητας τμημάτων κώδικα με βάση την προτίμησή/επαναχρησιμοποίησή τους από την κοινότητα. Το μοντέλο μας, που κατασκευάζεται χρησιμοποιώντας τεχνικές μηχανικής μάθησης, είναι σε θέση να αξιολογήσει την ποιότητα τμημάτων κώδικα όπως γίνεται αντιληπτή από το χρήστη.

Συμπερασματικά, στην παρούσα διατριβή βασιζόμαστε στην εφαρμογή κατάλληλων τεχνικών εξόρυξης δεδομένων σε δεδομένα τεχνολογίας λογισμικού, ούτως ώστε να παρέχουμε μια ολοκληρωμένη λύση για την εφαρμογή τεχνικών επαναχρησιμοποίησης στις διάφορες φάσεις της ανάπτυξης και συντήρησης λογισμικού. Η λύση αυτή στοχεύει στη δημιουργία καλύτερου λογισμικού με ελάχιστο κόστος και προσπάθεια, επηρεάζοντας έτσι σημαντικές οικονομικές και κοινωνικές πτυχές της καθημερινής ζωής.

Θεμιστοκλής Διαμαντόπουλος  
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών  
Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης  
Καλοκαίρι, 2018

# Mining Software Engineering Data for Software Reuse

## Abstract

The need for effective software development and maintenance has long been identified in the field of Software Engineering. Today, however, that new collaborative development paradigms and open-source software initiatives have been introduced, the amount of software engineering data that are available online is vast, thus the challenge is to effectively analyze them in order to produce better software. And this challenge is actually a problem of reuse. In this thesis, we provide a unified approach that includes applying data mining techniques on software engineering data in order to enable reuse in different phases of the software engineering life cycle. Our methodology offers solutions for the phases of requirements elicitation and specification extraction, software design and development, and quality assurance and testing.

Initially, we focus on requirements elicitation and specification extraction, where we build ontologies for storing software requirements. Using a set of our designed tools, the requirements engineer can instantiate our model from functional requirements written in natural language, graphical storyboards and/or UML diagrams. Our modeling approach allows validating the requirements, extracting specifications, as well as applying data mining techniques for reusing requirements. Furthermore, we propose different methodologies for recommending requirements, based on association rule extraction and matching techniques.

Concerning the phase of software development, we facilitate reuse at different levels. At first, we design a code search engine that is oriented towards reuse. Our engine is based on an index that allows searching for source code components, for code snippets, and for multiple objects (at project level, e.g. for finding architectural patterns). After that, we propose a source code recommendation system that encompasses test-driven reuse. Our system uses information retrieval techniques and employs a syntax-aware mining model to retrieve components that are relevant to the query of the developer. The results are also further examined using tests to ensure that they cover the required functionality. The next challenge that we face is that of connecting source code components, which is usually performed with snippets. In this context, we propose a system that accepts queries in natural language and downloads snippets, which are subsequently clustered according to the libraries used. The libraries and the snippets are ranked according to their preference by the developer community. Finally, in an effort to further improve on retrieved code (or generally any code written by the developer), we propose a code validation methodology given information from question-answering services. We analyze the problem of finding similar question posts in these services, and specifically recommend the use of source code for finding relevant snippets.

The third axis of this thesis involves evaluating the quality of the retrieved components. To this end, we initially present a source code recommendation system that evaluates the retrieved components both from a functional and from a reusability perspective. Our system

involves a syntax-aware matching mechanism and a model that assesses the reusability of each component. After that, we design a more fine-grained reusability estimation model for source code components, based on their preference/reuse rate by the developer community. Our model, which is built using machine learning techniques, is capable of assessing the quality of source code components as perceived by developers.

All in all, in this thesis we apply data mining techniques on software engineering data, in order to provide a unified solution for practicing reuse in all phases of software development and maintenance. This solution aims to create better software with minimal cost and effort, thus affecting important economic and social aspects of everyday life.

Themistoklis Diamantopoulos  
Electrical and Computer Engineering Department  
Aristotle University of Thessaloniki  
Summer, 2018

# Περιεχόμενα

Πρόλογος	v
Περίληψη	vii
Abstract	ix
Περιεχόμενα	xi
Κατάλογος Σχημάτων	xv
Κατάλογος Πινάκων	xix

## Μέρος I - Εισαγωγή και Ερευνητικό Υπόβαθρο

<b>1 Εισαγωγή</b>	<b>3</b>
1.1 Επισκόπηση .....	3
1.2 Σκοπός και Στόχος της Διατριβής .....	5
1.3 Πρόοδος και Συνεισφορά της Διατριβής .....	6
1.4 Διάρθρωση των Κεφαλαίων .....	10
1.5 Συμπληρωματικό υλικό .....	12
<b>2 Θεωρητικό Υπόβαθρο και Βιβλιογραφία</b>	<b>13</b>
2.1 Τεχνολογία Λογισμικού .....	13
2.2 Επαναχρησιμοποίηση Λογισμικού .....	17
2.3 Ποιότητα Λογισμικού .....	24
2.4 Ανάλυση Δεδομένων Τεχνολογίας Λογισμικού .....	35

## Μέρος II - Εξόρυξη Απαιτήσεων

<b>3 Μοντελοποίηση Απαιτήσεων Λογισμικού</b>	<b>49</b>
3.1 Επισκόπηση .....	49

3.2	Βιβλιογραφία για Καθορισμό Απαιτήσεων και Εξαγωγή Προδιαγραφών .....	50
3.3	Εξαγωγή Προδιαγραφών από Απαιτήσεις .....	52
3.4	Μελέτη Περίπτωσης .....	71
3.5	Συμπεράσματα .....	74
<b>4</b>	<b>Εξόρυξη Απαιτήσεων Λογισμικού</b>	<b>75</b>
4.1	Επισκόπηση .....	75
4.2	Βιβλιογραφία για την Εξόρυξη Απαιτήσεων .....	77
4.3	Εξόρυξη Λειτουργικών Απαιτήσεων .....	80
4.4	Εξόρυξη Μοντέλων UML .....	84
4.5	Αξιολόγηση .....	89
4.6	Συμπεράσματα .....	93

### Μέρος III - Εξόρυξη Πηγαίου Κώδικα

<b>5</b>	<b>Ευρετηριοποίηση Κώδικα για Επαναχρησιμοποίηση</b>	<b>97</b>
5.1	Επισκόπηση .....	97
5.2	Βιβλιογραφία για τις Μηχανές Αναζήτησης Κώδικα .....	98
5.3	Η Μηχανή Αναζήτησης Κώδικα AGORA .....	102
5.4	Αναζήτηση Κώδικα .....	111
5.5	Αξιολόγηση .....	121
5.6	Συμπεράσματα .....	127
<b>6</b>	<b>Εξόρυξη Κώδικα για Επαναχρησιμοποίηση Τμημάτων</b>	<b>129</b>
6.1	Επισκόπηση .....	129
6.2	Βιβλιογραφία για τα Συστήματα Προτάσεων στην Τεχνολογία Λογισμικού..	130
6.3	Το Σύστημα Επαναχρησιμοποίησης Κώδικα Mantissa .....	136
6.4	Περιβάλλον Διεπαφής και Σενάριο Αναζήτησης .....	151
6.5	Αξιολόγηση .....	157
6.6	Συμπεράσματα .....	166
<b>7</b>	<b>Εξόρυξη Κώδικα για Επαναχρησιμοποίηση API Snippets</b>	<b>169</b>
7.1	Επισκόπηση .....	169
7.2	Βιβλιογραφία για τα Συστήματα Εξόρυξης Snippets και Χρήσεων API .....	170
7.3	Το Σύστημα Προτάσεων Snippets CodeCatch .....	172
7.4	Αξιολόγηση .....	179
7.5	Συμπεράσματα .....	182

<b>8 Βελτίωση Ερωταπαντήσεων Υλοποίησης με Snippets</b>	<b>183</b>
8.1 Επισκόπηση .....	183
8.2 Συλλογή και Προεπεξεργασία Δεδομένων .....	184
8.3 Μεθοδολογία Αντιστοίχισης .....	187
8.4 Αξιολόγηση .....	189
8.5 Συμπεράσματα .....	191

### Μέρος IV - Αξιολόγηση Ποιότητας

<b>9 Προτάσεις Επαναχρησιμοποιήσιμου Κώδικα</b>	<b>195</b>
9.1 Επισκόπηση .....	195
9.2 Το Σύστημα Προτάσεων Επαναχρησιμοποιήσιμου Κώδικα QualBoa .....	196
9.3 Αξιολόγηση .....	201
9.4 Συμπεράσματα .....	204
<b>10 Αξιολόγηση της Επαναχρησιμοποιησιμότητας Κώδικα</b>	<b>205</b>
10.1 Επισκόπηση .....	205
10.2 Βιβλιογραφία για την Αξιολόγηση Επαναχρησιμοποιησιμότητας .....	206
10.3 Συσχέτιση Επαναχρησιμοποιησιμότητας με Δημοτικότητα .....	208
10.4 Μοντελοποίηση Επαναχρησιμοποιησιμότητας .....	209
10.5 Σύστημα Αξιολόγησης Επαναχρησιμοποιησιμότητας .....	214
10.6 Αξιολόγηση .....	217
10.7 Συμπεράσματα .....	220

### Μέρος V - Συμπεράσματα και Μελλοντική Εργασία

<b>11 Συμπεράσματα</b>	<b>223</b>
<b>12 Μελλοντική Εργασία</b>	<b>225</b>

### Βιβλιογραφία και Λίστα Δημοσιεύσεων

<b>Βιβλιογραφία</b>	<b>229</b>
<b>Λίστα Δημοσιεύσεων</b>	<b>249</b>





# Κατάλογος Σχημάτων

2.1	Βασικές Δραστηριότητες στην Τεχνολογία Λογισμικού .....	14
2.2	Μοντέλο Καταρράκτη .....	15
2.3	Επαναληπτικό Μοντέλο .....	15
2.4	Ευέλικτο Μοντέλο .....	16
2.5	Κατηγορίες Δραστηριοτήτων Τεχνολογίας Λογισμικού .....	17
2.6	Τοπίο Επαναχρησιμοποίησης.....	19
2.7	Δεδομένα Τεχνολογίας Λογισμικού .....	20
2.8	Χαρακτηριστικά και Υπο-χαρακτηριστικά Ποιότητας του ISO/IEC 25010:2011 .	26
2.9	Επίπεδα Ελέγχου και αντίστοιχα Αντικείμενα που Ελέγχονται .....	34
2.10	Εργασίες Εξόρυξης Δεδομένων και Ενδεικτικές Τεχνικές.....	37
2.11	Εξόρυξη Δεδομένων Τεχνολογίας Λογισμικού: Στόχοι, Δεδομένα Εισόδου και Τεχνικές Εξόρυξης .....	39
2.12	Επισκόπηση Εξόρυξης Δεδομένων για Επαναχρησιμοποίηση Λογισμικού και Σχετική Συνεισφορά της Διατριβής .....	43
3.1	Επισκόπηση της Αρχιτεκτονικής του Συστήματός μας .....	52
3.2	Στατική Οντολογία Έργων Λογισμικού.....	54
3.3	Ιδιότητες Στατικής Οντολογίας.....	55
3.4	Screenshot του Requirements Editor.....	56
3.5	Δυναμική Οντολογία Έργων Λογισμικού .....	57
3.6	Ιδιότητες Δυναμικής Οντολογίας .....	59
3.7	Screenshot του Storyboard Creator .....	60
3.8	Μεθοδολογία για την Ανάλυση Απαιτήσεων Λογισμικού .....	61
3.9	Παράδειγμα Σχολιασμού Πρότασης χρησιμοποιώντας την Ιεραρχική Μεθοδολογία Σχολιασμού .....	62
3.10	Σχολιασμένες Απαιτήσεις του Έργου Restmarks.....	63
3.11	Αθροιστική Οντολογία Έργων Λογισμικού .....	66
3.12	Ιδιότητες Αθροιστικής Οντολογίας .....	67
3.13	Σχήμα της Αναπαράστασης YAML .....	69
3.14	Τμήμα των Σχολιασμένων Λειτουργικών Απαιτήσεων του Έργου Restmarks ....	71
3.15	Διάγραμμα Σεναρίου “Add bookmark” του Έργου Restmarks .....	72
3.16	Παράδειγμα Αρχείου YAML για το Έργο Restmarks.....	73
4.1	Παράδειγμα απαίτησης με σχολιασμούς.....	80
4.2	Παράδειγμα τμήματος του WordNet όπου η κλάση <b>record</b> είναι η πιο ειδική κλάση των <b>account</b> και <b>profile</b> .....	81

4.3	Παράδειγμα διαγράμματος σεναρίων χρήσης για το έργο Restmarks .....	84
4.4	Παράδειγμα διαγράμματος σεναρίων χρήσης .....	86
4.5	Παράδειγμα διαγράμματος δραστηριοτήτων για το έργο Restmarks.....	87
4.6	Παράδειγμα διαγράμματος δραστηριοτήτων .....	88
4.7	Παράδειγμα όπου φαίνονται οι λειτουργικές απαιτήσεις του Restmarks, οι προτεινόμενες απαιτήσεις και η οπτικοποίηση των προτεινόμενων απαιτήσεων .....	89
4.8	Οπτικοποίηση των προτεινόμενων απαιτήσεων καθώς και τους ποσοστού των ορθά προτεινόμενων απαιτήσεων για διαφορετικές τιμές υποστήριξης και εμπιστοσύνης .....	91
4.9	Αποτελέσματα ταξινόμησης της προσέγγισής μας και της προσέγγισης του Kelter και των συνεργατών του για διαγράμματα σεναρίων χρήσης και διαγράμματα δραστηριοτήτων .....	92
5.1	Αρχιτεκτονική της AGORA .....	103
5.2	Κανονική έκφραση για τον αναλυτή CamelCase .....	106
5.3	Λίστα από Java stopwords .....	106
5.4	Αρχική Σελίδα της AGORA .....	112
5.5	Παράδειγμα ερωτήματος για μια κλάση “stack” με μεθόδους “push” και “pop” ..	113
5.6	Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.5 .....	114
5.7	Παράδειγμα ερωτήματος για μια κλάση τύπου “Export”, που επεκτείνει μια κλάση “WizardPage” και περιλαμβάνει δήλωση βιβλιοθήκης “eclipse” .....	115
5.8	Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.7 .....	115
5.9	Παράδειγμα ερωτήματος για κλάσεις τύπου “Model”, “View”, “Controller”, σχετικές με το “JFrame” .....	116
5.10	Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.9 .....	117
5.11	Παράδειγμα ερωτήματος για ένα snippet που αφορά το διάβασμα αρχείων XML	117
5.12	Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.11 .....	118
5.13	Ερώτημα για ένα τμήμα διαβάσματος αρχείων .....	119
5.14	Ερώτημα για έναν αλγόριθμο απόστασης επεξεργασίας .....	120
5.15	Ερώτημα για μια κλάση που αναπαριστά ένα ζεύγος .....	121
5.16	Παράδειγμα αλληλουχίας ερωτημάτων για το GitHub .....	123
5.17	Παράδειγμα τροποποιημένης περίπτωσης ελέγχου για ένα αποτέλεσμα .....	123
5.18	Διάγραμμα που απεικονίζει το μέσο πλήθος αποτελεσμάτων που μεταγλωττίστηκαν και αυτών που πέρασαν τους ελέγχους για τις τρεις Μηχανές Αναζήτησης Κώδικα .....	125
5.19	Διαγράμματα που απεικονίζουν το μήκος αναζήτησης για την εύρεση μεταγλωττίσιμων αποτελεσμάτων, και το μήκος αναζήτησης για την εύρεση αποτελεσμάτων που περνάνε τους ελέγχους, για τις τρεις Μηχανές Αναζήτησης Κώδικα .....	127
6.1	Αρχιτεκτονική του Mantissa .....	137
6.2	Παράδειγμα υπογραφής για μια κλάση “Stack” με μεθόδους “push” και “pop” ..	137
6.3	Παράδειγμα AST για τον πηγαίο κώδικα του Σχήματος 6.2 .....	138
6.4	Παράδειγμα αλληλουχίας ερωτημάτων για το Searchcode .....	139
6.5	Παράδειγμα αλληλουχίας ερωτημάτων για το GitHub .....	140
6.6	Κανονική έκφραση για την εξαγωγή δηλώσεων μεθόδων από κώδικα .....	141

6.7	Αλγόριθμος που υπολογίζει την ομοιότητα μεταξύ δύο συνόλων από στοιχεία και επιστρέφει την καλύτερη αντιστοιχισή ως ένα σύνολο από ζεύγη .....	143
6.8	Βήματα προεπεξεργασίας συμβολοσειρών .....	144
6.9	Παράδειγμα υπογραφής για μια κλάση στοίβας με δύο μεθόδους για εισαγωγή και εξαγωγή στοιχείων από τη στοίβα .....	146
6.10	Παράδειγμα αναπαράστασης σε διανυσματικό χώρο για το ερώτημα “Stack” και το αποτέλεσμα “MyStack” .....	147
6.11	Παράδειγμα ερωτήματος για μια κλάση “Customer” με μεθόδους “setAddress” και “getAddress” .....	148
6.12	Παράδειγμα αποτελέσματος για την κλάση “Customer” .....	148
6.13	Παράδειγμα μετασχηματισμού για το αρχείο του Σχήματος 6.12 .....	149
6.14	Παράδειγμα ελέγχου για την κλάση “Customer” του Σχήματος 6.11 .....	150
6.15	Σελίδα αναζήτησης του Mantissa .....	152
6.16	Σελίδα αποτελεσμάτων του Mantissa .....	153
6.17	Παράδειγμα υπογραφής για ένα τμήμα που διαβάζει και γράφει αρχεία.....	153
6.18	Παράδειγμα μεθόδου γραφής για ένα τμήμα που διαχειρίζεται αρχεία .....	154
6.19	Ροή κώδικα για μια μέθοδο γραφής ενός τμήματος που διαχειρίζεται αρχεία .....	154
6.20	Παράδειγμα υπογραφής για ένα τμήμα που υπολογίζει το MD5 hash μιας συμβολοσειράς .....	155
6.21	Παράδειγμα περίπτωσης ελέγχου για το τμήμα “MD5” του Σχήματος 6.20 .....	155
6.22	Παράδειγμα υπογραφής για μια serializable δομή συνόλου .....	155
6.23	Παράδειγμα ελέγχου για τη δομή συνόλου του Σχήματος 6.22 .....	156
6.24	Παράδειγμα για τη δημιουργία ενός συνόλου από συμβολοσειρές MD5 για τα αρχεία του φακέλου folderPath και αποθήκευση του συνόλου σε ένα αρχείο με όνομα setFilename .....	156
6.25	Διάγραμμα που απεικονίζει το μέσο πλήθος αποτελεσμάτων που μεταγλωττίστηκαν και αυτών που πέρασαν τους ελέγχους για τις τρεις Μηχανές Αναζήτησης Κώδικα και τις δύο υλοποιήσεις του Mantissa.....	160
6.26	Διάγραμμα που απεικονίζει το μέσο μήκος αναζήτησης για την εύρεση αποτελεσμάτων που περνάνε τους ελέγχους, για τις τρεις Μηχανές Αναζήτησης Κώδικα και τις δύο υλοποιήσεις του Mantissa.....	162
6.27	Διάγραμμα που απεικονίζει το μέσο πλήθος αποτελεσμάτων που μεταγλωττίστηκαν και αυτών που πέρασαν τους ελέγχους για το FAST και τις δύο υλοποιήσεις του Mantissa .....	164
7.1	Αρχιτεκτονική του CodeCatch .....	172
7.2	Παράδειγμα τμήματος κώδικα για το ερώτημα “How to read a CSV file” .....	174
7.3	Παράδειγμα τμήματος κώδικα για το ερώτημα “How to read a CSV file” .....	176
7.4	Τιμή silhouette για διαφορετικούς αριθμούς ομάδων για το ερώτημα “How to read a CSV file” .....	177
7.5	Τιμή silhouette κάθε ομάδας για ομαδοποίηση σε 5 ομάδες για το ερώτημα “How to read a CSV file” .....	178
7.6	Screenshot του CodeCatch για το ερώτημα “How to read a CSV file”, όπου απεικονίζονται οι πρώτες τρεις ομάδες .....	178
7.7	Screenshot του CodeCatch για το ερώτημα “How to read a CSV file”, όπου απεικονίζεται ένα τμήμα κώδικα .....	179

7.8	Reciprocal Rank των CodeCatch και Google για τις τρεις πιο δημοφιλείς υλοποιήσεις για κάθε ερώτημα .....	181
8.1	Παράδειγμα ανάρτησης στο Stack Overflow .....	185
8.2	Παράδειγμα ακολουθίας για το snippet του Σχήματος 8.1 .....	187
8.3	Παράδειγμα ακολουθίας που εξήχθη από ένα snippet .....	188
8.4	Ποσοστό σχετικών αποτελεσμάτων μέσα στα πρώτα 20 κάθε ερωτήματος, για πλήθος εντολών κώδικα μεγαλύτερο ή ίσο του 1 .....	190
8.5	Ποσοστό σχετικών αποτελεσμάτων μέσα στα πρώτα 20 κάθε ερωτήματος, για πλήθος εντολών κώδικα μεγαλύτερο ή ίσο του 3 και του 5 .....	190
9.1	Αρχιτεκτονική του QualBoa .....	196
9.2	Παράδειγμα υπογραφής για μια κλάση “Stack” με μεθόδους “push” και “pop” ..	196
9.3	Ερώτημα Boa για τμήματα κώδικα και μετρικές .....	198
9.4	Διαγράμματα αξιολόγησης του QualBoa που απεικονίζουν τη μέση ακρίβεια, και τη μέση βαθμολογία επαναχρησιμοποιησιμότητας, για κάθε ερώτημα .....	203
10.1	Διάγραμμα που απεικονίζει τον αριθμό των stars με τον αριθμό των forks για τα 100 πιο δημοφιλή αποθετήρια Java του GitHub .....	208
10.2	Κατανομή σε Επίπεδο Πακέτου της Μετρικής AD για όλα τα Αποθετήρια .....	211
10.3	Κατανομή σε Επίπεδο Πακέτου της Μετρικής AD για δύο Αποθετήρια .....	212
10.4	Βαθμολογία Επαναχρησιμοποιησιμότητας με βάση τα Stars συναρτήσει των Τιμών της Μετρικής AD .....	214
10.5	Επισκόπηση του Συστήματος Αξιολόγησης της Επαναχρησιμοποιησιμότητας ...	215
10.6	Μέσο NRMSE για τους τρεις αλγορίθμους μηχανικής μάθησης .....	216
10.7	Κατανομή της βαθμολογίας επαναχρησιμοποιησιμότητας σε επίπεδο κλάσης, και σε επίπεδο πακέτου .....	217
10.8	Boxplots που απεικονίζουν τις κατανομές επαναχρησιμοποιησιμότητας για 3 έργα που αναπτύχθηκαν από προγραμματιστές και 2 έργα που αναπτύχθηκαν αυτοματοποιημένα, σε επίπεδο κλάσης και σε επίπεδο πακέτου .....	219

# Κατάλογος Πινάκων

2.1	Υπηρεσίες Φιλοξενίας Πηγαίου Κώδικα.....	21
2.2	Μετρικές C&K .....	28
2.3	Μετρικές Αλλαγών του Moser και των συνεργατών του .....	31
3.1	Παράδειγμα Αποθήκευσης Γραφικού Σεναρίου.....	60
3.2	Αντικείμενα της Οντολογίας για τις Οντότητες του Restmarks .....	64
3.3	Ιδιότητες της Οντολογίας για την Απαίτηση FR4 του Restmarks .....	64
3.4	Ιδιότητες Αθροιστικής Οντολογίας .....	66
3.5	Αντιστοίχιση Κλάσεων από τη Στατική και τη Δυναμική Οντολογία στην Αθροιστική Οντολογία .....	68
3.6	Αντιστοίχιση Ιδιοτήτων από τη Στατική και τη Δυναμική Οντολογία στην Αθροιστική Οντολογία .....	68
3.7	Αντικείμενα Resource, Property και Activity για το έργο Restmarks .....	73
4.1	Δείγμα Κανόνων Συσχέτισης που Εξήχθησαν από το Σύνολο Δεδομένων .....	82
4.2	Ενεργοποίηση Κανόνων για ένα Έργο Λογισμικού .....	83
4.3	Αντιστοίχιση μεταξύ Διαγραμμάτων Σεναρίων Χρήσης .....	86
4.4	Αντιστοίχιση μεταξύ Διαγραμμάτων Δραστηριοτήτων .....	88
4.5	Αποτελέσματα Αξιολόγησης για τις Προτεινόμενες Απαιτήσεις.....	90
5.1	Δημοφιλείς Μηχανές Αναζήτησης Κώδικα .....	100
5.2	Σύγκριση Μηχανών Αναζήτησης Κώδικα με την AGORA .....	102
5.3	Mapping των Έργων στο Ευρετήριο της AGORA .....	107
5.4	Mapping των Αρχείων στο Ευρετήριο της AGORA .....	108
5.5	Εσωτερικό Mapping Κλάσης για τα Αρχεία της AGORA.....	109
5.6	Σύνολο Δεδομένων Αξιολόγησης για Μηχανές Αναζήτησης Κώδικα.....	122
5.7	Αποτελέσματα που μεταγλωττίστηκαν και αποτελέσματα που πέρασαν τους ελέγχους για τις τρεις CSEs, για κάθε ερώτημα και κατά μέσο όρο.....	124
5.8	Μήκη Αναζήτησης για τα Μεταγλωττισμένα και Ελεγμένα Αποτελέσματα, για κάθε Αποτέλεσμα και ως Μέσοι Όροι .....	126
6.1	Σύγκριση Χαρακτηριστικών Δημοφιλών Συστημάτων Επαναχρησιμοποίησης με το Mantissa.....	135
6.2	Παραδείγματα προεπεξεργασίας συμβολοσειρών.....	144
6.3	Σύνολο δεδομένων για την αξιολόγηση του Mantissa ενάντια σε Μηχανές Αναζήτησης Κώδικα.....	158

6.4	Αποτελέσματα που μεταγλωττίστηκαν και αποτελέσματα που πέρασαν τους ελέγχους για τις τρεις CSEs και τις δύο εκδόσεις του Mantissa, για κάθε ερώτημα και κατά μέσο όρο .....	159
6.5	Μήκη Αναζήτησης για τα Ελεγμένα Αποτελέσματα, για τις τρεις Μηχανές Αναζήτησης Κώδικα και τις δύο υλοποιήσεις του Mantissa, για κάθε Αποτέλεσμα και ως Μέσοι Όροι .....	161
6.6	Σύνολο δεδομένων για την αξιολόγηση του Mantissa και του FAST .....	163
6.7	Αποτελέσματα που μεταγλωττίστηκαν και αποτελέσματα που πέρασαν τους ελέγχους για το FAST και τις δύο εκδόσεις του Mantissa, για κάθε ερώτημα και κατά μέσο όρο .....	163
6.8	Σύνολο δεδομένων για την αξιολόγηση του Mantissa και του Code Conjuror ....	165
6.9	Αποτελέσματα που πέρασαν τους ελέγχους και χρόνοι απόκρισης για τις δύο εκδοχές του Code Conjuror (Interface-based και Adaptation) και τις δύο υλοποιήσεις του Mantissa (με την AGORA και με το GitHub), για κάθε ερώτημα και κατά μέσο όρο .....	166
7.1	Στατιστικά Ερωτημάτων του Συνόλου Δεδομένων .....	180
8.1	Παράδειγμα Πίνακα Αναζήτησης για το Snippet του Σχήματος 8.1 .....	186
9.1	Μετρικές Επαναχρησιμοποιησιμότητας του QualBoa .....	199
9.2	Μοντέλο Επαναχρησιμοποιησιμότητας του QualBoa .....	201
9.3	Σύνολο Ερωτημάτων για την Αξιολόγηση του QualBoa .....	202
9.4	Αποτελέσματα Αξιολόγησης του QualBoa .....	203
10.1	Κατηγορίες Μετρικών Στατικής Ανάλυσης που σχετίζονται με την Επαναχρησιμοποιησιμότητα .....	209
10.2	Μετρικές Στατικής Ανάλυσης και Εφαρμογή τους σε Διαφορετικά Επίπεδα .....	210
10.3	Στατιστικά του Συνόλου Δεδομένων Αξιολόγησης .....	218
10.4	Τιμές Μετρικών για Κλάσεις και Πακέτα με διαφορετική Επαναχρησιμοποιησιμότητα .....	220

**Μέρος**

**I**

**ΕΙΣΑΓΩΓΗ ΚΑΙ ΕΡΕΥΝΗΤΙΚΟ ΥΠΟΒΑΘΡΟ**





# 1

## Εισαγωγή

### 1.1 Επισκόπηση

Στις μέρες μας, το λογισμικό είναι σχεδόν παντού: πληροφοριακά συστήματα για μεγάλους οργανισμούς και εταιρείες, διαδικτυακές εφαρμογές και εφαρμογές κινητών συσκευών, ενσωματωμένα συστήματα, κ.α., είναι μερικά μόνο παραδείγματα συστημάτων λογισμικού. Στη βιομηχανία ανάπτυξης λογισμικού έχει ήδη εντοπιστεί η ανάγκη για αποτελεσματική ανάπτυξη και συντήρηση λογισμικού για την παραγωγή καλύτερου λογισμικού, το οποίο με τη σειρά του έχει αντίκτυπο σε διάφορες πτυχές της καθημερινής ζωής. Η Τεχνολογία Λογισμικού ήταν πάντα ένας δύσκολος τομέας, κυρίως λόγω του γεγονότος ότι συνεχίζει να εξελίσσεται μαζί με την αδιαμφισβήτητη εξέλιξη της Επιστήμης της Πληροφορικής. Οι τρέχουσες εξελίξεις έχουν αλλάξει τον τρόπο που αντιμετωπίζονται προκλήσεις σε διάφορους τομείς, καθώς και τον τρόπο που αναπτύσσεται λογισμικό με βάση τις καθημερινές ανάγκες.

Προκειμένου να αντιμετωπιστούν οι προκλήσεις που τίθενται από αυτές τις εξελίξεις, η Τεχνολογία Λογισμικού έχει επίσης εξελιχθεί. Παρόλο που λογισμικό αναπτύσσεται για μεγάλο χρονικό διάστημα πριν από την αρχή της εποχής της πληροφορίας [1], οι μεθοδολογίες σχεδίασης, τα εργαλεία που χρησιμοποιούνται, ακόμα και η φιλοσοφία γύρω από την ανάπτυξη λογισμικού έχουν αλλάξει σημαντικά κατά τη διάρκεια αυτής της περιόδου. Μία τέτοια θεμελιώδης αλλαγή είναι το πέρασμα από τον δομημένο προγραμματισμό σε αντικειμενοστραφείς γλώσσες στη δεκαετία του '80, σε λειτουργικές και γλώσσες σεναρίων στη δεκαετία του '90 και ακόμη και σε μεταπρογραμματισμό για τα χρόνια που ακολούθησαν. Τα μοντέλα διαδικασιών λογισμικού έχουν επίσης εξελιχθεί, ξεκινώντας από απλά μοντέλα καταρράκτη (waterfall) και μεταβαίνοντας στο επαναληπτικό (iterative) μοντέλο, στο μοντέλο ανάπτυξης βασισμένο σε τμήματα (component-based) και αργότερα στο ευέλικτο (agile) μοντέλο [2]. Συμπερασματικά, με το πέρασμα του χρόνου άλλαξε παράλληλα και το λογισμικό: η βιομηχανία λογισμικού αναπτύχθηκε, νέες πρακτικές προτάθηκαν, ακόμα και ο σκοπός της ανάπτυξης λογισμικού έχει αλλάξει.

Σε κάθε περίπτωση, οι προκλήσεις που παραδοσιακά αφορούν τη βιομηχανία λογισμικού παραμένουν οι ίδιες (ή τουλάχιστον είναι παρόμοιες). Σύμφωνα με την έκθεση CHAOS του Standish Group [3], το 1994 στις Η.Π.Α., περισσότερα από 250 δισεκατομμύρια δολάρια

δαπανούνται κάθε χρόνο για την ανάπτυξη εφαρμογών πληροφορικής σε περίπου 175.000 έργα. Αυτά τα κόστη δείχνουν μια αναπτυσσόμενη βιομηχανία και μπορεί να μην είναι από μόνα τους ανησυχητικά. Το συγκλονιστικό γεγονός, όμως, προκύπτει από τα συμπεράσματα της έκθεσης, όπου εκτιμάται ότι 81 δισεκατομμύρια δολάρια προερχόμενα από εταιρικά/κυβερνητικά κεφάλαια δαπανήθηκαν για έργα λογισμικού που τελικά ακυρώθηκαν, και άλλα 59 δισεκατομμύρια δολάρια δαπανήθηκαν για έργα που ολοκληρώθηκαν, ωστόσο υπερέβησαν τις αρχικές εκτιμήσεις κόστους και χρόνου. Ακόμα και σήμερα, περισσότερα από 20 χρόνια αργότερα, η βιομηχανία λογισμικού αντιμετωπίζει ουσιαστικά τις ίδιες προκλήσεις: η επικαιροποιημένη έκθεση του 2015 από τον ίδιο οργανισμό [4] δείχνει ότι το 19% των έργων λογισμικού συνήθως αποτυγχάνουν ενώ το 52% κοστίζουν περισσότερο από την αρχική εκτίμηση και/ή παραδίδονται εκπρόθεσμα. Αυτό σημαίνει ότι μόνο ένα 29%, δηλαδή λιγότερο από το ένα τρίτο των έργων, παραδίδονται επιτυχώς, έγκαιρα και εντός των αρχικών οικονομικών προβλέψεων<sup>1</sup>.

Οι αιτίες για τις παραπάνω αποτυχίες/αποκλίσεις είναι ποικίλες. Μια πρόσφατη έρευνα [5] εντόπισε 26 αιτίες, όπως τον ανεπαρκή προσδιορισμό των απαιτήσεων λογισμικού, τις ανακριβείς εκτιμήσεις χρόνου/κόστους, τις μη επαρκώς καθορισμένες προδιαγραφές, την ανεπάρκεια ποιοτικών ελέγχων, την έλλειψη εξοικείωσης της ομάδας ανάπτυξης με τις εμπλεκόμενες τεχνολογίες κ.α. Αν και η αποφυγή κάποιων από αυτές τις αιτίες είναι εφικτή υπό ορισμένες συνθήκες, η πραγματική πρόκληση είναι να διαχειριστεί κανείς όλες ή τουλάχιστον τις περισσότερες από αυτές, διατηρώντας παράλληλα το κόστος και την προσπάθεια που απαιτούνται στο ελάχιστο. Και αυτό μπορεί να επιτευχθεί μόνο με τη λήψη των κατάλληλων αποφάσεων σε όλα τα στάδια του κύκλου ζωής της ανάπτυξης του λογισμικού.

Στο πλαίσιο της βιομηχανίας λογισμικού, ο όρος *Ευφυΐα Λογισμικού (Software Intelligence)*<sup>2</sup> χρησιμοποιείται συνήθως για να περιγράψει ένα σύνολο μεθόδων που “προσφέρουν στους επαγγελματίες λογισμικού ενημερωμένες και συναφείς πληροφορίες για την υποστήριξη των καθημερινών διαδικασιών λήψης αποφάσεων”, ένας ορισμός που δόθηκε από τους Hassan και Xie [8]. Οι επαγγελματίες λογισμικού δεν είναι μόνο οι προγραμματιστές (developers), αλλά όλοι όσοι συμμετέχουν ενεργά στον κύκλο ζωής του λογισμικού, δηλαδή οι ιδιοκτήτες του προϊόντος (product owners), οι υπεύθυνοι για τη συντήρηση του λογισμικού (software maintainers), οι διευθυντές (managers), ακόμα και οι τελικοί χρήστες (end users). Το όραμα του τομέα της Ευφυΐας Λογισμικού είναι η παροχή των μέσων για τη βελτιστοποίηση των διαδικασιών ανάπτυξης ώστε να διευκολύνει όλες τις φάσεις της τεχνολογίας λογισμικού, συμπεριλαμβανομένων του καθορισμού των απαιτήσεων, της συγγραφής του κώδικα, της διασφάλισης ποιότητας κ.λπ. Έτσι, θα γίνει γνωστή η δυναμική των ομάδων λογισμικού και θα ενισχυθεί η λήψη αποφάσεων με βάση την ανάλυση των εκάστοτε δεδομένων.

Όπως ήδη αναφέρθηκε, η έννοια της χρήσης δεδομένων για τη λήψη αποφάσεων αποτελεί βασική επιδίωξη εδώ και αρκετό καιρό. Σήμερα, ωστόσο, υπάρχουν σημαντικές δυ-

<sup>1</sup><https://www.infoq.com/articles/standish-chaos-2015>

<sup>2</sup>Η Ευφυΐα Λογισμικού μπορεί να θεωρηθεί εξειδίκευση της *Επιχειρηματικής Ευφυΐας (Business Intelligence)*, η οποία, σύμφωνα με τον αναλυτή της Gartner Howard Dresner [6], αποτελεί έναν γενικό όρο που χρησιμοποιείται για να περιγράψει “το σύνολο των εννοιών και των μεθόδων που ακολουθούνται για τη βελτίωση των επιχειρηματικών αποφάσεων βάσει στοιχείων και γεγονότων”. Αν και αυτός ο σύγχρονος ορισμός εντοπίζεται στα τέλη της δεκαετίας του '80, ο όρος Επιχειρηματική Ευφυΐα είναι στην πραγματικότητα παλαιότερος από έναν αιώνα και αποδίδεται στον Richard M. Devens [7], ο οποίος τον χρησιμοποίησε για να περιγράψει την ικανότητα του τραπεζίτη Sir Henry Furnese να αποκομίζει κέρδος λαμβάνοντας επιτυχημένες αποφάσεις σύμφωνα με τις πληροφορίες που λάμβανε από το περιβάλλον του.

νατότητες σε αυτό τον άξονα, διότι σήμερα περισσότερο από ποτέ τα δεδομένα είναι διαθέσιμα. Η δημιουργία νέων μοντέλων συνεργατικής ανάπτυξης λογισμικού (collaborative development) και οι πρωτοβουλίες λογισμικού ανοιχτού κώδικα, όπως το Open Source Initiative<sup>3</sup>, έχουν οδηγήσει στη δημιουργία διάφορων ηλεκτρονικών υπηρεσιών, όπως π.χ. οι υπηρεσίες φιλοξενίας κώδικα (code hosting services), οι κοινότητες ερωταπαντήσεων (question-answering communities), τα συστήματα διαχείρισης σφαλμάτων (bug tracking systems), κ.α. Τα δεδομένα που παρέχονται σε αυτές τις υπηρεσίες μπορούν να αξιοποιηθούν για την ανάπτυξη έξυπνων συστημάτων που με τη σειρά τους μπορούν να χρησιμοποιηθούν για τη υποστήριξη της λήψης αποφάσεων για την τεχνολογία λογισμικού.

Η απάντηση στο πώς αυτές οι πηγές δεδομένων μπορούν να αναλυθούν και να αξιοποιηθούν αποτελεσματικά είναι στην πραγματικότητα ένα πρόβλημα επαναχρησιμοποίησης. Η επαναχρησιμοποίηση λογισμικού είναι ένας όρος που αναφέρεται για πρώτη φορά από τον μηχανικό της Bell Douglas McIlroy, ο οποίος πρότεινε τη χρήση ήδη υπαρχόντων τμημάτων κατά την ανάπτυξη νέων προϊόντων [9]. Γενικότερα, η επαναχρησιμοποίηση λογισμικού μπορεί να οριστεί ως “η χρήση υπαρχόντων στοιχείων σε κάποια μορφή μέσα στη διαδικασία ανάπτυξης λογισμικού” [10], όπου τα στοιχεία μπορούν να είναι συστατικά λογισμικού (software artifacts) (π.χ. απαιτήσεις, πηγαίος κώδικας κ.α.) ή γνώση που προκύπτει από δεδομένα (π.χ. μετρήσεις λογισμικού, σημασιολογικές πληροφορίες κ.α.) [11]. Προφανώς, η επαναχρησιμοποίηση εφαρμόζεται ήδη σε κάποιο βαθμό, π.χ. οι βιβλιοθήκες λογισμικού είναι ένα καλό παράδειγμα επαναχρησιμοποίησης υπάρχουσας λειτουργικότητας. Ωστόσο, η πρόκληση είναι να αναγνωριστεί πλήρως η δυναμική των δεδομένων προκειμένου να αξιοποιηθούν πραγματικά τα υπάρχοντα στοιχεία και στη συνέχεια να βελτιωθούν οι πρακτικές της βιομηχανίας λογισμικού. Έτσι, μπορεί κανείς να υποστηρίξει ότι τα δεδομένα είναι διαθέσιμα και η πρόκληση πλέον είναι να τα αναλύσουμε για να εξαγάγουμε την απαιτούμενη γνώση και να την αναμορφώσουμε για να την ενσωματώσουμε επιτυχώς στη διαδικασία ανάπτυξης λογισμικού [12].

## 1.2 Σκοπός και Στόχος της Διατριβής

Τα δεδομένα ενός έργου λογισμικού περιλαμβάνουν τον πηγαίο κώδικά του καθώς και όλα τα αρχεία που τον συνοδεύουν, δηλαδή τεκμηρίωση, αρχεία readme κ.α. Στην περίπτωση χρήσης κάποιου συστήματος ελέγχου εκδόσεων, τα δεδομένα ενδέχεται επίσης να περιλαμβάνουν πληροφορίες σχετικά με commits, κύκλους εκδόσεων (release cycles), κ.α., ενώ η ενσωμάτωση και ενός συστήματος διαχείρισης σφαλμάτων (bug tracking system) μπορεί να περιέχει ζητήματα (issues), σφάλματα (bug), συνδέσμους σε commits που τα επιλύουν, κ.λπ. Σε ορισμένες περιπτώσεις, μπορεί επίσης να υπάρχει πρόσβαση στις απαιτήσεις του έργου, οι οποίες μπορεί να έχουν διάφορες μορφές, π.χ. κείμενο, διαγράμματα UML κ.λπ. Τέλος, τα ακατέργαστα δεδομένα αυτών των πηγών μπορούν να χρησιμεύσουν για την παραγωγή επεξεργασμένων δεδομένων με τη μορφή μετρικών λογισμικού [13] που χρησιμοποιούνται για τη μέτρηση διαφόρων πτυχών ποιότητας της διαδικασίας λογισμικού (π.χ. μετρήσεις που αφορούν τον κώδικα του λογισμικού, μετρήσεις που αφορούν το προσωπικό κ.λπ.). Έτσι, δεν υπάρχει καμία αμφιβολία ότι τα δεδομένα είναι διαθέσιμα· η πρόκληση είναι να τα αξιοποιήσουμε αποτελεσματικά για την επίτευξη συγκεκριμένων στόχων.

---

<sup>3</sup><http://opensource.org/>

Η φιλοδοξία που θέτει η παρούσα εργασία είναι να αξιοποιήσει αυτά τα δεδομένα τεχνολογίας λογισμικού, προκειμένου να διευκολύνει τις ενέργειες και τις αποφάσεις των διαφόρων φορέων που συμμετέχουν σε ένα πρόγραμμα λογισμικού. Ειδικότερα, θα μπορούσαμε να συνοψίσουμε το αντικείμενο της διατριβής ως εξής:

*Εφαρμογή τεχνικών εξόρυξης δεδομένων σε δεδομένα  
τεχνολογίας λογισμικού για τη διευκόλυνση των  
διαφόρων φάσεων του κύκλου ζωής του λογισμικού*

Αυτός ο σκοπός είναι προφανώς αρκετά ευρύς ώστε να καλύπτει διαφορετικούς τύπους δεδομένων και διαφορετικά αποτελέσματα, ωστόσο ο στόχος δεν είναι να εξαντληθούν όλες οι πιθανές μεθοδολογίες στον τομέα της τεχνολογίας λογισμικού. Αντί αυτού, εστιάζουμε στη δημιουργία μιας ενιαίας προσέγγισης, δείχνοντας πως η ανάλυση δεδομένων μπορεί να βελτιώσει τις διαδικασίες λογισμικού.

Εξετάζουμε το συνολικό εγχείρημα κάτω από το πρίσμα της επαναχρησιμοποίησης λογισμικού (*software reuse*) και επεξηγούμε πως ήδη υπάρχοντα δεδομένα μπορούν να αξιοποιηθούν για να βοηθήσουν στη διαδικασία ανάπτυξης λογισμικού. Η λύση που προτείνουμε προωθεί την έρευνα στη φάση καθορισμού των απαιτήσεων (*requirements elicitation*), συνεχίζει με τις φάσεις της σχεδίασης (*design*) και εξαγωγής προδιαγραφών (*specification extraction*) όπου δίνεται έμφαση στην ανάπτυξη λογισμικού και στον έλεγχο (*testing*), ενώ συμβάλλει επίσης στην αξιολόγηση της ποιότητας του λογισμικού (*software quality evaluation*). Χρησιμοποιώντας τα προϊόντα της έρευνάς μας, φιλοδοξούμε ότι οι ομάδες ανάπτυξης λογισμικού θα είναι σε θέση να εξοικονομήσουν σημαντικό χρόνο και ανθρωποπροσπάθεια και να πλοηγηθούν ευκολότερα στη διαδικασία παραγωγής λογισμικού.

### 1.3 Πρόσδος και Συνεισφορά της Διατριβής

Η περιοχή της Εξόρυξης Δεδομένων Τεχνολογίας Λογισμικού (*Mining Software Engineering Data*) βρίσκεται επί του παρόντος σε μια “έκρηξη” παραγωγής καινοτόμων ιδεών και ερευνητικών κατευθύνσεων. Η περιοχή αυτή έφτασε στην τρέχουσα κατάστασή της στις αρχές του 21ου αιώνα, όταν η ερευνητική κοινότητα έθεσε ενδιαφέροντα ερωτήματα σχετικά με το λογισμικό, την διαδικασία ανάπτυξής του και την ποιότητά του.

Η θέση της παρούσας διατριβής σε αυτήν την αρκετά ενεργή επιστημονική περιοχή είναι ιδιαίτερη: δεν επιθυμούμε να αμφισβητήσουμε την τρέχουσα έρευνα από άποψη αποτελεσμάτων. Αντίθετα, επιδιώκουμε να παράγουμε μια ενιαία προσέγγιση που θα βοηθήσει πρακτικά τη διαδικασία ανάπτυξης λογισμικού. Ως εκ τούτου, εστιάζουμε σε μερικές από τις πιο επιρρεπείς σε σφάλματα φάσεις της τεχνολογίας λογισμικού και προσπαθούμε όχι μόνο να εξάγουμε συμπεράσματα πέραν από το τρέχον *state-of-the-art*, αλλά και να δείξουμε πώς ο συνδυασμός αυτών των συμπερασμάτων μπορεί να αποβεί πολύ χρήσιμος στις ομάδες ανάπτυξης λογισμικού. Συγκεκριμένα, εστιάζουμε στις ακόλουθες περιοχές:

- Εξόρυξη Απαιτήσεων (*Requirements Mining*)
- Εξόρυξη Πηγαίου Κώδικα (*Source Code Mining*)
- Αξιολόγηση Ποιότητας (*Quality Assessment*)

Το state-of-the-art σε αυτές τις περιοχές και η σχετική συνεισφορά της παρούσας διατριβής αναλύονται στις ακόλουθες ενότητες. Σημειώνουμε ότι σε αυτό το υποκεφάλαιο παρέχεται μια γενικότερη ανάλυση, ενώ διεξοδικές ανασκοπήσεις της εκάστοτε βιβλιογραφίας πραγματοποιούνται στα αντίστοιχα κεφάλαια.

### 1.3.1 Εξόρυξη Απαιτήσεων

Οι φάσεις του καθορισμού των απαιτήσεων και της εξαγωγής των προδιαγραφών ενός έργου λογισμικού είναι πολύ σημαντικές. Οι ανακριβείς ή ελλιπείς απαιτήσεις φαίνεται να είναι ο συνηθέστερος λόγος αποτυχίας [5], καθώς έχουν επιπτώσεις σε όλες τις μεταγενέστερες φάσεις της διαδικασίας ανάπτυξης λογισμικού. Στην παρούσα διατριβή, εστιάζουμε στη μοντελοποίηση και στην εξόρυξη απαιτήσεων. Ως εκ τούτου, στόχος μας είναι να σχεδιάσουμε ένα μοντέλο ικανό να αποθηκεύει απαιτήσεις λογισμικού που θα επιτρέψει επιπλέον την επαναχρησιμοποίησή τους. Το τρέχον state-of-the-art στην περιοχή αυτή επικεντρώνεται κυρίως σε μοντέλα που αφορούν συγκεκριμένους τομείς (domain-specific models), και έχουν ως σκοπό την επικύρωση απαιτήσεων, την πρόταση νέων απαιτήσεων και την ανάκτηση εξαρτήσεων μεταξύ απαιτήσεων ή μεταξύ απαιτήσεων και τμημάτων λογισμικού.

Οι τρέχουσες μέθοδοι μπορούν να είναι αποτελεσματικές υπό ορισμένες συνθήκες, ωστόσο περιορίζονται από τον τομέα για τον οποίο έχουν αναπτυχθεί. Η συμβολή μας σε αυτή την περιοχή είναι ένα νέο μοντέλο που δεν περιορίζεται στο λεξιλόγιο κάποιου τομέα (είναι δηλαδή domain-agnostic) και υποστηρίζει την αποθήκευση απαιτήσεων λογισμικού που εξάγονται από πολλαπλές πηγές (multimodal). Το μοντέλο μας επιπλέον είναι συμβατό με τη στατική όψη (static view) και τη δυναμική όψη (dynamic view) των έργων λογισμικού. Τέλος, συνεισφέρουμε στην έρευνα για τα συστήματα προτάσεων για απαιτήσεις λογισμικού (requirements recommendation systems), όπου σχεδιάζουμε μια μεθοδολογία εξόρυξης που ελέγχει αν οι απαιτήσεις ενός έργου λογισμικού είναι πλήρεις και προτείνει νέες απαιτήσεις, επιτρέποντας έτσι στους μηχανικούς να βελτιώσουν την υπάρχουσα λειτουργικότητα και τη ροή δεδομένων (data flow) στο έργο τους.

### 1.3.2 Εξόρυξη Πηγαίου Κώδικα

Η περιοχή της εξόρυξης πηγαίου κώδικα έχει πρόσφατα γίνει αρκετά δημοφιλής, καθώς η ταχύτερη ανάπτυξη λογισμικού υψηλής ποιότητας συνήθως μεταφράζεται σε ταχύτερους χρόνους διάθεσης στην αγορά (time-to-market) και μικρότερα κόστη. Η ιδέα της ενίσχυσης των προγραμματιστών κατά τη συγγραφή κώδικα αρχικά καλύφθηκε κυρίως από τα κατάλληλα εργαλεία (π.χ. Ολοκληρωμένα Περιβάλλοντα Ανάπτυξης - Integrated Development Environments (IDEs)) και πλαίσια ανάπτυξης (γλώσσες, βιβλιοθήκες κ.λπ.). Κατά τη διάρκεια των τελευταίων ετών, μπορεί να παρατηρηθεί μια σαφής τάση προς την ανάπτυξη λογισμικού βασισμένη σε τμήματα (component-based). Ως εκ τούτου, οι προγραμματιστές συχνά αναζητούν έτοιμες λύσεις για την κάλυψη της επιθυμητής λειτουργικότητας. Αυτές οι λύσεις μπορούν να είναι επαναχρησιμοποιήσιμα τμήματα έργων λογισμικού, λειτουργίες που προσφέρονται από APIs βιβλιοθηκών λογισμικού, απαντήσεις σε ερωτήματα σε μηχανές αναζήτησης (search engines) και/ή συστήματα ερωταπαντήσεων (question-answering systems) κ.λπ. Επομένως, εστιάζουμε σε ένα πλήθος διαφορετικών περιοχών προκειμένου να καλύψουμε ένα μεγάλο μέρος αυτού που ονομάζουμε επαναχρησιμοποίηση κώδικα (code reuse) ή επαναχρησιμοποίηση λειτουργικότητας (functionality reuse).

Η πρώτη συνεισφορά μας είναι μια εξειδικευμένη μηχανή αναζήτησης για πηγαίο κώδικα. Υπάρχουν αρκετές *Μηχανές Αναζήτησης Κώδικα (Code Search Engines - CSEs)*, οι περισσότερες εκ των οποίων έχουν αναπτυχθεί τα τελευταία 15 χρόνια. Ωστόσο, οι περισσότερες σύγχρονες CSEs έχουν λίγο έως πολύ τις ίδιες αδυναμίες: δεν προσφέρουν προηγμένους τύπους ερωτημάτων σύμφωνα με τη σύνταξη του κώδικα, δεν είναι συνδεδεμένες με υπηρεσίες φιλοξενίας κώδικα (code hosting services), επομένως μπορούν εύκολα να καταστούν παρωχημένες, και δεν προσφέρουν πλήρη και τεκμηριωμένα APIs. Αντίθετα, η CSE μας εκτελεί εκτεταμένη ευρετηριοποίηση (indexing), επιτρέποντας αρκετούς διαφορετικούς τύπους ερωτημάτων, ενώ ταυτόχρονα ενημερώνεται συνεχώς από υπηρεσίες φιλοξενίας κώδικα. Επιπλέον, το API που προσφέρει μπορεί να χρησιμεύσει για την κατασκευή διαφόρων εργαλείων: ενδεικτικά, εμείς οι ίδιοι το χρησιμοποιούμε ως πυρήνα για την κατασκευή διαφορετικών υπηρεσιών.

Ένα από τα πιο τυπικά σενάρια χρήσης για μια CSE είναι η σύνδεσή της με ένα *Σύστημα Προτάσεων στην Τεχνολογία Λογισμικού (Recommendation System in Software Engineering - RSSE)* [14]. Όπως και προηγουμένως, εντοπίζουμε πληθώρα τέτοιων συστημάτων, τα περισσότερα από τα οποία ακολουθούν την ίδια πορεία ενεργειών: δέχονται ένα ερώτημα για ένα τμήμα κώδικα από τον προγραμματιστή, αναζητούν χρήσιμα τμήματα κώδικα χρησιμοποιώντας μια μηχανή αναζήτησης, εκτελούν κάποιου τύπου βαθμολόγηση για την αξιολόγηση των τμημάτων, και ενδεχομένως ενσωματώνουν κάποιο τμήμα στον πηγαίο κώδικα του προγραμματιστή. Για να επιτύχουμε πρόοδο πέρα από το state-of-the-art, σχεδιάσαμε και αναπτύξαμε ένα νέο RSSE, το οποίο δέχεται ερωτήματα σε μορφή διασυνδέσεων (interfaces) τμημάτων που είναι συνήθως σαφείς για τον προγραμματιστή. Το RSSE μας κατεβάζει σχετικά τμήματα και τα κατατάσσει χρησιμοποιώντας ένα μοντέλο εξόρυξης που ακολουθεί τη σύνταξη της γλώσσας (syntax-aware), ενώ ταυτόχρονα τα μετασχηματίζει αυτόματα ώστε να επιτρέψει την εύκολη ενσωμάτωσή τους στον πηγαίο κώδικα του προγραμματιστή. Τέλος, τα αποτελέσματα μπορούν να αναλυθούν περαιτέρω χρησιμοποιώντας ελέγχους (test cases) που παρέχονται από τον προγραμματιστή προκειμένου να διασφαλιστεί ότι καλύπτεται η επιθυμητή λειτουργικότητα.

Παρόμοια με τα παραπάνω είναι και η συμβολή μας στις περιοχές έρευνας της εξόρυξης μικρών τμημάτων κώδικα και χρήσεων API (snippet and API usage mining). Συγκεκριμένα, αναπτύσσουμε μια μεθοδολογία για την ανάκτηση παραδειγμάτων χρήσης API για κοινά προγραμματιστικά ερωτήματα. Βασιζόμαστε στη σύνταξη του πηγαίου κώδικα για να εξαγάγουμε πληροφορίες που μπορούν να χρησιμοποιηθούν για την παροχή χρήσιμων snippets. Όσον αφορά τις τρέχουσες προσεγγίσεις, η συνεισφορά μας σε αυτή την περίπτωση μπορεί να αξιολογηθεί ποσοτικά, καθώς αποδεικνύουμε ότι η μεθοδολογία μας μπορεί να προσφέρει παραδείγματα χρήσεων API για διαφορετικά ερωτήματα. Ως άλλη μια ακόμα συνεισφορά μας σε αυτήν την περιοχή, παρέχουμε επίσης μια μεθοδολογία που έχει ως σκοπό την αξιολόγηση της χρησιμότητας και της ορθότητας των snippets χρησιμοποιώντας δεδομένα από κοινότητες ερωταπαντήσεων (question-answering communities).

### 1.3.3 Αξιολόγηση Ποιότητας

Έχοντας αναπτύξει ένα τμήμα/πρωτότυπο/προϊόν λογισμικού, μπορεί κανείς να το αξιολογήσει σε δύο άξονες: αν καλύπτει επαρκώς τη λειτουργικότητα για την οποία σχεδιάστηκε και αναπτύχθηκε (που συνήθως εκφράζεται με λειτουργικές απαιτήσεις - functional

requirements) και εάν καλύπτει τα μη λειτουργικά κριτήρια (non-functional requirements) σύμφωνα με στα οποία σχεδιάστηκε [15]. Η μέτρηση αυτού που ονομάζουμε Ποιότητα Λογισμικού (Software Quality) είναι μια ιδιαίτερη τέχνη, καθώς ο όρος “ποιότητα” εξαρτάται σε μεγάλο βαθμό από το γενικότερο πλαίσιο και μπορεί να έχει διαφορετική σημασία για διαφορετικούς ανθρώπους [16]. Κατά τη διάρκεια των τελευταίων ετών, η κοινή πρακτική είναι η μέτρηση χαρακτηριστικών ποιότητας τα οποία καθορίζονται από το διεθνές πρότυπο ISO [17] χρησιμοποιώντας μετρικές λογισμικού [13]. Έτσι, η έρευνα σε αυτόν τον τομέα επικεντρώνεται σε μεγάλο βαθμό στην πρόταση νέων μετρικών που προέρχονται από διαφορετικά δεδομένα τεχνολογίας λογισμικού.

Αρχικά, εστιάζουμε στα RSSEs επαναχρησιμοποίησης κώδικα που περιγράφηκαν στην προηγούμενη ενότητα. Όπως ήδη αναφέρθηκε, πολλά από αυτά τα συστήματα μπορεί να είναι αποτελεσματικά από άποψη κάλυψης λειτουργικότητας, καθώς προτείνουν τμήματα που είναι κατάλληλα για το ερώτημα του προγραμματιστή. Ωστόσο, δεν αξιολογούν την επαναχρησιμοποιησιμότητα (reusability) των προτεινόμενων τμημάτων λογισμικού, δηλαδή το βαθμό στον οποίο ένα τμήμα είναι επαναχρησιμοποιήσιμο. Συνεπώς, σχεδιάζουμε και αναπτύσσουμε ένα RSSE που αξιολογεί τμήματα λογισμικού όχι μόνο από λειτουργική αλλά και από μη λειτουργική σκοπιά. Εκτός από το να προτείνει λειτουργικά χρήσιμα τμήματα, το σύστημά μας επιπλέον χρησιμοποιεί ένα προσαρμόσιμο μοντέλο που βασίζεται σε μετρικές στατικής ανάλυσης (static analysis metrics) για την αξιολόγηση της επαναχρησιμοποιησιμότητας κάθε τμήματος.

Σε μια προσπάθεια να αξιολογήσουμε καλύτερα την ποιότητα και την επαναχρησιμοποιησιμότητα των εξαρτημάτων λογισμικού, διεξάγουμε περαιτέρω έρευνα στους τομείς της αξιολόγησης ποιότητας (quality assessment) και αξιολόγησης επαναχρησιμοποιησιμότητας (reusability assessment) με τη χρήση μετρικών στατικής ανάλυσης. Υπάρχει πληθώρα προσεγγίσεων που αφορά την κατασκευή μοντέλων αξιολόγησης της ποιότητας, δηλαδή μοντέλων που λαμβάνουν ως είσοδο τιμές μετρικών πηγαίου κώδικα και παράγουν μια τιμή ποιότητας (quality score) με βάση το εάν οι μετρικές υπερβαίνουν ορισμένα όρια (metric thresholds). Ωστόσο, αυτές οι προσεγγίσεις βασίζονται συνήθως στη γνώση εμπειρογνομόνων είτε για τον προσδιορισμό των ορίων των μετρικών, είτε για την κατασκευή ενός συνόλου από εξακριβωμένες τιμές ποιότητας (ground truth) που στη συνέχεια χρησιμοποιούνται για τον αυτόματο προσδιορισμό των εν λόγω ορίων. Η συνεισφορά μας σε αυτόν τον άξονα είναι μια μεθοδολογία για τον προσδιορισμό της ποιότητας ενός τμήματος λογισμικού όπως γίνεται αντιληπτή από τους προγραμματιστές (developer-perceived quality), και επομένως δεν απαιτεί τη βοήθεια εμπειρογνομόνων· αντιθέτως, χρησιμοποιούμε τη δημοτικότητα (popularity) όπως εκφράζεται από online υπηρεσίες και αποδεικνύουμε ότι τμήματα που προτιμώνται συχνά για επαναχρησιμοποίηση (highly reused) είναι συνήθως υψηλής ποιότητας.

Σχεδιάζουμε και αναπτύσσουμε ένα σύστημα που χρησιμοποιεί δεδομένα από online υπηρεσίες για την κατασκευή ενός συνόλου εξακριβωμένων τιμών ποιότητας (ground truth) και στη συνέχεια εκπαιδεύουμε μοντέλα ικανά να εκτιμήσουν το βαθμό επαναχρησιμοποιησιμότητας τόσο ενός έργου λογισμικού στο σύνολό του, όσο και μεμονωμένων τμημάτων. Το σύστημά μας εστιάζει στα βασικά χαρακτηριστικά και ιδιότητες ποιότητας (π.χ. πολυπλοκότητα - complexity, σύζευξη - coupling, κ.α.) που πρέπει να αξιολογηθούν για να προσδιοριστεί αν ένα τμήμα λογισμικού είναι επαναχρησιμοποιήσιμο.

## 1.4 Διάρθρωση των Κεφαλαίων

Όπως ήδη αναφέρθηκε στα υποκεφάλαια 1.2 και 1.3 αυτού του κεφαλαίου, το αντικείμενο αυτής της εργασίας εφάπτεται στην περιοχή της επαναχρησιμοποίησης λογισμικού και αφορά τρεις διαφορετικές φάσεις του λογισμικού. Ως εκ τούτου, το κείμενο επιτρέπει μια άνετη ανάγνωση από πάνω προς τα κάτω. Ωστόσο, καθώς ορισμένοι αναγνώστες μπορεί να ενδιαφέρονται για μία μόνο από τις φάσεις της τεχνολογίας λογισμικού που πραγματευόμαστε, σημαντική προσπάθεια έχει καταβληθεί για να χωριστεί το κείμενο σε αυτόνομα μέρη, που επιτρέπουν έτσι σε κάποιον να επικεντρωθεί στην περιοχή έρευνας που επιθυμεί, όπως αυτές περιγράφηκαν στο υποκεφάλαιο 1.3. Έτσι, μετά το πρώτο μέρος της παρούσας διατριβής, μπορεί κανείς να παραλείψει κάποιο από τα τρία επόμενα μέρη και να ολοκληρώσει την ανάγνωση με τα συμπεράσματα του τελευταίου μέρους. Η διατριβή αποτελείται από 5 διαφορετικά μέρη, ενώ κάθε μέρος περιλαμβάνει ένα σύνολο διαφορετικών κεφαλαίων. Η διάρθρωση της διατριβής είναι η παρακάτω:

### Μέρος I: Εισαγωγή και Ερευνητικό Υπόβαθρο

Αυτό το μέρος περιέχει όλες τις εισαγωγικές πληροφορίες της διατριβής, αναλύοντας το πεδίο εφαρμογής της και τις σχετικές ερευνητικές περιοχές. Περιλαμβάνει τα ακόλουθα κεφάλαια:

#### Κεφάλαιο 1. Εισαγωγή

Το τρέχον κεφάλαιο που περιλαμβάνει όλες τις πληροφορίες που απαιτούνται για την κατανόηση του αντικειμένου και του σκοπού αυτής της διατριβής, καθώς και μια ανάλυση των συνεισφορών της διατριβής.

#### Κεφάλαιο 2. Θεωρητικό Υπόβαθρο και Βιβλιογραφία

Αυτό το κεφάλαιο περιέχει το βασικό υπόβαθρο που απαιτείται για την ανάγνωση των υπόλοιπων μερών και κεφαλαίων της διατριβής. Σε αυτό δίνονται όλοι οι απαραίτητοι ορισμοί σχετικά με την Τεχνολογία Λογισμικού, την Επαναχρησιμοποίηση Λογισμικού, την Ποιότητα Λογισμικού, καθώς και όλες τις σχετικές περιοχές.

### Μέρος II: Εξόρυξη Απαιτήσεων

Αυτό το μέρος αφορά τη συνεισφορά της διατριβής στον τομέα της Μηχανικής Απαιτήσεων (Requirements Engineering). Περιλαμβάνει τα ακόλουθα κεφάλαια:

#### Κεφάλαιο 3. Μοντελοποίηση Απαιτήσεων Λογισμικού

Σε αυτό το κεφάλαιο παρουσιάζεται ένα μοντέλο για απαιτήσεις λογισμικού που μπορεί να αποθηκεύσει απαιτήσεις που εξάγονται από πολλαπλές πηγές και επιτρέπει την εκτέλεση διαφορετικών ενεργειών στις αποθηκευμένες απαιτήσεις. Ως εκ τούτου, το μοντέλο αυτό αποτελεί και τη βάση του επόμενου κεφαλαίου.



**Κεφάλαιο 4. Εξόρυξη Απαιτήσεων Λογισμικού**

Σε αυτό το κεφάλαιο παρουσιάζονται οι μεθοδολογίες εξόρυξης για λειτουργικές απαιτήσεις και μοντέλα UML. Οι δύο μεθοδολογίες εξηγούνται βήμα προς βήμα και δίνονται παραδείγματα προτάσεων απαιτήσεων για κάθε περίπτωση.

**Μέρος III: Εξόρυξη Πηγαίου Κώδικα**

Αυτό το μέρος αφορά τη συνεισφορά της διατριβής στον τομέα της Ανάπτυξης Λογισμικού (Software Development). Περιλαμβάνει τα ακόλουθα κεφάλαια:

**Κεφάλαιο 5. Ευρετηριοποίηση Κώδικα για Επαναχρησιμοποίηση**

Σε αυτό το κεφάλαιο παρουσιάζουμε τη σχεδίαση μιας CSE που είναι προσανατολισμένη στην επαναχρησιμοποίηση πηγαίου κώδικα. Το σύστημά μας διευκολύνει την επαναχρησιμοποίηση σε επίπεδο τμημάτων, σε επίπεδο snippets και σε επίπεδο έργων λογισμικού και μπορεί να φανεί χρήσιμο για τη διερεύνηση διάφορων ερευνητικών ερωτημάτων.

**Κεφάλαιο 6. Εξόρυξη Κώδικα για Επαναχρησιμοποίηση Τμημάτων**

Σε αυτό το κεφάλαιο παρουσιάζουμε αρχικά τη σχεδίαση ενός RSSE που αφορά την επαναχρησιμοποίηση οδηγούμενη από ελέγχους (test-driven reuse). Επιδεικνύουμε πως το σύστημά μας μπορεί να είναι χρήσιμο για τον εντοπισμό επαναχρησιμοποιήσιμων τμημάτων κώδικα και το αξιολογούμε σε πλήθος διαφορετικών ερωτημάτων έναντι παρόμοιων συστημάτων.

**Κεφάλαιο 7. Εξόρυξη Κώδικα για Επαναχρησιμοποίηση Snippets**

Αυτό το κεφάλαιο περιέχει τη μεθοδολογία μας για την εξόρυξη παραδειγμάτων API σε μορφή snippets. Σχεδιάζουμε ένα σύστημα για την ανεύρεση διαφορετικών υλοποιήσεων (implementations) που είναι σχετικές με συχνά προγραμματιστικά ερωτήματα.

**Κεφάλαιο 8. Βελτίωση Ερωταπαντήσεων Υλοποίησης με Snippets**

Αυτό το κεφάλαιο περιέχει τη μεθοδολογία μας για την επικύρωση των snippets χρησιμοποιώντας πληροφορίες από συστήματα ερωταπαντήσεων. Σχεδιάζουμε ένα σύστημα για τον εντοπισμό σχετικών ερωτήσεων για κοινά ερωτήματα.

**Μέρος IV: Αξιολόγηση Ποιότητας**

Αυτό το μέρος αφορά τη συνεισφορά της διατριβής στον τομέα της Ποιότητας Λογισμικού (Software Quality). Περιλαμβάνει τα ακόλουθα κεφάλαια:

**Κεφάλαιο 9. Προτάσεις Επαναχρησιμοποιήσιμου Κώδικα**

Σε αυτό το κεφάλαιο παρουσιάζουμε ένα RSSE που αξιολογεί τα τμήματα λογισμικού τόσο από λειτουργική σκοπιά, όσο και από τη δυνατότητα επαναχρησιμοποίησής τους. Έτσι, το σύστημα περιλαμβάνει έναν μηχανισμό αντιστοίχισης ερωτημάτων σε τμήματα κώδικα με βάση τη σύνταξη (syntax-aware), καθώς και ένα μοντέλο που αξιολογεί την επαναχρησιμοποιησιμότητα κάθε τμήματος.

**Κεφάλαιο 10. Αξιολόγηση της Επαναχρησιμοποιησιμότητας Κώδικα**

Σε αυτό το κεφάλαιο παρουσιάζουμε έναν καινοτόμο τρόπο αξιολόγησης της επαναχρησιμοποίησης των τμημάτων κώδικα, χρησιμοποιώντας την δημοτικότητα και την πληροφορία επαναχρησιμοποίησης για την κατασκευή ενός συνόλου από εξακριβωμένες τιμές ποιότητας (ground truth).

**Μέρος V: Συμπεράσματα και Μελλοντική Εργασία**

Αυτό το μέρος περιέχει τα συμπεράσματα της διατριβής καθώς και ιδέες για πιθανή μελλοντική εργασία. Περιλαμβάνει τα ακόλουθα κεφάλαια:

**Κεφάλαιο 11. Συμπεράσματα**

Σε αυτό το κεφάλαιο παρουσιάζονται τα συμπεράσματα της διατριβής, ενώ επίσης συνοψίζεται η συνεισφορά της στις σχετικές ερευνητικές περιοχές.

**Κεφάλαιο 12. Μελλοντική Εργασία**

Αυτό το κεφάλαιο περιλαμβάνει ιδέες για πιθανές επεκτάσεις σε όλους τους τομείς συνεισφοράς αυτής της διατριβής.

## 1.5 Συμπληρωματικό υλικό

Ένα σημαντικό ζήτημα για κάθε επιστημονική εργασία είναι η δυνατότητα αναπαραγωγής (reproducibility) των αποτελεσμάτων. Ως εκ τούτου, για αυτή τη διατριβή, θα θέλαμε να παρέχουμε τα δεδομένα που απαιτούνται για την αναπαραγωγή των ερευνητικών αποτελεσμάτων μας, τα οποία ελπίζουμε ότι θα βοηθήσουν άλλους ερευνητές να σημειώσουν πρόοδο στις σχετικές επιστημονικές περιοχές. Ωστόσο, η παροχή δεδομένων μέσω πολλών διαδικτυακών συνδέσμων είναι μια επισφαλής πρακτική, καθώς οι συνδέσεις σε διαδικτυακά εργαλεία σε ιστοσελίδες με σύνολα δεδομένων, κ.α., συχνά καταργούνται και τα δεδομένα δεν είναι πια διαθέσιμα. Αντί αυτού, αποφασίσαμε να δημιουργήσουμε μια συνολική ιστοσελίδα που συγκεντρώνει όλα τα δεδομένα που σχετίζονται με αυτή τη διατριβή, όπως π.χ. σύνολα δεδομένων, μοντέλα, εργαλεία (σε μορφή εκτελέσιμου και/ή πηγαίου κώδικα) κ.λπ. Η ιστοσελίδα είναι διαθέσιμη στο:

<https://thdiaman.github.io/phdthesis/>

# 2

## Θεωρητικό Υπόβαθρο και Βιβλιογραφία

### 2.1 Τεχνολογία Λογισμικού

#### 2.1.1 Επισκόπηση και Δραστηριότητες της Τεχνολογίας Λογισμικού

Σε αυτό το υποκεφάλαιο παρέχουμε το απαραίτητο υπόβαθρο για την περιοχή της Τεχνολογίας Λογισμικού. Σύμφωνα με τον οργανισμό IEEE [18], η Τεχνολογία Λογισμικού ορίζεται ως:

Η εφαρμογή μιας συστηματικής, πειθαρχημένης, μετρήσιμης προσέγγισης στην ανάπτυξη, λειτουργία και συντήρηση του λογισμικού· δηλαδή, η εφαρμογή της μηχανικής στο λογισμικό.

Αν και κάπως γενικός, ο ορισμός αυτός περιλαμβάνει ορισμένες από τις σημαντικότερες αρχές του πεδίου. Καταρχάς, είναι σημαντικό να σημειωθεί ότι η διαδικασία ανάπτυξης λογισμικού πρέπει να είναι συστηματική, να ακολουθεί ορισμένους κανόνες και να είναι μετρήσιμη (measurable) ώστε να είναι εφικτό να πραγματοποιούνται βελτιώσεις. Το δεύτερο μέρος του ορισμού είναι εξίσου σημαντικό· η παραγωγή λογισμικού δεν περιορίζεται στο στάδιο της ανάπτυξης (development), αλλά αφορά επίσης την ορθή λειτουργία (operation), τη συντήρηση (maintenance) και την εξέλιξη (evolution) του λογισμικού.

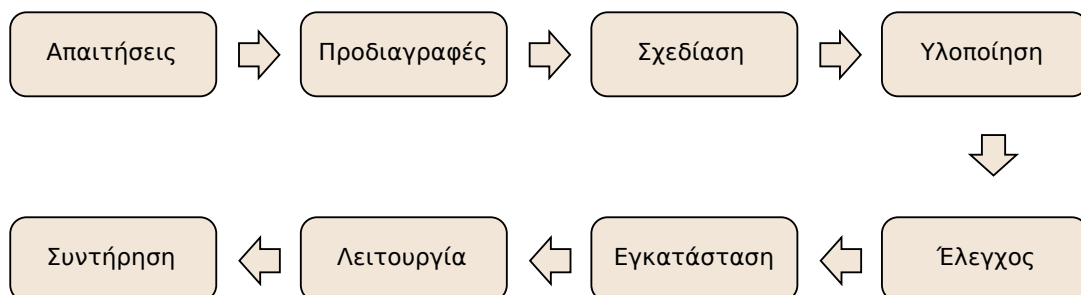
Κατά τα πρώτα βήματα στην περιοχή της Τεχνολογίας Λογισμικού, που τοποθετούνται στα τέλη της δεκαετίας του 1960<sup>1</sup>, ήταν σαφές ότι ένας οργανωμένος τρόπος δημιουργίας λογισμικού ήταν απαραίτητος· οι μεμονωμένες προσεγγίσεις στην ανάπτυξη λογισμικού δε επέτρεπαν την κλιμάκωση (scaling) σε πραγματικά συστήματα. Ως εκ τούτου, το πεδίο γνώρισε ραγδαία και συνεχή ανάπτυξη με την εισαγωγή διάφορων μεθοδολογιών για την ανάπτυξη λογισμικού. Παρόλο που πράγματι υπάρχουν πολλές διαφορετικές μεθοδολογίες, οι βασικές δραστηριότητες της διαδικασίας ανάπτυξης λογισμικού είναι κοινές μεταξύ τους. Σύμφωνα με τον Sommerville [2], κάθε μεθοδολογία ανάπτυξης λογισμικού περιλαμβάνει τις δραστηριότητες της εξαγωγής προδιαγραφών (specifications), του σχεδιασμού (design)

---

<sup>1</sup>Σύμφωνα με τον Sommerville [2], ο όρος προτάθηκε για πρώτη φορά το 1968 σε μια διάσκεψη που διοργάνωσε το NATO.

του λογισμικού και της υλοποίησής του (implementation), της επικύρωσης (validation) και της εξέλιξής του (evolution). Η εξαγωγή προδιαγραφών αναφέρεται στην περιγραφή της αναμενόμενης λειτουργικότητας του λογισμικού, ενώ ο σχεδιασμός και η υλοποίηση αφορούν τον καθορισμό του τρόπου με τον οποίο θα επιτευχθεί η λειτουργικότητα αυτή και την ανάπτυξη του λογισμικού σε κάποια γλώσσα προγραμματισμού, αντίστοιχα. Η επικύρωση πραγματοποιείται για να διασφαλιστεί ότι το λογισμικό παρέχει πράγματι τη λειτουργικότητα που απαιτείται, ενώ η εξέλιξη του λογισμικού επικεντρώνεται στον τρόπο με τον οποίο το λογισμικό θα συνεχίσει να λειτουργεί, καθώς πιθανά αλλάζουν οι απαιτήσεις ή οι σχετικές υποδομές (infrastructure).

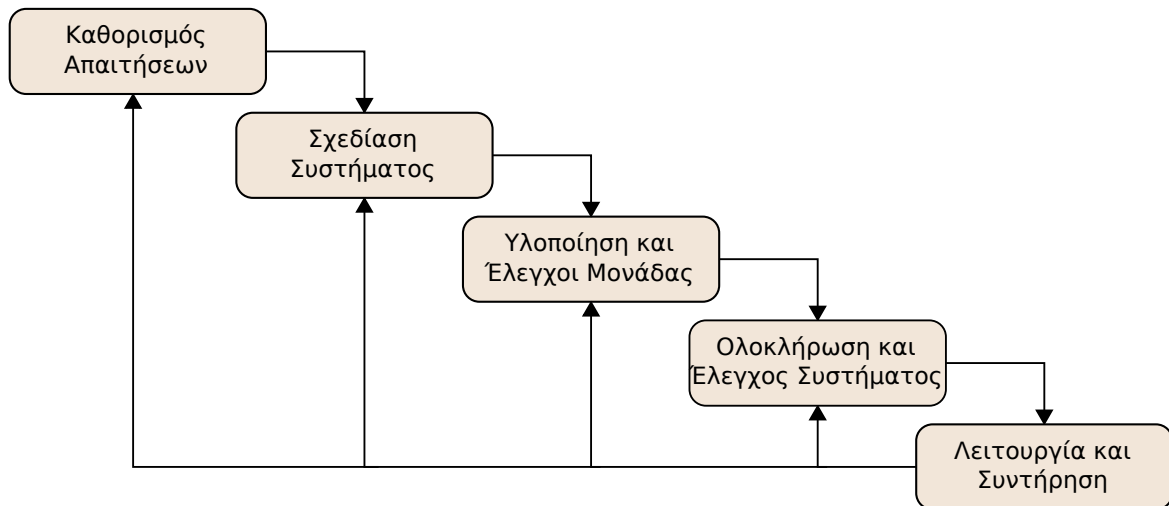
Στο Σχήμα 2.1 απεικονίζονται αναλυτικότερα οι βασικές δραστηριότητες/φάσεις της Τεχνολογίας Λογισμικού, όπως συνήθως ορίζονται από την τρέχουσα βιβλιογραφία [2, 15, 19]. Αυτές οι δραστηριότητες είναι λίγο έως πολύ παρούσες σε όλα τα συστήματα λογισμικού. Για την ανάπτυξη οποιουδήποτε έργου, θα πρέπει αρχικά να καθοριστούν οι απαιτήσεις του (requirements), να εξαχθούν με βάση αυτές οι προδιαγραφές (specifications) και στη συνέχεια να σχεδιαστεί η αρχιτεκτονική του (design). Κατόπιν, το έργο υλοποιείται (implementation) και πραγματοποιείται έλεγχος (testing), και στη συνέχεια εγκαθίσταται (deployment). Ακόμη και μετά την ολοκλήρωσή του, το έργο πρέπει να συντηρείται (maintenance) για να συνεχίσει να είναι χρήσιμο. Σημειώνουμε εδώ ότι έργα που δε συντηρούνται τείνουν να απορρίπτονται· αυτό συμβαίνει για διάφορους λόγους, π.χ. η έλλειψη ενημερώσεων ασφαλείας μπορεί να κάνει το σύστημα ευάλωτο σε εξωτερικές απειλές.



Σχήμα 2.1: Βασικές Δραστηριότητες στην Τεχνολογία Λογισμικού

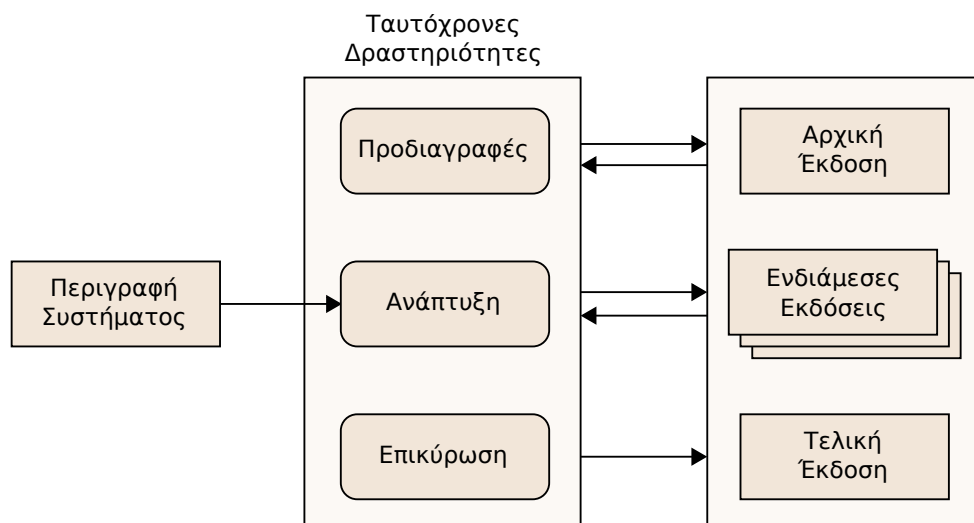
### 2.1.2 Μοντέλα Ανάπτυξης Λογισμικού

Οι δραστηριότητες λογισμικού (software activities) και ο τρόπος με τον οποίο συνδυάζονται συνιστούν αυτό που ονομάζουμε διαδικασία ανάπτυξης λογισμικού (software development process). Η πρόκληση σε κάθε περίπτωση είναι να επιλεγεί ένα κατάλληλο μοντέλο ανάπτυξης λογισμικού (software process model), ένας τρόπος δηλαδή συνδυασμού των δραστηριοτήτων που αναφέρθηκαν, με σκοπό την ανάπτυξη λογισμικού υψηλής ποιότητας. Με το πέρασμα του χρόνου, έχουν προταθεί διάφορα μοντέλα ανάπτυξης λογισμικού. Το πρώτο χρονολογικά μοντέλο είναι το *μοντέλο καταρράκτη* (waterfall model), που απεικονίζεται στο Σχήμα 2.2. Σύμφωνα με αυτό το μοντέλο, όλες οι δραστηριότητες προγραμματίζονται εκ των προτέρων και κάθε δραστηριότητα πρέπει να ολοκληρωθεί για να ακολουθήσει η επόμενη.



Σχήμα 2.2: Μοντέλο Καταρράκτη

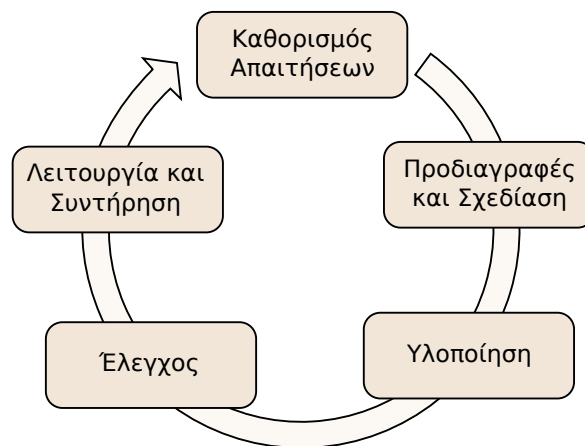
Το μοντέλο καταρράκτη είναι αποτελεσματικό σε περιπτώσεις όπου όλες οι φάσεις του υπό ανάπτυξη λογισμικού είναι ξεκάθαρες, π.χ. όταν οι απαιτήσεις είναι καλά καθορισμένες και δεν υπόκεινται σε αλλαγές. Ωστόσο, αυτό δεν συμβαίνει πάντα. Επιπλέον, με αυτό το μοντέλο ανάπτυξης, το λογισμικό δε χτίζεται κλιμακωτά, συνεπώς δεν είναι συνήθως εφικτό οι τελικοί χρήστες να ελέγχουν αν το έργο καλύπτει όλα όσα επιθυμούν (και τα οποία είχαν καθοριστεί στις αρχικές απαιτήσεις) παρά μόνο μετά από τη διαδικασία ανάπτυξης του λογισμικού. Για αυτούς τους λόγους, σχεδιάστηκαν και άλλα μοντέλα ανάπτυξης λογισμικού, όπως το επαναληπτικό μοντέλο (*iterative model*) [2], που απεικονίζεται στο Σχήμα 2.3. Το επαναληπτικό μοντέλο υπαγορεύει ένα σύνολο από δραστηριότητες που επαναλαμβάνονται για την κατασκευή διαδοχικών εκδόσεων λογισμικού. Κάθε φορά που παρέχονται νέες απαιτήσεις (ή ανανεώνονται οι υπάρχουσες), η ομάδα ανάπτυξης διαμορφώνει τις προδιαγραφές, αναπτύσσει τη νέα λειτουργικότητα και επικυρώνει το λογισμικό ώστε να παραχθεί μια νέα έκδοση. Μετά από αρκετές επαναλήψεις, παράγεται η τελική έκδοση του λογισμικού.



Σχήμα 2.3: Επαναληπτικό Μοντέλο

Το επαναληπτικό μοντέλο είναι ένα από τα πιο δημοφιλή μοντέλα σήμερα [2], πιθανώς επειδή εμπεριέχει την αυστηρή λογική των παραδοσιακών μοντέλων (όπως το μοντέλο καταρράκτη), παρέχοντας ωστόσο σχετική ευελιξία. Ένα ακόμα θετικό αυτού του μοντέλου είναι ότι ακολουθώντας το παράγονται ενδιάμεσες εκδόσεις του λογισμικού, που μπορούν να δοθούν στους χρήστες ώστε να αναφέρουν χρήσιμα σχόλια. Ωστόσο, αυτή η συνδυαστική λογική του μοντέλου μπορεί να οδηγήσει και σε ορισμένα μειονεκτήματα, το σημαντικότερο από τα οποία είναι ότι η ποιότητα του λογισμικού συνήθως τείνει να υποβαθμίζεται μεταξύ διαδοχικών εκδόσεων. Αυτό έχει οδηγήσει στην ανάπτυξη ακόμα πιο ευέλικτων μοντέλων ανάπτυξης λογισμικού που απομακρύνονται από την παραδοσιακή λογική της σχεδίασης ολόκληρου του λογισμικού πριν την ανάπτυξη του.

Αυτές οι σχετικά πιο σύγχρονες πρακτικές ανάπτυξης είναι ευρέως γνωστές ως *ευέλικτες (agile) προσεγγίσεις* στην ανάπτυξη λογισμικού. Παρότι οι πρώτες ευέλικτες προσεγγίσεις είχαν προταθεί από τη δεκαετία του '90 [2], υιοθετήθηκαν από τη βιομηχανία λογισμικού κυρίως στις αρχές του 21ου αιώνα<sup>2</sup>. Σύμφωνα με το μανιφέστο για την ευέλικτη ανάπτυξη λογισμικού<sup>3</sup>, βασικός στόχος είναι η έγκαιρη και συνεχής παράδοση χρήσιμου λογισμικού ώστε να καλύπτονται οι ανάγκες των χρηστών, ενώ ταυτόχρονα να είναι εφικτή η προσαρμογή του λογισμικού όταν μεταβάλλονται οι απαιτήσεις. Σε ένα ευέλικτο μοντέλο ανάπτυξης λογισμικού, το υπό ανάπτυξη προϊόν συνήθως χωρίζεται σε διαφορετικά σύνολα χαρακτηριστικών (*features*), τα οποία αναπτύσσονται ξεχωριστά το ένα μετά το άλλο. Συγκεκριμένα, για κάθε χαρακτηριστικό ή σύνολο χαρακτηριστικών πραγματοποιείται ένας κύκλος δραστηριοτήτων [15], που απεικονίζεται στο Σχήμα 2.4, ενώ είναι αρκετά συχνή η παραγωγή νέων εκδόσεων του λογισμικού.



Σχήμα 2.4: Ευέλικτο Μοντέλο

Σημειώνουμε ότι σε αυτή την ενότητα δεν καλύφθηκαν όλα τα πιθανά μοντέλα ανάπτυξης λογισμικού<sup>4</sup>. Εστιάζουμε κυρίως στις διάφορες δραστηριότητες που ορίζονται στο

<sup>2</sup>Συγκεκριμένα, οι ευέλικτες προσεγγίσεις αρχικά προτιμήθηκαν κυρίως από τις Μικρές και Μεσαίες Επιχειρήσεις (Small-to-Medium Enterprises - SMEs) και τις νεοφυείς επιχειρήσεις (startups), ωστόσο πρόσφατα άρχισαν να επιλέγονται και στην αγορά των μεγάλων επιχειρήσεων [20].

<sup>3</sup><http://agilemanifesto.org/>

<sup>4</sup>Ο αναγνώστης παραπέμπεται στο [15] για μια εκτεταμένη ανασκόπηση των μοντέλων ανάπτυξης λογισμικού καθώς και μια συζήτηση για την εφαρμογή τους είτε μεμονωμένα είτε συνδυαστικά (υβριδικές προσεγγίσεις).

Σχήμα 2.1 (π.χ. καθορισμός απαιτήσεων, υλοποίηση πηγαίου κώδικα, κ.λπ.) και στις συνδέσεις μεταξύ των διαδοχικών δραστηριοτήτων (π.χ. εξαγωγή των προδιαγραφών από τις απαιτήσεις). Διαχωρίζουμε τις δραστηριότητες σε τρεις κατηγορίες που αλληλοσυνδέονται, οι οποίες φαίνονται στο Σχήμα 2.5, και επιδιόκουμε να βοηθήσουμε τις ομάδες ανάπτυξης στην κατασκευή νέων έργων λογισμικού, ανεξάρτητα από το μοντέλο ανάπτυξης που μπορεί να έχουν επιλέξει.



Σχήμα 2.5: Κατηγορίες Δραστηριοτήτων Τεχνολογίας Λογισμικού

Αναμφισβήτητα, το ρίσκο κατά την ανάπτυξη λογισμικού μπορεί να ελαχιστοποιηθεί με πολλούς τρόπους. Σε αυτή τη διατριβή, εστιάζουμε σε δύο άξονες για την ελαχιστοποίηση αυτού του ρίσκου: στη μοντελοποίηση και επαναχρησιμοποίηση (reuse) των υφιστάμενων λύσεων στην περιοχή της τεχνολογίας λογισμικού, και στην αξιολόγηση τόσο του παραγόμενου λογισμικού, όσο και της διαδικασίας ανάπτυξής του. Αυτοί οι δύο άξονες αντιστοιχούν στις περιοχές της επαναχρησιμοποίησης λογισμικού (*software reuse*) και της αξιολόγησης ποιότητας λογισμικού (*software quality assessment*), οι οποίες αναλύονται στα ακόλουθα υποκεφάλαια.

## 2.2 Επαναχρησιμοποίηση Λογισμικού

### 2.2.1 Επισκόπηση Επαναχρησιμοποίησης

Στο πλαίσιο της παρούσας διατριβής, η Επαναχρησιμοποίηση Λογισμικού (Software Reuse) ορίζεται ως [21]:

Η συστηματική αξιοποίηση υπαρχόντων αντικειμένων λογισμικού κατά τη διαδικασία κατασκευής νέου λογισμικού ή η ενσωμάτωση υπαρχόντων αντικειμένων λογισμικού σε νέο λογισμικό.

Ο όρος “αντικείμενο λογισμικού” (software artifact) στον παραπάνω ορισμό δεν περιορίζεται σε τμήματα πηγαίου κώδικα (source code components), αλλά αντιθέτως καλύπτει και απαιτήσεις λογισμικού (software requirements), πληροφορίες σχεδίασης (design), τεκμηρίωση (documentation) κ.α. Ένα ακόμα σημαντικό στοιχείο σε αυτόν τον ορισμό είναι ότι τα αντικείμενα προς επαναχρησιμοποίηση είναι πιθανό είτε να χρησιμοποιηθούν ως έχουν είτε να αξιοποιηθούν για να παράγουν χρήσιμη γνώση για την παραγωγή νέου λογισμικού. Συνεπώς, η επαναχρησιμοποίηση δεν περιορίζεται στην εύρεση και ενσωμάτωση τμημάτων στον πηγαίο κώδικα. Σε αυτό το υποκεφάλαιο θα επικεντρωθούμε αρχικά στα οφέλη της επαναχρησιμοποίησης και στη συνέχεια θα αναφερθούμε σε πιθανές πηγές δεδομένων και σε τομείς εφαρμογής της επαναχρησιμοποίησης λογισμικού.

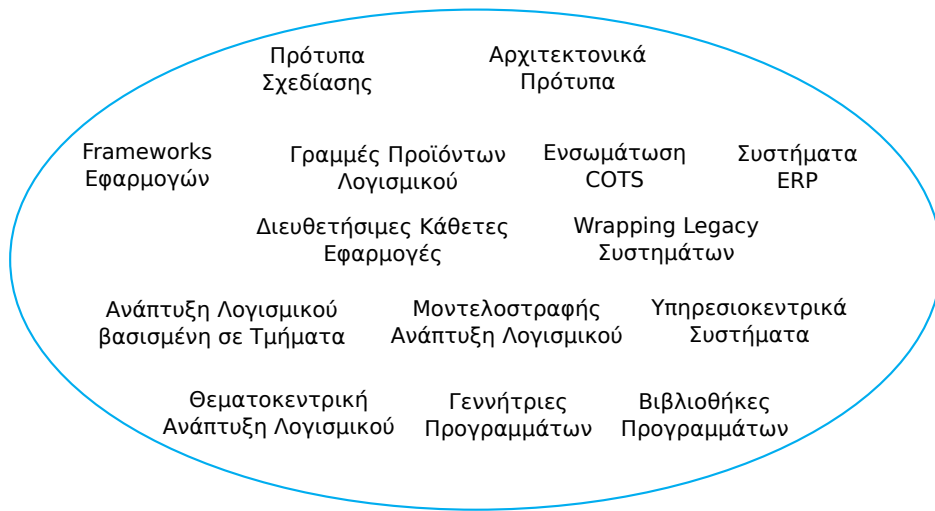
Από μια σχετικά απλοϊκή σκοπιά, η επαναχρησιμοποίηση σημαίνει να χρησιμοποιούμε ήδη υπάρχουσες γνώσεις και δεδομένα αντί να ανακαλύπτουμε εκ νέου τον τροχό. Προφανώς, αυτό έχει πολλά πλεονεκτήματα [22]. Το πρώτο και ίσως το σημαντικότερο όφελος της επαναχρησιμοποίησης είναι η μείωση του χρόνου και της απαιτούμενης ανθρωποπροσπάθειας. Επαναχρησιμοποιώντας υπάρχοντα τμήματα λογισμικού, η ομάδα ανάπτυξης μπορεί να παράγει λογισμικό γρηγορότερα και ενώ το τελικό προϊόν θα έχει ταχύτερο χρόνο διάθεσης στην αγορά (time-to-market). Οι ίδιοι οι προγραμματιστές θα έχουν περισσότερο χρόνο να αφιερώσουν σε άλλες πτυχές του έργου, οι οποίες επηρεάζονται επίσης από την επαναχρησιμοποίηση, όπως η τεκμηρίωση. Το δεύτερο σημαντικό κέρδος από την επαναχρησιμοποίηση λογισμικού είναι η διασφάλιση ότι το τελικό προϊόν είναι υψηλής ποιότητας. Η ανάπτυξη λογισμικού είναι γενικά μια αρκετά πολύπλοκη διαδικασία, καθώς η δημιουργία και ο λεπτομερής έλεγχος (testing) ενός στοιχείου μπορεί να είναι μη εφικτά μέσα σε συγκεκριμένα χρονικά πλαίσια. Αντιθέτως, η επαναχρησιμοποίηση υπάρχουσας λειτουργικότητας παρέχει έναν τρόπο διαχωρισμού μεταξύ του υπό ανάπτυξη λογισμικού και του εξωτερικού εξαρτήματος, καθώς η παραγωγή και η συντήρηση του τελευταίου είναι εκτός των καθηκόντων της ομάδας ανάπτυξης. Ακόμη και αν το στοιχείο δεν ανήκει σε κάποιον τρίτο (third-party), η συντήρηση ενός συνόλου διαφορετικών, σαφώς διαχωρισμένων τμημάτων είναι σχεδόν πάντα προτιμότερη από τη συντήρηση ενός μεγάλου και πολύπλοκου συστήματος λογισμικού. Ασφαλώς, η επαναχρησιμοποίηση ορισμένων τμημάτων μπορεί να εμπεριέχει κινδύνους για την ποιότητα του τελικού προϊόντος. Ωστόσο, σε αυτό το υποκεφάλαιο επικεντρωνόμαστε σε ορθά σενάρια επαναχρησιμοποίησης, ενώ το πρόβλημα του κατά πόσο ορισμένα τμήματα λογισμικού είναι κατάλληλα για επαναχρησιμοποίηση αναλύεται στο επόμενο υποκεφάλαιο.

Καθώς εμβαθύνουμε στην έννοια της επαναχρησιμοποίησης, τίθεται το ερώτημα ποιες είναι οι μέθοδοι επαναχρησιμοποίησης ή, με άλλα λόγια, τι θεωρείται επαναχρησιμοποίηση και τι όχι. Μια ενδιαφέρουσα οπτική είναι αυτή του Kueger [12], που υποστηρίζει ότι η επαναχρησιμοποίηση εφαρμόζεται παντού, από μεταγλωττιστές (compilers) και εργαλεία αυτόματης παραγωγής κώδικα (code generators) μέχρι βιβλιοθήκες (software libraries) και πρότυπα σχεδίασης λογισμικού (design patterns). Η βασική ιδέα, ωστόσο, των προτάσεων του Kueger είναι ότι πρέπει να εξετάσουμε το ενδεχόμενο της επαναχρησιμοποίησης ως πρακτική με κεντρικό ρόλο (standard practice) στην ανάπτυξη λογισμικού. Σήμερα, έχουν πράγματι αναπτυχθεί ορισμένες μεθοδολογίες προς την κατεύθυνση αυτή. Όπως σχολιάζει ο Sommerville, η επαναχρησιμοποίηση ως μέρος της ανάπτυξης επιλέγεται και εφαρμόζεται ανεξάρτητα από την επιλεγμένη διαδικασία ανάπτυξης λογισμικού<sup>5</sup>. Οι μεθοδολογίες που έχουν αναπτυχθεί ουσιαστικά διευκολύνουν την ενσωμάτωση της επαναχρησιμοποίη-

<sup>5</sup>Μια πιο τυποποιημένη προσέγγιση είναι αυτή της ανάπτυξης λογισμικού προσανατολισμένης στην επαναχρησιμοποίηση (reuse-oriented software engineering), η οποία περιλαμβάνει τα εξής στάδια: τον καθορισμό των απαιτήσεων, τον έλεγχο εάν μπορούν να καλυφθούν κάποιες απαιτήσεις από την επαναχρησιμοποίηση υπάρχοντων τμημάτων και τέλος τη σχεδίαση και ανάπτυξη του συστήματος με επαναχρησιμοποίηση (που περιλαμβάνει τη σχεδίαση των εισόδων και των εξόδων των τμημάτων και την ενσωμάτωσή τους). Ωστόσο, η επαναχρησιμοποίηση ως πρακτική ακολουθείται ήδη από διάφορες ομάδες ανάπτυξης και πλαισιώνεται από επιλεγμένο μοντέλο ανάπτυξης, ανεξάρτητα από τον πιο παραδοσιακό ή πιο σύγχρονο χαρακτήρα του. Ως εκ τούτου, υποστηρίζουμε ότι δεν χρειάζεται να αναγκάσουμε τις ομάδες ανάπτυξης να χρησιμοποιήσουν μια διαφορετική μεθοδολογία και άρα δεν εστιάζουμε στην ανάπτυξη λογισμικού προσανατολισμένη στην επαναχρησιμοποίηση ως τυποποιημένη προσέγγιση. Αντιθέτως, παρουσιάζουμε εδώ μια σειρά από μεθοδολογίες και εξετάζουμε τις δυνατότητες που προκύπτουν από τη χρήση δεδομένων τεχνολογίας λογισμικού προκειμένου να ενσωματωθεί πρακτικά η επαναχρησιμοποίηση ως φιλοσοφία στην ανάπτυξη λογισμικού.



σης λογισμικού σε όλους τους άξονες της τεχνολογίας λογισμικού. Έτσι, σχηματίζεται αυτό που ονομάζουμε *τοπίο επαναχρησιμοποίησης (reuse landscape)* [2], το οποίο απεικονίζεται στο Σχήμα 2.6. Το τοπίο αυτό περιλαμβάνει μεθοδολογίες επαναχρησιμοποίησης που αναφέρονται τόσο πριν, όσο και κατά τη διάρκεια της ανάπτυξης νέου λογισμικού. Ορισμένες χαρακτηριστικές μεθοδολογίες είναι η ανάπτυξη λογισμικού βασισμένη σε τμήματα (component-based software engineering), η χρήση πρότυπων σχεδίασης (design patterns) και/ή αρχιτεκτονικών πρότυπων (architectural patterns) για την κατασκευή νέου λογισμικού, η αξιοποίηση υπάρχουσας λειτουργικότητας μέσω frameworks ή βιβλιοθηκών (libraries), κ.λπ.



Σχήμα 2.6: Τοπίο Επαναχρησιμοποίησης

### 2.2.2 Δεδομένα Τεχνολογίας Λογισμικού

Στο πλαίσιο της παρούσας διατριβής, εστιάζουμε στην επαναχρησιμοποίηση από τη σκοπιά της αξιοποίησης των δεδομένων που είναι διαθέσιμα σε κάθε περίπτωση. Τα δεδομένα στην τεχνολογία λογισμικού προέρχονται από διάφορες πηγές [23]. Στο Σχήμα 2.7 απεικονίζονται ενδεικτικά κάποιες κατηγορίες δεδομένων που είναι σχετικά με την τεχνολογία λογισμικού. Σημειώνουμε ότι η μορφή των δεδομένων του Σχήματος 2.7 ποικίλει. Για παράδειγμα, οι απαιτήσεις ενδέχεται να παρέχονται σε ελεύθερο κείμενο (free text), ως ιστορίες χρηστών (user stories), ως διαγράμματα UML, κ.α. Επίσης, τα δεδομένα αυτά μπορεί να μην είναι όλα διαθέσιμα για κάθε έργο λογισμικού, π.χ. κάποια έργα λογισμικού μπορεί να μην έχουν συστήματα εντοπισμού σφαλμάτων ή λίστες αλληλογραφίας. Μια άλλη σημαντική παρατήρηση που αφορά τη φύση των δεδομένων είναι ότι κάποια από αυτά, όπως ο πηγαίος κώδικας ή οι απαιτήσεις, είναι διαθέσιμα απευθείας από το έργο λογισμικού, ενώ άλλα, όπως οι μετρήσεις ποιότητας (quality metrics), θα πρέπει να εξαχθούν χρησιμοποιώντας κατάλληλα εργαλεία. Σε κάθε περίπτωση, οι κατηγορίες δεδομένων που αναλύθηκαν είναι αυτές που οι προγραμματιστές θα πρέπει να είναι σε θέση να κατανοήσουν και να αξιοποιήσουν, προκειμένου να εφαρμόσουν πρακτικές επαναχρησιμοποίησης και επομένως να βελτιώσουν την διαδικασία ανάπτυξης και το προϊόν τους.



Σχήμα 2.7: Δεδομένα Τεχνολογίας Λογισμικού<sup>6</sup>

Με την πρώτη ματιά, όλες οι παραπάνω πηγές δεδομένων μπορεί να φαίνεται δύσκολο να συγκεντρωθούν. Επιπλέον, παρόλο που μια ομάδα ανάπτυξης μπορεί να διατηρεί μια βάση δεδομένων για τα δικά της έργα, το να την εμπλουτίσει και με δεδομένα τρίτων (third-party), τα οποία ομολογουμένως στην επαναχρησιμοποίηση λογισμικού μπορεί να είναι πολύ χρήσιμα, είναι δύσκολο. Ωστόσο, στη σύγχρονη τεχνολογία λογισμικού, υπάρχει πλήθος εργαλείων για τη συλλογή αυτών των δεδομένων, ενώ τα ίδια τα δεδομένα βρίσκονται σε αφθονία σε online αποθετήρια. Συνεπώς, ο τομέας της τεχνολογίας λογισμικού έχει αλλάξει, καθώς οι προγραμματιστές σήμερα χρησιμοποιούν το διαδίκτυο για όλες τις προκλήσεις που μπορεί να προκύψουν κατά τη διάρκεια της ανάπτυξης λογισμικού [25]. Μια πρόσφατη έρευνα έδειξε ότι σχεδόν το ένα πέμπτο του χρόνου των προγραμματιστών δαπανάται για την online αναζήτηση λύσεων στα προβλήματα που προκύπτουν στο υπό ανάπτυξη έργο [26]. Αναφέρουμε περαιτέρω ότι υπάρχει πλήθος πηγών στις οποίες οι προγραμματιστές αναζητούν λύσεις [27], όπως π.χ. τεχνικά ιστολόγια (technical blogs), κοινότητες ερωταπαντήσεων (question-answering communities), forums, υπηρεσίες φιλοξενίας κώδικα (code hosting services), ιστοσελίδες με παραδείγματα κώδικα, κ.λπ.

Καθώς, λοιπόν, πλήθος υπηρεσιών έχουν αναπτυχθεί προς την κατεύθυνση αυτών των νέων πρακτικών τεχνολογίας λογισμικού, μπορούμε με ασφάλεια να συμπεράνουμε ότι η κοινωνική συγγραφή κώδικα (social coding) και γενικότερα η συνεργατική ανάπτυξη λογισμικού (collaborative software development) αποτελούν μια νέα τάση στο λογισμικό. Στις ακόλουθες παραγράφους, αναλύεται ένα ενδεικτικό σύνολο αυτών των υπηρεσιών.

**Υπηρεσίες Φιλοξενίας Κώδικα** Αναμφισβήτητα, ο πιο σημαντικός τύπος υπηρεσιών είναι αυτός των υπηρεσιών φιλοξενίας κώδικα (code hosting services), καθώς είναι αυτές που επιτρέπουν στους προγραμματιστές να ανεβάσουν τον πηγαίο κώδικά τους στο διαδίκτυο.

<sup>6</sup>Σημειώστε ότι εστιάζουμε κυρίως σε δεδομένα τεχνολογίας λογισμικού που είναι διαθέσιμα στο διαδίκτυο. Υπάρχουν και άλλες κατηγορίες δεδομένων που είναι άξια ανάλυσης, όπως για παράδειγμα αυτά που συνιστούν το *τοπίο των πληροφοριών* (landscape of information) με το οποίο έρχεται αντιμέτωπος ένας προγραμματιστής όταν ενταχθεί σε ένα έργο [24]. Συγκεκριμένα, εκτός από τον πηγαίο κώδικα του έργου, απαιτείται εξοικείωση με την αρχιτεκτονική (architecture) του έργου, τις διαδικασίες λογισμικού που ακολουθούνται, τα σχετικά APIs, το περιβάλλον ανάπτυξης (development environment), κ.λπ.

Ως εκ τούτου, οι υπηρεσίες φιλοξενίας κώδικα παρέχουν ουσιαστικά την αρχική πληροφορία που μπορεί να χρησιμοποιηθεί για διάφορους σκοπούς. Στον Πίνακα 2.1 παρουσιάζονται ενδεικτικά ορισμένες δημοφιλείς υπηρεσίες φιλοξενίας κώδικα<sup>7</sup>. Όπως μπορούμε να δούμε

Πίνακας 2.1: Υπηρεσίες Φιλοξενίας Πηγαίου Κώδικα

Υπηρεσία	URL	# Χρήστες	# Έργα	Ίδρυση
SourceForge	<a href="https://sourceforge.net/">https://sourceforge.net/</a>	3,700,000	500,000	1999
GNU Savannah	<a href="https://savannah.gnu.org/">https://savannah.gnu.org/</a>	80,000	4,000	2001
Launchpad	<a href="https://launchpad.net/">https://launchpad.net/</a>	4,000,000	40,000	2004
GitHub	<a href="https://github.com/">https://github.com/</a>	24,000,000	69,000,000	2008
Bitbucket	<a href="https://bitbucket.org/">https://bitbucket.org/</a>	5,000,000	-	2008
GitLab	<a href="https://gitlab.com/">https://gitlab.com/</a>	100,000	550,000	2011

σε αυτόν τον Πίνακα, η εποχή-σταθμός για τη δημιουργία υπηρεσιών φιλοξενίας κώδικα μπορεί να τοποθετηθεί γύρω στο 2008, όταν το GitHub και το Bitbucket προστέθηκαν στην αγορά του ήδη μεγάλου SourceForge· σήμερα οι τρεις αυτές υπηρεσίες μαζί φιλοξενούν τα έργα περισσότερων από 30 εκατομμυρίων προγραμματιστών. Εάν επιπλέον εξετάσουμε τα στατιστικά στοιχεία για τον συνολικό αριθμό των έργων στις υπηρεσίες αυτές, μπορούμε εύκολα να ισχυριστούμε ότι υπάρχουν εκατομμύρια έργα που φιλοξενούνται στο διαδίκτυο σε κάποια υπηρεσία φιλοξενίας πηγαίου κώδικα.

**Ιστοσελίδες Ερωταπαντήσεων** Έχουν καθιερωθεί αρκετές online κοινότητες, στις οποίες οι προγραμματιστές μπορούν να επικοινωνήσουν τα προβλήματα που προκύπτουν κατά την ανάπτυξη λογισμικού και να βρουν πιθανές λύσεις. Ακόμη και στην περίπτωση που οι προγραμματιστές δεν χρησιμοποιούν απευθείας αυτές τις υπηρεσίες, είναι πολύ πιθανό να ανακατευθύνονται σε αυτές από μηχανές αναζήτησης καθώς τα ερωτήματά τους μπορεί να είναι παρόμοια με αυτά που έχουν ήδη αναρτηθεί και απαντηθεί από άλλους προγραμματιστές. Η πιο δημοφιλής από αυτές τις υπηρεσίες είναι το Stack Overflow<sup>8</sup>, μια συλλογική κοινότητα προγραμματιστών (collective programmer community<sup>9</sup>) που ιδρύθηκε το 2008. Με περισσότερους από 8 εκατομμύρια εγγεγραμμένους χρήστες και πάνω από 50 εκατομμύρια προγραμματιστές να επισκέπτονται τη σχετική ιστοσελίδα κάθε μήνα, το Stack Overflow παρέχει μια τεράστια συλλογή απαντήσεων για ερωτήματα που σχετίζονται με τον προγραμματισμό εφαρμογών. Το Stack Overflow μάλιστα αποτελεί μέρος του δικτύου Stack Exchange, που προσφέρει σήμερα πολλές διαφορετικές υπηρεσίες ερωταπαντήσεων<sup>10</sup>, αρκετές από τις οποίες είναι αφιερωμένες στη διαδικασία ανάπτυξης λογισμικού. Ενδεικτικά αναφέρουμε ότι υπάρχουν σελίδες γενικά για την τεχνολογία λογισμικού<sup>11</sup>, για τη διαχεί-

<sup>7</sup>Οι αριθμοί σε αυτόν τον Πίνακα δίνουντα προσεγγιστικά. Σημειώστε, επίσης, ότι αυτά τα στοιχεία είναι έγκυρα κατά τη στιγμή της συγγραφής της διατριβής, καθώς ενδέχεται να υπόκεινται σε αλλαγές αρκετά συχνά. Για πιο πλήρεις και ενημερωμένες στατιστικές πληροφορίες σχετικά με τις υπηρεσίες φιλοξενίας κώδικα, ο αναγνώστης παραπέμπεται στο ηλεκτρονικό άρθρο [https://en.wikipedia.org/wiki/Comparison\\_of\\_source\\_code\\_hosting\\_facilities](https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities).

<sup>8</sup><https://stackoverflow.com/>

<sup>9</sup>Ο όρος αυτός αποδίδεται στον Jeff Atwood, που είναι ο ένας από τους δύο δημιουργούς του Stack Overflow (ο άλλος είναι ο Joel Spolsky) [28, 29].

<sup>10</sup><https://stackexchange.com/sites>

<sup>11</sup><https://softwareengineering.stackexchange.com/>

ριση έργων (project management)<sup>12</sup>, για τη διασφάλιση ποιότητας και τον έλεγχο λογισμικού (software quality assurance and testing)<sup>13</sup>, κ.α.

Άλλες δημοφιλείς υπηρεσίες παρόμοιες με το Stack Exchange είναι το Quora<sup>14</sup> και το Reddit<sup>15</sup>. Το Quora ιδρύθηκε το 2010 ως κοινότητα ερωταπαντήσεων γενικού περιεχομένου. Δεδομένου όμως ότι υποστηρίζει ετικέτες για θέματα προγραμματισμού, επιλέγεται συχνά από προγραμματιστές που αναζητούν απαντήσεις στα ερωτήματά τους. Η Reddit, από την άλλη πλευρά, η οποία ιδρύθηκε το 2005, περιλαμβάνει ένα σύνολο κοινοτήτων, αρκετές από τις οποίες σχετίζονται με την ανάπτυξη λογισμικού. Ενδεικτικά αναφέρουμε την κοινότητα προγραμματισμού (programming community)<sup>16</sup> με περισσότερους από 800 χιλιάδες εγγεγραμμένους χρήστες, την κοινότητα ανάπτυξης λογισμικού (software development community)<sup>17</sup> με περίπου 12 χιλιάδες εγγεγραμμένους χρήστες, την κοινότητα αρχιτεκτονικής λογισμικού (software architecture community)<sup>18</sup> με περισσότερους από 3 χιλιάδες εγγεγραμμένους χρήστες, κ.α. Τέλος, σε αυτήν την κατηγορία υπηρεσιών μπορούμε επίσης να εντάξουμε τα forums των ιστότοπων προγραμματισμού, όπως π.χ. αυτό του CodeProject<sup>19</sup>, που ιδρύθηκε το 1999 και περιλαμβάνει tutorials προγραμματισμού, άρθρα, newsletters κ.α. Οι χώροι συζήτησης (discussion boards) της υπηρεσίας έχουν περισσότερα από 10 εκατομμύρια μέλη που μπορούν να συμμετέχουν σε συζητήσεις οι οποίες αφορούν γλώσσες προγραμματισμού, βιβλιοθήκες κ.λπ.

**Κατάλογοι Έργων Λογισμικού** Έως αυτό το σημείο, περιγράψαμε υπηρεσίες που επιτρέπουν στους προγραμματιστές να ανεβάσουν τον κώδικα τους online και να βρουν πιθανές λύσεις στα προβλήματα που μπορεί να προκύπτουν κατά την ανάπτυξη λογισμικού. Ένας άλλος πολύ σημαντικός τύπος υπηρεσιών που χρησιμοποιείται ευρέως σήμερα είναι αυτός των καταλόγων έργων λογισμικού (software project directories). Υπάρχουν πολλά διαφορετικά είδη καταλόγων που καλύπτουν διαφορετικές ανάγκες, ωστόσο όλα σχεδόν προσαρμόζονται στην πρακτική της ανάπτυξης λογισμικού βασισμένης σε τμήματα (component-based software engineering), καθώς ο στόχος τους συνήθως είναι η επαναχρησιμοποίηση. Κάποια ενδεικτικά παραδείγματα υπηρεσιών καταλόγων είναι το κεντρικό αποθετήριο του maven<sup>20</sup>, το μητρώο του Node Package Manager (npm registry)<sup>21</sup> και το ευρετήριο πακέτων της Python (Python Package Index - PyPI)<sup>22</sup>.

Όλες αυτές οι υπηρεσίες περιλαμβάνουν ένα online ευρετήριο βιβλιοθηκών λογισμικού (software libraries), καθώς και ένα εργαλείο για την ανάκτηση των βιβλιοθηκών που επιθυμεί ο προγραμματιστής. Το Maven χρησιμοποιείται κυρίως για έργα σε Java και το αποθετήριο του κατά τη στιγμή της συγγραφής περιέχει περισσότερες από 200.000 βιβλιοθήκες<sup>23</sup>. Ο προγραμματιστής μπορεί να περιηγηθεί ή να πραγματοποιήσει αναζήτηση στο αποθετήριο

<sup>12</sup><https://pm.stackexchange.com/>

<sup>13</sup><https://sqa.stackexchange.com/>

<sup>14</sup><https://www.quora.com/>

<sup>15</sup><https://www.reddit.com/>

<sup>16</sup><https://www.reddit.com/r/programming/>

<sup>17</sup><https://www.reddit.com/r/softwaredevelopment/>

<sup>18</sup><https://www.reddit.com/r/softwarearchitecture/>

<sup>19</sup><https://www.codeproject.com/>

<sup>20</sup><http://central.sonatype.org/>

<sup>21</sup><https://www.npmjs.com/>

<sup>22</sup><https://pypi.python.org/pypi>

<sup>23</sup><https://search.maven.org/#stats>

για χρήσιμες βιβλιοθήκες, οι οποίες στη συνέχεια μπορούν εύκολα να ενσωματωθούν στον κώδικά του, δηλώνοντάς τες ως εξαρτήσεις σε ένα αρχείο ρυθμίσεων (maven configuration file). Παρόμοια λογική ακολουθείται και για το μητρώο npm. Το μητρώο φιλοξενεί περισσότερα από 450.000 πακέτα για τη γλώσσα προγραμματισμού Javascript, επιτρέποντας στους προγραμματιστές να κατεβάσουν εύκολα τα πακέτα που επιθυμούν και να τα ενσωματώσουν στον πηγαίο κώδικά τους. Το αντίστοιχο ευρετήριο για τη γλώσσα προγραμματισμού Python, PyPI, περιέχει περισσότερες από 120.000 βιβλιοθήκες, οι οποίες μπορούν να εγκατασταθούν στη διανομή κάποιου χρησιμοποιώντας το σύστημα διαχείρισης πακέτων pip.

Τέλος, αναφέρουμε ότι οι κατάλογοι λογισμικού δεν περιορίζονται σε βιβλιοθήκες. Υπάρχουν, για παράδειγμα, κατάλογοι για υπηρεσίες διαδικτύου (web services), όπως το ευρετήριο του ProgrammableWeb<sup>24</sup>, το οποίο περιέχει τα APIs περισσότερων από 18.500 υπηρεσιών διαδικτύου. Οι προγραμματιστές μπορούν να περιηγηθούν σε αυτόν τον κατάλογο για να βρουν μια υπηρεσία διαδικτύου που μπορεί να είναι χρήσιμη στην περίπτωσή τους και να την ενσωματώσουν χρησιμοποιώντας το API της.

**Άλλες Πηγές** Αναμφίβολα, οι πηγές δεδομένων που συζητήθηκαν παραπάνω περιλαμβάνουν πολύ μεγάλο πλήθος δεδομένων. Οι περισσότερες από αυτά περιέχουν ακατέργαστα δεδομένα, δηλαδή πηγαίο κώδικα, δημοσιεύσεις σε forums (forum posts), βιβλιοθήκες λογισμικού κ.λπ. Μια διαφορετική κατηγορία συστημάτων είναι αυτή των συστημάτων που φιλοξενούν *μεταδεδομένα λογισμικού (software metadata)*. Τα τελευταία εμπεριέχουν δεδομένα από συστήματα διαχείρισης σφαλμάτων (bug tracking systems), υπηρεσίες συνεχούς ενσωμάτωσης (continuous integration) κ.α. Όσον αφορά τα συστήματα διαχείρισης σφαλμάτων, ένα από τα πιο δημοφιλή είναι η ενσωματωμένη υπηρεσία ζητημάτων (issues) του GitHub. Εκτός από το GitHub, σημειώνουμε ότι υπάρχουν αρκετές ακόμα ηλεκτρονικές υπηρεσίες που διευκολύνουν τη διαχείριση της ανάπτυξης λογισμικού. Χαρακτηριστικό παράδειγμα είναι το Bugzilla<sup>25</sup>, που είναι ένα σύστημα διαχείρισης σφαλμάτων (bug tracking system). Το Bugzilla χρησιμοποιείται από πλήθος έργων λογισμικού ανοικτού κώδικα (open-source), όπως ο περιηγητής ιστού (web browser) Mozilla Firefox<sup>26</sup>, το ολοκληρωμένο περιβάλλον ανάπτυξης Eclipse<sup>27</sup>, ή ακόμα και ο πυρήνας του Linux<sup>28</sup>. Τα συστήματα που προαναφέρθηκαν είναι ανοικτά στο κοινό, έτσι οποιοσδήποτε μπορεί να δημοσιεύσει πιθανά ζητήματα που προέκυψαν κατά τη χρήση της εκάστοτε εφαρμογής, να προτείνει λύσεις ή γενικά να αξιοποιήσει αυτά τα δεδομένα με σκοπό πιθανώς να βελτιώσει το δικό του έργο λογισμικού (π.χ. κατά την ανάπτυξη ενός plugin για το Eclipse IDE ή για τον περιηγητή Firefox).

Τέλος, ως πηγές μεταδεδομένων μπορούμε να αναφέρουμε και τις υπηρεσίες που περιέχουν επεξεργασμένη πληροφορία η οποία προέκυψε από την ανάλυση δεδομένων πηγαίου κώδικα, δεδομένων από συστήματα ελέγχου έκδοσης, κ.λπ. Παραδείγματα τέτοιων υπηρεσιών είναι η υποδομή της Boa (Boa Language and Infrastructure)<sup>29</sup> ή το GHTorrent project<sup>30</sup>, τα οποία εξάγουν μεταδεδομένα από έργα του GitHub, όπως π.χ. το Αφηρημένο Συντακτικό

<sup>24</sup><https://www.programmableweb.com/>

<sup>25</sup><https://www.bugzilla.org/>

<sup>26</sup><https://bugzilla.mozilla.org/>

<sup>27</sup><https://bugs.eclipse.org/bugs/>

<sup>28</sup><https://bugzilla.kernel.org/>

<sup>29</sup><http://boa.cs.iastate.edu/>

<sup>30</sup><http://ghtorrent.org/>

Δένδρο (Abstract Syntax Tree - AST) του πηγαίου κώδικα. Αυτές οι υπηρεσίες, ωστόσο, αναλύονται κυρίως σε άλλα κεφάλαια αυτής της εργασίας και όχι στο παρόν, καθώς είναι συνήθως ερευνητικά προϊόντα, τα οποία λειτουργούν με κάποιας μορφής πρωτογενή δεδομένα (όπως π.χ. η Boa χρησιμοποιεί τα δεδομένα του GitHub).

### 2.2.3 Προκλήσεις της Επαναχρησιμοποίησης Λογισμικού

Όπως είδαμε, τα δεδομένα τεχνολογίας λογισμικού που είναι ανά πάσα στιγμή διαθέσιμα στο διαδίκτυο είναι άφθονα. Αυτό δημιουργεί μια νέα πρακτική ανάπτυξης λογισμικού προσανατολισμένη σε δεδομένα (data-oriented) που σημαίνει ότι οι προγραμματιστές μπορούν να βρουν οτιδήποτε χρειάζονται, από εργαλεία που υποστηρίζουν τη διαχείριση έργων, τον καθορισμό των απαιτήσεων ή της σχεδίασης συστήματος, μέχρι και την επαναχρησιμοποίηση τμημάτων, τον ποιοτικό έλεγχο, κ.λπ. Συνεπώς, η κύρια πρόκληση έγκειται στην εξεύρεση των κατάλληλων αντικειμένων (εργαλείων ή τμημάτων λογισμικού ή γενικώς λύσεων που βασίζονται σε δεδομένα), στην κατανόηση της σημασιολογίας τους και στην ενσωμάτωσή τους στο έργο που αναπτύσσεται (στον πηγαίο κώδικα ή ακόμη και στη διαδικασία ανάπτυξης).

Όταν η παραπάνω διαδικασία ακολουθείται αποτελεσματικά, τα οφέλη στον χρόνο και στην προσπάθεια ανάπτυξης λογισμικού (και στη συνέχεια στο κόστος) είναι πραγματικά εντυπωσιακά. Ωστόσο, διάφορες δυσκολίες παρουσιάζονται όταν οι προγραμματιστές προσπαθούν να εφαρμόσουν αυτό το είδος επαναχρησιμοποίησης στην πράξη. Η πρώτη και ίσως η πιο σημαντική πρόκληση είναι ο εντοπισμός αυτού που αναζητεί ο προγραμματιστής: δεδομένου του όγκου των διαθέσιμων πληροφοριών στο διαδίκτυο, ο προγραμματιστής μπορεί εύκολα να βρεθεί σε δύσκολη θέση και να μην μπορέσει να εντοπίσει τη βέλτιστη λύση στην περίπτωσή του. Επιπλέον, ακόμα και σε περίπτωση που εντοπιστούν και ανακτηθούν τα απαιτούμενα τμήματα λογισμικού, ενδέχεται να υπάρχουν αρκετά προβλήματα στην κατανόηση του τρόπου λειτουργίας τους, π.χ. ορισμένα API βιβλιοθηκών λογισμικού ενδέχεται να μην είναι επαρκώς τεκμηριωμένα. Τέλος, αυτή η πρακτική της αναζήτησης και επαναχρησιμοποίησης (search-and-reuse) μπορεί να οδηγήσει σε προβλήματα ποιότητας, καθώς τα προβλήματα των εξαρτημάτων τρίτων (third-party) είναι πιθανό να μεταδοθούν στο ίδιο το προϊόν. Ακόμη και αν ένα τμήμα λογισμικού έχει σχεδιαστεί σωστά, εξακολουθεί να υπάρχει κάποιος κίνδυνος, καθώς η μη σωστή ενσωμάτωση (integration) μπορεί επίσης να οδηγήσει σε χαμηλής ποιότητας λογισμικό. Έτσι, καθώς η ποιότητα του λογισμικού είναι μια σημαντική πτυχή της επαναχρησιμοποίησης, επιλέγουμε στο επόμενο υποκεφάλαιο να αναλύσουμε τους βασικούς άξονές της, ενώ στο υποκεφάλαιο 2.4 θα είμαστε σε θέση να επανέλθουμε στο πρόβλημα στο οποίο αναφερόμαστε εδώ έχοντας πλέον κατά νου το σχετικό υπόβαθρο.

## 2.3 Ποιότητα Λογισμικού

### 2.3.1 Επισκόπηση της Ποιότητας και των Χαρακτηριστικών της

Η Ποιότητα Λογισμικού (Software Quality) είναι ένας από τους πιο σημαντικούς τομείς της Τεχνολογίας Λογισμικού, αλλά ταυτόχρονα είναι ένας τομέας που είναι αρκετά δύσκολο να καθοριστεί. Όπως αναφέρει ο David Garvin [30], “η ποιότητα είναι μια πολύπλοκη και

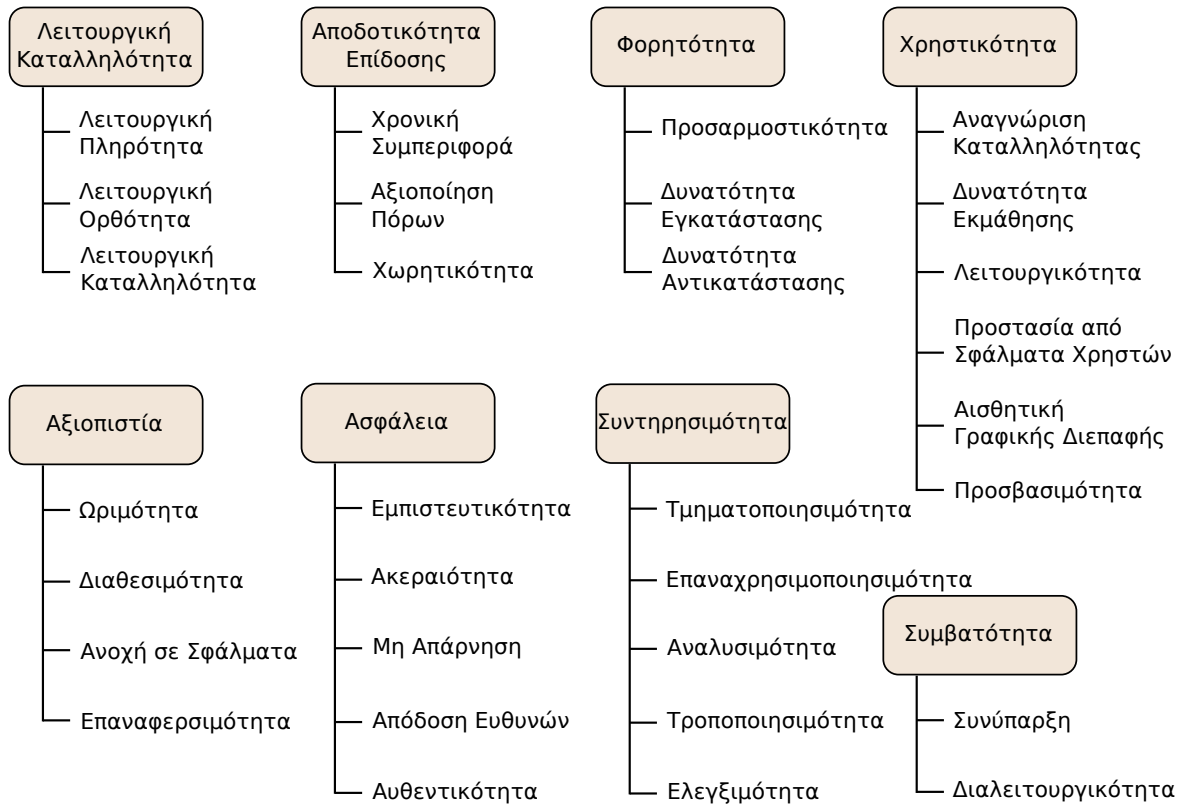
πολύπλευρη έννοια (complex and multifaceted concept)”. Παρόλο που η έννοια της ποιότητας δεν είναι καινούργια, η προσπάθεια να την ορίσουμε είναι αξιοσημείωτη<sup>31</sup>, καθώς η σημασία της είναι διαφορετική για διαφορετικούς ανθρώπους και σε διαφορετικά σενάρια. Σύμφωνα με τα συμπεράσματα του Garvin, που υιοθετήθηκαν αργότερα από διάφορους ερευνητές [16, 32], η ποιότητα ενός προϊόντος σχετίζεται με την επίδοσή του (performance), με το αν τα χαρακτηριστικά του είναι επιθυμητά, με το αν είναι αξιόπιστο (reliable), με τη συμμόρφωσή του με ορισμένα πρότυπα (standards), με την ανθεκτικότητα (durability) και την υποστηριξιμότητά του (serviceability), με την αισθητική (aesthetics) και τον τρόπο που την αντιλαμβάνεται ο χρήστης. Ένας άλλος ορισμός των παραγόντων που επηρεάζουν την ποιότητα του λογισμικού είναι αυτός του McCall και των συνεργατών του [33–35], οι οποίοι εστιάζουν (α) στη λειτουργία (operation) του προϊόντος, που μετράται με τα χαρακτηριστικά της ορθότητας (correctness), της αξιοπιστίας (reliability), της χρηστικότητας (usability), της ακεραιότητας (integrity) και της αποτελεσματικότητας (efficiency), (β) στην δυνατότητα προσαρμογής του προϊόντος, που μετράται με τα χαρακτηριστικά της συντηρησιμότητας (maintainability), της ευελιξίας (flexibility) και της ελεγχιμότητας (testability), και (γ) στην προσαρμοστικότητα του προϊόντος, που μετράται με τα χαρακτηριστικά της φορητότητας (portability), της επαναχρησιμοποιησιμότητας (reusability) και της διαλειτουργικότητας (interoperability). Αν και χρήσιμα για τον ορισμό της ποιότητας, τα χαρακτηριστικά που αναφέρθηκαν παραπάνω είναι συχνά δύσκολο να μετρηθούν [15].

Σήμερα, η πιο ευρέως αποδεκτή κατηγοριοποίηση της ποιότητας του λογισμικού είναι αυτή του τελευταίου ISO ποιότητας, ενός προτύπου που αναπτύχθηκε για να προσδιορίσει τα βασικά χαρακτηριστικά της ποιότητας (quality characteristics), τα οποία διακρίνονται περαιτέρω σε υπο-χαρακτηριστικά (sub-characteristics) που είναι σε κάποιο βαθμό μετρήσιμα. Το πρότυπο ISO/IEC 9126 [36], το οποίο εκδόθηκε το 1991, περιελάμβανε ως χαρακτηριστικά ποιότητας ενός προϊόντος τη Λειτουργικότητα (Functionality), την Αξιοπιστία (Reliability), τη Χρηστικότητα (Usability), την Αποδοτικότητα (Efficiency), τη Συντηρησιμότητα (Maintainability) και τη Φορητότητα (Portability). Τα χαρακτηριστικά αυτά υποδιαιρούνται περαιτέρω σε υπο-χαρακτηριστικά, τα οποία με τη σειρά τους χωρίστηκαν σε *ιδιότητες (attributes)* οι οποίες, θεωρητικά τουλάχιστον, είναι μετρήσιμες και επαληθεύσιμες. Η πρόκληση για την οποία δεν υπάρχει μέχρι στιγμής κάποια κοινά αποδεκτή λύση είναι να οριστούν οι μετρικές (metrics) που θα χρησιμοποιούνται για τη μέτρηση κάθε ιδιότητας ποιότητας. Οι μετρικές αυτές συχνά εξαρτώνται από το προϊόν και τις εμπλεκόμενες τεχνολογίες. Η πιο βασική επανέκδοση του προτύπου ποιότητας ήταν αυτή του προτύπου ISO/IEC 25010 [17], το οποίο προτάθηκε το 2011 και περιλαμβάνει τα οκτώ χαρακτηριστικά ποιότητας που απεικονίζονται στο Σχήμα 2.8, μαζί με τα σχετικά υπο-χαρακτηριστικά τους.

Κάθε χαρακτηριστικό του ISO αναφέρεται σε μια διαφορετική πτυχή του προϊόντος λογισμικού. Αυτές οι πτυχές είναι οι εξής:

- **Λειτουργική Καταλληλότητα (Functional Suitability):** ο βαθμός στον οποίο το προϊόν πληροί τις απαιτήσεις που τίθενται,
- **Αποδοτικότητα Επίδοσης (Performance Efficiency):** η απόδοση του προϊόντος σε σχέση με τους διαθέσιμους πόρους,

<sup>31</sup> Ενδεικτικά, μπορούμε να αναφερθούμε σε ένα απόσπασμα του μάλλον αφηρημένου “ορισμού” του Robert Persig [31]: “Ποιότητα... ξέρετε τι είναι, αλλά δεν ξέρετε τι είναι”.



Σχήμα 2.8: Χαρακτηριστικά και Υπο-χαρακτηριστικά Ποιότητας του ISO/IEC 25010:2011

- Φορητότητα (Portability): η δυνατότητα μεταφοράς ενός προϊόντος σε περιβάλλον με διαφορετικό υλικό ή λογισμικό,
- Χρηστικότητα (Usability): ο βαθμός στον οποίο το προϊόν μπορεί να χρησιμοποιηθεί αποτελεσματικά και ικανοποιητικά από τους χρήστες,
- Αξιοπιστία (Reliability): η αποτελεσματικότητα με την οποία το προϊόν εκτελεί τις λειτουργίες του κάτω από συγκεκριμένες συνθήκες και συγκεκριμένο χρόνο,
- Ασφάλεια (Security): ο βαθμός στον οποίο τα δεδομένα του προϊόντος προστατεύονται και προσπελάζονται μόνο όταν ζητούνται από εξουσιοδοτημένους χρήστες, και
- Συντηρησιμότητα (Maintainability): η αποτελεσματικότητα με την οποία το προϊόν μπορεί να τροποποιηθεί είτε για διορθώσεις είτε για προσαρμογές σε νέα περιβάλλοντα ή απαιτήσεις.

Όπως ήδη αναφέρθηκε, παρουσιάζονται αρκετές δυσκολίες κατά την προσπάθεια μέτρησης της ποιότητας ενός προϊόντος λογισμικού. Έτσι, καταβάλλεται συνεχής προσπάθεια στο σχεδιασμό *μετρικών (metrics)* που θα αντιστοιχηθούν αποτελεσματικά στα χαρακτηριστικά και στα υπο-χαρακτηριστικά ποιότητας του Σχήματος 2.8. Οι διάφορες μετρικές που έχουν προταθεί κατά καιρούς [13, 37] μπορούν να ταξινομηθούν σύμφωνα με το πεδίο όπου εφαρμόζονται για τη διαχείριση ποιότητας ή, με απλά λόγια, σύμφωνα με τον τύπο



των δεδομένων που μετράται. Σύμφωνα με το SWEBOK [38]<sup>32</sup>, υπάρχουν τέσσερις κατηγορίες τεχνικών διαχείρισης ποιότητας λογισμικού (software quality management techniques): οι τεχνικές στατικής ανάλυσης (static analysis), οι τεχνικές δυναμικής ανάλυσης (dynamic analysis), ο έλεγχος (testing) και οι τεχνικές μέτρησης διαδικασιών λογισμικού (software process measurement). Κάθε κατηγορία έχει τον δικό της σκοπό, όπως ορίζεται από την τρέχουσα έρευνα και πρακτική, ενώ διαφορετικές μετρικές έχουν προταθεί με την πάροδο του χρόνου καθώς αλλάζουν οι μεθοδολογίες ανάπτυξης λογισμικού. Οι τεχνικές αυτές αναλύονται στις ακόλουθες ενότητες, όπου δίνονται και παραδείγματα μετρικών για κάθε κατηγορία.

### 2.3.2 Μετρικές Ποιότητας

Όπως ήδη αναφέρθηκε, υπάρχουν διάφοροι τύποι μετρικών στην Τεχνολογία Λογισμικού, που μετρούν διαφορετικές πτυχές του υπό ανάπτυξη συστήματος καθώς και της ίδιας της διαδικασίας ανάπτυξης λογισμικού [13, 37]. Ως μια ενδεικτική (όμως όχι απαραίτητα διεξοδική) αντιστοίχιση στα ποιοτικά χαρακτηριστικά του Σχήματος 2.8, μπορούμε να πούμε ότι χαρακτηριστικά ποιότητας όπως η συντηρησιμότητα (maintainability) ή η χρηστικότητα (usability) συνήθως μετρώνται με κάποιας μορφής μετρικές στατικής ανάλυσης (static analysis metrics). Αντίστοιχα, οι μετρικές δυναμικής ανάλυσης (dynamic analysis metrics) χρησιμοποιούνται συνήθως για τη μέτρηση χαρακτηριστικών όπως η αξιοπιστία (reliability) ή η αποδοτικότητα επίδοσης (performance efficiency). Ο έλεγχος λογισμικού (testing), που αποτελεί ουσιαστικά μια διακριτή περιοχή κι έτσι αναλύεται στο επόμενο υποκεφάλαιο, αφορά κυρίως την αξιολόγηση της λειτουργικής καταλληλότητας (functional suitability) του προϊόντος. Τέλος, υπάρχουν και μετρικές που αφορούν την ίδια τη διαδικασία λογισμικού (software process metrics).

**Μετρικές Στατικής Ανάλυσης** Η πρώτη κατηγορία μετρικών που πρόκειται να αναλύσουμε αναφέρεται σε μετρικές που προκύπτουν από την ανάλυση του πηγαίου κώδικα μιας εφαρμογής, χωρίς να την εκτελέσουμε. Οι μετρικές στατικής ανάλυσης προτάθηκαν αρχικά τη δεκαετία του '70, με το μέγεθος (*size*) και την πολυπλοκότητα (*complexity*) να είναι οι δύο πιο βασικές ιδιότητες προς μέτρηση. Οι πιο ευρέως χρησιμοποιούμενες μετρικές ήταν ο αριθμός γραμμών κώδικα (Lines of Code - LoC), η κυκλωματική πολυπλοκότητα (McCabe cyclomatic complexity) [39] και οι μετρικές πολυπλοκότητας του Halstead [40]. Αν και σχετικά απλοϊκή, η μετρική LoC ήταν αρχικά ιδιαίτερα χρήσιμη, καθώς σε ορισμένες ευρέως χρησιμοποιούμενες γλώσσες της εποχής, όπως η FORTRAN, κάθε μη κενή γραμμή αντιπροσώπευε μια συγκεκριμένη εντολή. Σήμερα, η μετρική αυτή συνεχίζει να είναι χρήσιμη, ωστόσο ως επί το πλείστον για να παρέχει μια τάξη μεγέθους για τον πηγαίο κώδικα μιας εφαρμογής ή ενός τμήματος. Η κυκλωματική πολυπλοκότητα προτάθηκε το 1976 σε μια προσπάθεια να μετρηθεί πόσο πολύπλοκο είναι ένα σύστημα λογισμικού. Η μετρική υπολογίζεται ως το πλήθος των γραμμικά ανεξάρτητων διαδρομών (linearly independent paths) του πηγαίου κώδικα του λογισμικού. Το σύνολο μετρικών που αναπτύχθηκε το 1977 από

<sup>32</sup>Το SWEBOK, που αποτελεί ακρωνύμιο του Software Engineering Body of Knowledge, είναι ένα διεθνές πρότυπο ISO που έχει δημιουργηθεί από τις συνεργατικές προσπάθειες των επαγγελματιών της Τεχνολογίας Λογισμικού και δημοσιεύεται από τον οργανισμό IEEE. Βασικός σκοπός του προτύπου είναι να συνοψίσει τη βασική θεωρία και τις πρακτικές της Τεχνολογίας Λογισμικού και να συγκεντρώσει αναφορές για όλες τις έννοιες που είναι σχετικές με την περιοχή.

τον Maurice Halstead [40] είχε κι αυτό ως σκοπό τη μέτρηση της πολυπλοκότητας, ωστόσο βασιζόταν στο πλήθος των τελεστών (operators) και των τελεστέων (operands) του κώδικα.

Παρόλο που οι παραπάνω μετρικές προτάθηκαν πριν από περισσότερα από 40 χρόνια, εξακολουθούν να χρησιμοποιούνται ευρέως (και στις περισσότερες περιπτώσεις ορθώς<sup>33</sup>) ως ενδείξεις του μεγέθους και της πολυπλοκότητας ενός προγράμματος. Ωστόσο, η υιοθέτηση και ιδιαίτερα η εδραίωση του Αντικειμενοστραφούς Προγραμματισμού (Object-Oriented Programming) τη δεκαετία του '90 δημιούργησε νέες ανάγκες σχετικές με τη μέτρηση ποιότητας, καθώς η ποιότητα του πηγαίου κώδικα ενός αντικειμενοστραφούς προγράμματος σχετίζεται περαιτέρω με τις ιδιότητες της *σύζευξης* (coupling) και της *κληρονομικότητας* (inheritance). Ως εκ τούτου, αρκετές νέες μετρικές προτάθηκαν για την αξιολόγηση αυτών των ιδιοτήτων. Ενδεικτικά, μπορούμε να αναφερθούμε στις μετρικές C&K (C&K metrics) [41] οι οποίες πήραν το όνομά τους από τους δημιουργούς τους, Chidamber και Kemerer, και παρουσιάζονται στον Πίνακα 2.2. Η μετρική WMC υπολογίζεται συνήθως

Πίνακας 2.2: Μετρικές C&K

ID	Όνομα	Περιγραφή
WMC	Weighted Methods per Class	Σταθμισμένη πολυπλοκότητα μεθόδων
DIT	Depth of Inheritance Tree	Βάθος του δένδρου κληρονομικότητας
NOC	Number Of Children	Πλήθος των απογόνων μιας κλάσης
CBO	Coupling Between Objects	Σύζευξη μεταξύ των αντικειμένων
RFC	Response For a Class	Απόκριση για μια κλάση
LCOM	Lack of Cohesion in Methods	Έλλειψη συνοχής μεταξύ των μεθόδων

ως το άθροισμα των τιμών πολυπλοκότητας όλων των μεθόδων της κλάσης που αναλύεται. Οι μετρικές DIT και NOC αναφέρονται στον αριθμό των υπερκλάσεων (superclasses) και των υποκλάσεων (subclasses) της κλάσης, αντίστοιχα. Οι μετρικές RFC και CBO χρησιμοποιούνται για την ανάλυση των σχέσεων μεταξύ των κλάσεων. Η μετρική RFC τυπικά ορίζεται ως το πλήθος των μεθόδων της κλάσης καθώς και το πλήθος των εξωτερικών μεθόδων που καλούνται από αυτές, ενώ η μετρική CBO αναφέρεται στον αριθμό των κλάσεων από τις οποίες εξαρτάται η κλάση υπό ανάλυση, των οποίων δηλαδή οι μέθοδοι/μεταβλητές προσπελαύνονται από την κλάση. Ορίζεται επιπλέον η αντίστροφη αυτής της μετρικής ως η αντίστροφη σύζευξη μεταξύ αντικειμένων (Coupling Between Objects Inverse - CBOI). Τέλος, η μετρική LCOM είναι ένα μέτρο για τη συνοχή μεταξύ των μεθόδων μιας κλάσης και υπολογίζεται ως το πλήθος των ζευγών μεθόδων με κοινές μεταβλητές μεταξύ τους, έχοντας αφαιρέσει το πλήθος των ζευγών μεθόδων χωρίς κοινές μεταβλητές μεταξύ τους<sup>34</sup>.

<sup>33</sup>Μια σημαντική σημείωση είναι ότι οποιαδήποτε μετρική πρέπει να χρησιμοποιείται όταν η τιμή της έχει πράγματι νόημα. Ένα δημοφιλές ρητό, το οποίο συχνά αποδίδεται στο συνιδρυτή και πρώην πρόεδρο της Microsoft Bill Gates, αναφέρει σε ελεύθερη μετάφραση ότι “το να μετρά κανείς την πρόοδο της κατασκευής λογισμικού με βάση το πλήθος των γραμμών κώδικα είναι σαν να μετρά την πρόοδο της κατασκευής αεροσκαφών με βάση το βάρος.”

<sup>34</sup>Αυτός ο αρχικός ορισμός του LCOM θεωρείται συχνά δύσκολο να υπολογιστεί, καθώς η μέγιστη τιμή του εξαρτάται από το πλήθος των ζευγών μεθόδων. Ένας άλλος τρόπος υπολογισμού προτάθηκε αργότερα από τον Henderson-Sellers [42], που συνήθως αναφέρεται ως LCOM3 (και η αρχική μετρική τότε αναφέρεται ως LCOM1), και ορίζει την τιμή της μετρικής ως  $(M - A/V)/(M - 1)$ , όπου  $M$  είναι το πλήθος των μεθόδων και  $A$  είναι το πλήθος των προσπελάσεων για τις  $V$  μεταβλητές της κλάσης.

Ένα άλλο δημοφιλές σύνολο αντικειμενοστραφών μετρικών που προτάθηκε το 1994 είναι οι μετρικές MOOD (Metrics for Object Oriented Design) [43]. Στις μετρικές MOOD περιλαμβάνονται μετρικές για την ενθυλάκωση (encapsulation), όπως ο συντελεστής απόκρυψης μεθόδων (Method Hiding Factor - MHF) και ο συντελεστής απόκρυψης χαρακτηριστικών (Attribute Hiding Factor - AHF), μετρικές για την κληρονομικότητα (inheritance), όπως ο συντελεστής κληρονομικότητας μεθόδων (Method Inheritance Factor - MIF) και ο συντελεστής κληρονομικότητας χαρακτηριστικών (Attribute Inheritance Factor - AIF), καθώς και μετρικές για τον πολυμορφισμό (polymorphism) και τη σύζευξη (coupling), όπως ο συντελεστής πολυμορφισμού (Polymorphism Factor - POF) και ο συντελεστής σύζευξης (Coupling Factor - COF), αντίστοιχα. Οι προτάσεις μετρικών, ωστόσο, δεν περιορίζονται στις παραπάνω· υπάρχουν αρκετές άλλες δημοφιλείς προσεγγίσεις, όπως π.χ. το σύνολο μετρικών που προτάθηκε από τους Lorenz και Kidd [44]. Σε κάθε περίπτωση όμως, η βασική λογική πίσω από όλες αυτές τις μετρικές είναι παρόμοια, καθώς όλες μετρούν τις βασικές ιδιότητες ποιότητας της αντικειμενοστραφούς σχεδίασης χρησιμοποιώντας ως είσοδο τον πηγαίο κώδικα ενός έργου λογισμικού<sup>35</sup>.

**Μετρικές Δυναμικής Ανάλυσης** Σε αντίθεση με τη στατική ανάλυση, η δυναμική ανάλυση πραγματοποιείται εκτελώντας ένα πρόγραμμα και μετρώντας στοιχεία της συμπεριφοράς του. Ως εκ τούτου, η σύγκριση μεταξύ των δύο τύπων αναλύσεων είναι αρκετά ενδιαφέρουσα. Οι μετρικές στατικής ανάλυσης είναι συνήθως πιο εύκολο να υπολογιστούν και είναι επίσης διαθέσιμες σε πρώιμα στάδια της ανάπτυξης λογισμικού. Από την άλλη πλευρά, η δυναμική ανάλυση επιτρέπει επιπλέον τη μέτρηση των παραμέτρων συμπεριφοράς του λογισμικού, κάτι που δεν είναι εφικτό με τη χρήση στατικής ανάλυσης [46]. Ένα ενδεικτικό σύνολο μετρικών για αυτήν την κατηγορία είναι οι μετρικές σύζευξης EOC (Export Object Coupling) και IOC (Import Object Coupling), που προτάθηκαν από τον Yacoub και τους συνεργάτες του [47]. Οι μετρικές EOC και IOC αφορούν τον αριθμό των μηνυμάτων που αποστέλλονται ή λαμβάνονται, αντίστοιχα, από ένα αντικείμενο σε ένα άλλο. Υπάρχουν αρκετές άλλες προτάσεις στη βιβλιογραφία, όπως το σύνολο των δώδεκα μετρικών που πρότειναν ο Arisholm και οι συνεργάτες του [48]. Οι συγκεκριμένες μετρικές μετρώνται κατά την εκτέλεση της εφαρμογής και σχετίζονται με τη σύζευξη, περιλαμβάνοντας π.χ. τον αριθμό των εξωτερικών μεθόδων που καλούνται από μια κλάση, τον αριθμό των κλάσεων που χρησιμοποιούνται από μια κλάση, κ.α.<sup>36</sup>

Ένας άλλος σημαντικός άξονας εφαρμογής δυναμικής ανάλυσης είναι η δυνατότητα μέτρησης χαρακτηριστικών όπως η αξιοπιστία (reliability)<sup>37</sup>. Οι μετρικές αξιοπιστίας είναι εξωτερικές, καθώς αφορούν κυρίως τη συμπεριφορά ενός συστήματος ή ενός τμήματος κατά την εκτέλεση του, χωρίς να εστιάζουν στις εσωτερικές του λειτουργίες. Οι μετρικές αυτής της κατηγορίας καθώς και ο ορισμός τους ποικίλουν, καθώς ενδέχεται να εξαρτώνται από τις ιδιαιτερότητες του κάθε συστήματος [49]. Οι πιο δημοφιλείς μετρικές σε αυτή την κατηγορία είναι ο μέσος χρόνος μεταξύ αποτυχιών (Mean Time Between Failures - MTBF) που μετρά την ποσότητα των βλαβών για ένα συγκεκριμένο χρονικό διάστημα εκτέλεσης,

<sup>35</sup>Ο αναγνώστης παραπέμπεται στο [45] για μια συνολική ανασκόπηση των αντικειμενοστραφών μετρικών στατικής ανάλυσης.

<sup>36</sup>Καθώς υπάρχουν αρκετές άλλες μετρικές σε αυτόν τον τομέα, για μια εκτεταμένη ανασκόπηση μετρικών, ο αναγνώστης παραπέμπεται στο [15] [46].

<sup>37</sup>Αντίστοιχα, η στατική ανάλυση συνήθως εστιάζει στα χαρακτηριστικά της συντηρησιμότητας (maintainability) και της χρηστικότητας (usability).

ο μέσος χρόνος έως την αποτυχία (Mean Time To Failure - MTTF) που αναφέρεται στο αναμενόμενο χρονικό διάστημα μέχρι την πρώτη αποτυχία του συστήματος, καθώς και ο μέσος χρόνος για την επιδιόρθωση (Mean Time To Repair - MTTR) που μετράται ως το αναμενόμενο χρονικό διάστημα που χρειάζεται για την επισκευή ενός συστήματος μετά την αποτυχία του. Άλλες σημαντικές μετρικές περιλαμβάνουν τη μετρική POFOD (Probability Of Failure On Demand), η οποία μετρά την πιθανότητα αποτυχίας ενός συστήματος κατά την υποβολή ενός αιτήματος (request), τη μετρική ROCOF (Rate Of Occurrence Of Failures) που αφορά τη συχνότητα των αποτυχιών, και τη διαθεσιμότητα (Availability - AVAIL) που υπολογίζεται ως η πιθανότητα το σύστημα να είναι διαθέσιμο για χρήση σε ένα δεδομένο χρονικό διάστημα.

**Μετρικές Ανάλυσης Διαδικασίας Ανάπτυξης Λογισμικού** Όπως και με τις προηγούμενες κατηγορίες μετρικών, τα μοντέλα μετρικών των διαδικασιών λογισμικού καλύπτουν ένα ευρύ φάσμα πιθανών εφαρμογών. Αυτή η κατηγορία μετρά τη δυναμική των διαδικασιών που ακολουθούνται από μια ομάδα ανάπτυξης λογισμικού. Ορισμένες από τις πιο βασικές έννοιες σε αυτό τον τομέα αποδίδονται στον ερευνητή της IBM, Allan Albrecht. Το 1979, ο Albrecht προσπάθησε να μετρήσει την παραγωγικότητα ανάπτυξης λογισμικού χρησιμοποιώντας μια μετρική που ο ίδιος εφηύρε, που είναι γνωστή ως το *Λειτουργικό Σημείο (Function Point - FP)* [50]. Σε αντίθεση με τις μετρικές μεγέθους (size-oriented metrics) που αναφέρθηκαν προηγουμένως σε αυτή την ενότητα, η μετρική FP, η οποία καθορίζεται εμπειρικά με βάση το πεδίο εφαρμογής (domain) και την πολυπλοκότητα του λογισμικού, οδήγησε σε αυτό που ονομάζουμε *μετρικές λειτουργιών (function-oriented metrics)*<sup>38</sup>. Εάν καταφέρουμε να ορίσουμε το Λειτουργικό Σημείο, τότε αυτό μπορεί να χρησιμοποιηθεί ως τιμή αναφοράς για να αξιολογηθεί η απόδοση της διαδικασίας ανάπτυξης λογισμικού. Οι μετρικές που έχουν προταθεί αφορούν διάφορα στοιχεία της διαδικασίας ανάπτυξης, όπως π.χ. τον αριθμό των σφαλμάτων/ελαττωμάτων (number of errors/defects per FP), το κόστος (cost per FP), την τεκμηρίωση (amount of documentation per FP), την ανθρωποπροσπάθεια (number of FPs per person-month of work), κ.α.

Όπως μπορεί να παρατηρήσει ο σκεπτικός αναγνώστης, μετρικές απόδοσης όπως οι παραπάνω μπορούν να χρησιμοποιηθούν όχι μόνο για να αξιολογήσουν τη διαδικασία ανάπτυξης ή το προϊόν μιας ομάδας ανάπτυξης, αλλά και την ίδια την ομάδα ως σύνολο ή ακόμα και μεμονωμένα. Αναμφίβολα, οι μετρήσεις που αφορούν τη δραστηριότητα των προγραμματιστών μπορούν να φανούν χρήσιμες σε διάφορες περιπτώσεις, όπως π.χ. για την πρόβλεψη της εμφάνισης σφαλμάτων [52]. Ωστόσο, η αξιολόγηση της παραγωγικότητας των προγραμματιστών με τη χρήση τέτοιων μετρικών είναι ιδιαίτερα επισφαλής, καθώς μπορεί εύκολα να οδηγήσει σε εσφαλμένες εκτιμήσεις και ακόμα και σε αθέμιτες αποφάσεις που ενδέχεται να βλάψουν την παραγωγικότητα μιας ομάδας<sup>39</sup>.

Τον τελευταίο καιρό, η τάση προς τη χρήση συστημάτων ελέγχου έκδοσης (version control systems) έχει συνεισφέρει στην εγκαθίδρυση μιας νέας κατηγορίας μετρικών, των λεγόμενων *μετρικών αλλαγών (change metrics)*. Οι μετρικές αυτής της κατηγορίας επιλέγο-

<sup>38</sup> Στην πραγματικότητα, η αρχική δημοσίευση από τον Albrecht [50] ανέφερε ότι οι μετρικές λειτουργιών και οι μετρικές μεγέθους είχαν υψηλή συσχέτιση, ωστόσο η μετέπειτα ανάπτυξη διαφόρων γλωσσών και τεχνικών προγραμματισμού απέδειξε ότι οι διαφορές τους είναι αρκετά σημαντικές [51].

<sup>39</sup> Σύμφωνα με τον Austin [53], η χρήση τέτοιων συστημάτων μέτρησης οδηγεί συχνά σε δυσλειτουργικές περιπτώσεις (*measurement dysfunction*), καθώς οι προγραμματιστές τελικά προσαρμόζονται στη διαδικασία μέτρησης και εστιάζουν στην ίδια τη μέτρηση και όχι στην ποιότητα του προϊόντος.

νται για διάφορους σκοπούς, όπως π.χ. για την πρόβλεψη ελαττωμάτων (defect prediction) ή για τον εντοπισμό λανθασμένων αποφάσεων κατά τη διαδικασία ανάπτυξης. Ένα ενδεικτικό σύνολο μετρικών αλλαγών που προτάθηκε από τον Moser και τους συνεργάτες του [54] παρουσιάζεται στον Πίνακα 2.3.

Πίνακας 2.3: Μετρικές Αλλαγών του Moser και των συνεργατών του [54]

Name	Definition
REVISIONS	Αριθμός αναθεωρήσεων ενός αρχείου
REFACTORINGS	Πλήθος φορών που έγινε refactoring σε ένα αρχείο
BUGFIXES	Πλήθος φορών που ένα αρχείο άλλαξε κατά τη διόρθωση σφαλμάτων
AUTHORS	Αριθμός ατόμων που έκαναν αλλαγές σε ένα αρχείο
LOC_ADDED	Αριθμός γραμμών κώδικα που προστέθηκαν σε ένα αρχείο κατά τις αναθεωρήσεις, υπολογισμένος ως άθροισμα, ως το μέγιστο μεταξύ όλων των αναθεωρήσεων και ως ο μέσος όρος ανά αναθεώρηση
LOC_DELETED	Αριθμός γραμμών κώδικα που αφαιρέθηκαν από ένα αρχείο κατά τις αναθεωρήσεις, υπολογισμένος ως άθροισμα, ως το μέγιστο μεταξύ όλων των αναθεωρήσεων και ως ο μέσος όρος ανά αναθεώρηση
CODECHURN	Αριθμός γραμμών κώδικα που προστέθηκαν μείον αριθμός γραμμών κώδικα που αφαιρέθηκαν, υπολογισμένος ως άθροισμα, ως το μέγιστο μεταξύ όλων των αναθεωρήσεων και ως ο μέσος όρος ανά αναθεώρηση
CHANGESET	Αριθμός αρχείων που έγιναν commit μαζί στο αποθετήριο, υπολογιζόμενα ως το μέγιστο μεταξύ όλων των αναθεωρήσεων και ως ο μέσος όρος ανά αναθεώρηση
AGE	Ηλικία ενός αρχείου σε εβδομάδες (μετρώντας προς τα πίσω από κάποιο release)
WEIGHTED_AGE	Ηλικία ενός αρχείου σε εβδομάδες σταθμισμένη με το τον αριθμό των γραμμών κώδικα που προστέθηκαν κάθε εβδομάδα

Αυτές οι μετρικές μπορούν να υπολογιστούν χρησιμοποιώντας πληροφορίες από τα commits και τα issues (bugs) που υπάρχουν στα συστήματα ελέγχου έκδοσης, ενώ το βασικό πεδίο εφαρμογής τους είναι συνήθως η πρόβλεψη ελαττωμάτων (defect prediction) σε έργα λογισμικού. Το σκεπτικό για κάθε μία από αυτές τις μετρικές είναι αρκετά λογικό: για παράδειγμα, αρχεία (ή, γενικώς, τμήματα λογισμικού) με πολλές αλλαγές από διάφορα άτομα αναμένεται να είναι πιο επιρρεπή σε σφάλματα. Οι μετρικές CHANGESET παρουσιάζουν επίσης ενδιαφέρον, καθώς υποδεικνύουν πόσα αρχεία επηρεάζονται σε κάθε αναθεώρηση (revision). Βασιζόμενος στην υπόθεση ότι οι αλλαγές που επηρεάζουν πολλά αρχεία μπορούν να οδηγήσουν πιο εύκολα σε σφάλματα, ο Hassan πρότεινε επιπλέον ως μετρική την εντροπία των αλλαγών στον κώδικα (entropy of code changes) [55], που υπολογίζεται για κάποιο χρονικό διάστημα από τον τύπο  $\sum_{k=1}^n p_k \cdot \log_2 p_k$ , όπου  $p_k$  είναι η πιθανότητα να έχει αλλαγές το  $k$ -στο αρχείο.

Ως τελική σημείωση, οι μετρικές που αναλύθηκαν στις προηγούμενες παραγράφους μπορούν να εφαρμοστούν για διάφορους σκοπούς, όπως για παράδειγμα για την αξιολό-

γηση της επαναχρησιμοποιησιμότητας κάποιων τμημάτων (component reusability), την ανίχνευση σφαλμάτων στον πηγαίο κώδικα (bug detection), τον προσδιορισμό του καταλληλότερου προγραμματιστή για να αναλάβει ένα τμήμα του πηγαίου κώδικα (developer task assignment), κ.α. Είναι σημαντικό, ωστόσο, να αναφέρουμε ότι οι μετρικές δεν επαρκούν όταν προσπαθούμε να αξιολογήσουμε την ποιότητα του λογισμικού ή ενδεχομένως να ανιχνεύσουμε ελαττώματα. Αυτό συμβαίνει καθώς οι μετρικές από μόνες τους δεν περιγράφουν τα τμήματα λογισμικού από μια ολιστική προσέγγιση. Για παράδειγμα, μια κλάση με μεγάλο μέγεθος δεν είναι απαραίτητο ότι έχει πολύπλοκο κώδικα που είναι δύσκολο να συντηρηθεί. Αντίστοιχα, μια πολύπλοκη κλάση με αρκετές συνδέσεις με άλλες κλάσεις μπορεί να είναι απολύτως αποδεκτή αν είναι σχετικά μικρή σε μέγεθος και σωστά σχεδιασμένη. Ως εκ τούτου, η πρόκληση με αυτές τις μετρικές έγκειται συνήθως στον ορισμό ενός μοντέλου [56], δηλαδή ενός συνόλου κατωφλιών (thresholds), τα οποία όταν εφαρμόζονται ταυτόχρονα, μπορούν να εκτιμήσουν συγκεκριμένα την ποιότητα του λογισμικού, να ανιχνεύσουν ελαττώματα κ.λπ.

### 2.3.3 Έλεγχος Λογισμικού

Σύμφωνα με τον Sommerville [2], η διαδικασία ελέγχου έχει δύο στόχους: (α) να αποδείξει ότι το λογισμικό πληροί τις απαιτήσεις που τίθενται, και (β) να εντοπίσει σφάλματα στο λογισμικό. Οι στόχοι αυτοί έρχονται επίσης σε συμφωνία με την άποψη ότι ο έλεγχος ανήκει στο ευρύτερο φάσμα των διαδικασιών επαλήθευσης και επικύρωσης (verification and validation) [15]. Εάν υιοθετήσουμε τους ορισμούς του Boehm [57], η επαλήθευση διασφαλίζει ότι “δημιουργούμε το προϊόν σωστά”, ενώ η επικύρωση διασφαλίζει ότι “δημιουργούμε το σωστό προϊόν”. Θα προτιμήσουμε αυτή την ερμηνεία του ελέγχου λογισμικού, η οποία είναι ευρέως αποδεκτή<sup>40</sup>, καθώς περιλαμβάνει τη σύνδεση του ελέγχου όχι μόνο με το τελικό προϊόν, αλλά και με τις απαιτήσεις του λογισμικού.

Για να συζητήσουμε περαιτέρω αυτή μας την προτίμηση και πριν αναλύσουμε τον τρόπο με τον οποίο πραγματοποιείται ο έλεγχος λογισμικού, θα παρέχουμε κάποιους ορισμούς των διάφορων όρων που σχετίζονται με τον έλεγχο και με την απομάκρυνση των σφαλμάτων (debugging). Πιθανώς ο πιο γνωστός όρος (ακόμα και σε άτομα εκτός του τομέα της Επιστήμης Υπολογιστών) είναι το *bug* (σφάλμα). Αν και ο όρος “bug” συναντάται στη μηχανική (engineering) ήδη από τα τέλη του 19ου αιώνα<sup>41</sup>, στη συνέχεια κατέστη πολύ δημοφιλής για την περιγραφή των σφαλμάτων λογισμικού ακόμα και σε γλώσσες προγραμματισμού υψη-

<sup>40</sup>Υπάρχουν βεβαίως και άλλες πιο συγκεκριμένες ερμηνείες, όπως αυτή που παρέχεται στο [58], η οποία ορίζει τον έλεγχο ως “τη διαδικασία εκτέλεσης ενός προγράμματος με σκοπό την εύρεση σφαλμάτων”. Ωστόσο, καθώς οι καιροί αλλάζουν, ο έλεγχος έχει μετατραπεί σε μια σημαντική διαδικασία που μπορεί να προσθέσει αξία στο λογισμικό και συνεπώς δεν περιορίζεται στην ανακάλυψη σφαλμάτων. Μια ενδιαφέρουσα οπτική για αυτό το θέμα παρέχεται από τον Beizer [59], ο οποίος ισχυρίζεται ότι διαφορετικές ομάδες ανάπτυξης έχουν διαφορετικά επίπεδα ωριμότητας όσον αφορά τον σκοπό του ελέγχου (testing maturity levels): κάποιες ομάδες αντιλαμβάνονται τον έλεγχο ως τον εντοπισμό σφαλμάτων, ενώ κάποιες άλλες τον χρησιμοποιούν ως έναν τρόπο να δείξουν ότι το λογισμικό λειτουργεί, επίσης άλλες εφαρμόζουν ελέγχους για να ελαχιστοποιήσουν τα ρίσκα που αφορούν την ποιότητα του προϊόντος (product quality risks), κ.λπ.

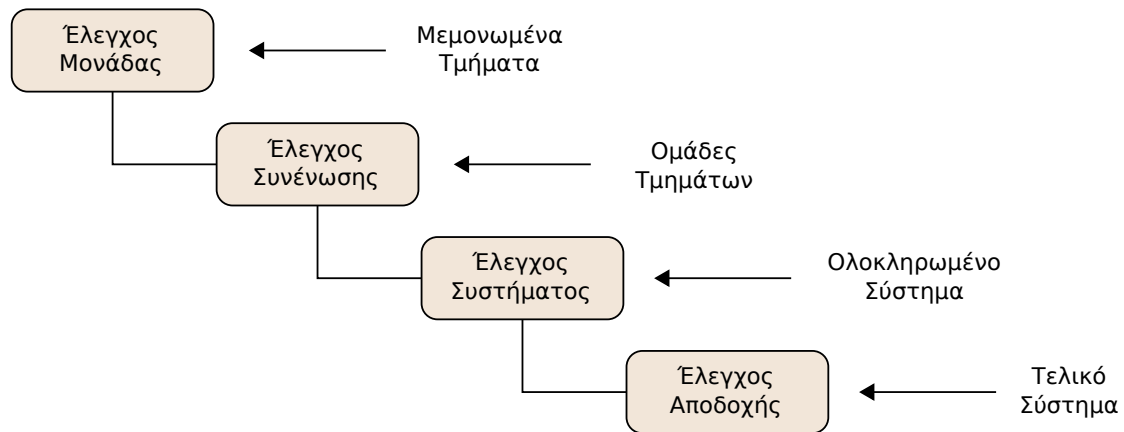
<sup>41</sup>Ο όρος αυτός στην πραγματικότητα αποδίδεται στον Thomas Edison, ο οποίος τον χρησιμοποίησε σε μια επιστολή προς έναν από τους συνεργάτες του για να περιγράψει μηχανικές δυσλειτουργίες [60]. Η χρήση του όρου “bug” στην μηχανική υπολογιστών (computer engineering) εντοπίζεται για πρώτη φορά το 1946, όταν ο Grace Hopper χρησιμοποίησε τον όρο για να περιγράψει την αιτία ενός σφάλματος στο σύστημα πληροφορικής Mark I, το οποίο αποδείχθηκε ότι ήταν κυριολεκτικά ένα “bug”, ένα έντομο δηλαδή που ήταν παγιδευμένο σε ένα ρελέ του τέρστιου αυτού συστήματος [61].

λού επιπέδου. Σημειώστε, ωστόσο, ότι ο όρος δεν καλύπτει όλες τις πιθανές καταστάσεις αποτυχίας. Σύμφωνα με τους βασικούς ορισμούς του IEEE [18], που ακολουθούνται συνήθως από την τρέχουσα βιβλιογραφία [62], ένα σφάλμα είναι στην πραγματικότητα ένας τύπος λάθους (*fault*). Εάν θεωρήσουμε τα λάθη ως ανωμαλίες που κάνουν το λογισμικό να συμπεριφέρεται εσφαλμένα, μπορούμε να διακρίνουμε μεταξύ σφαλμάτων (*bugs*) και ελαττωμάτων (*defects*), όπου τα πρώτα αναφέρονται σε προβλήματα ποιότητας του λογισμικού και τα δεύτερα αφορούν τη μη συμμόρφωση του λογισμικού με τις αρχικές προδιαγραφές. Τα λάθη οφείλονται ουσιαστικά σε λανθασμένο κώδικα που συντάζαν οι προγραμματιστές, ενώ το αποτέλεσμα ενός λάθους λογισμικού είναι αυτό που ονομάζουμε *αποτυχία* (*failure*), δηλαδή η αδυναμία του συστήματος να εκτελέσει τις απαιτούμενες λειτουργίες. Έτσι, σύμφωνα με τον ορισμό που επιλέξαμε, στις επόμενες παραγράφους μπορούμε τώρα να συζητήσουμε με σαφήνεια τις διάφορες μεθοδολογίες ελέγχου με σκοπό την εύρεση σφαλμάτων και ελαττωμάτων λογισμικού.

Ένας από τους πρώτους διαχωρισμούς που μπορεί να γίνει κατά την επιλογή μεθόδων για τον έλεγχο ενός προϊόντος λογισμικού είναι αυτός μεταξύ του ελέγχου μαύρου κουτιού (*black-box testing*) και του ελέγχου λευκού κουτιού (*white-box testing*). Κατά τον έλεγχο μαύρου κουτιού, το πρόγραμμα θεωρείται ως ένα μαύρο κουτί, και ως εκ τούτου η λειτουργικότητά του αξιολογείται χωρίς καμία γνώση της εσωτερικής υλοποίησης [58]. Αυτός ο τύπος ελέγχων είναι συχνά κατάλληλος για την αξιολόγηση της συμμόρφωσης με ορισμένες απαιτήσεις, καθώς το αντικείμενο της αξιολόγησης είναι η απόκριση του συστήματος σε διαφορετικές εισόδους<sup>42</sup>. Ο έλεγχος λευκού κουτιού, από την άλλη πλευρά, αξιολογεί την εσωτερική υλοποίηση ενός προγράμματος. Ως εκ τούτου, περιλαμβάνει περιπτώσεις ελέγχου για την κάλυψη (*coverage*) της εσωτερικής ροής του προγράμματος (*control flow*) και όχι τόσο για τις προδιαγραφές, επομένως χρησιμοποιείται κυρίως για τον εντοπισμό και την επιδιόρθωση σφαλμάτων. Ως άλλη προσέγγιση περί των διαφορών μεταξύ των ελέγχων μαύρου κουτιού και λευκού κουτιού, μπορούμε να παραπέμψουμε στην αναλογία που παρέχεται στο [58], όπου αναφέρεται ότι ο έλεγχος μαύρου κουτιού με όλες τις πιθανές εισόδους (*exhaustive input*) είναι ανάλογος με τον έλεγχο λευκού κουτιού σε όλες τις πιθανές διαδρομές προγράμματος, δηλαδή με όλες τις εντολές που εκτελούνται τουλάχιστον μία φορά (*exhaustive path*). Σημειώστε, ωστόσο, ότι και οι δύο περιπτώσεις είναι ακραίες, δεδομένου ότι είναι σχεδόν αδύνατο να σχεδιαστούν εξαντλητικοί έλεγχοι για πραγματικά έργα.

Κατά τον έλεγχο ενός προϊόντος λογισμικού πρέπει να ληφθούν διάφορες αποφάσεις. Δύο από τις πιο σημαντικές είναι το τι θα πρέπει να ελεγχθεί και το σε ποιο βαθμό λεπτομέρειας θα πρέπει να σχεδιαστεί κάθε περίπτωση ελέγχου (*test case*) και κάθε αντίστοιχη ακολουθία ελέγχων (*test suite*) που την περικλείει. Δεδομένου ότι ο έλεγχος ενδέχεται να απαιτεί σημαντικό χρόνο και προσπάθεια [15], η πρόκληση της βελτιστοποίησης των πόρων της ομάδας ανάπτυξης για το σχεδιασμό μιας κατάλληλης στρατηγικής ελέγχων (*testing strategy*) είναι ζωτικής σημασίας. Μια αποτελεσματική στρατηγική (η αποτελεσματικότητα της οποίας φυσικά εξαρτάται από κάθε συγκεκριμένη περίπτωση) μπορεί να περιλαμβάνει περιπτώσεις δοκιμών σε τέσσερα διαφορετικά επίπεδα: σε επίπεδο μονάδας, σε επίπεδο συνένωσης, σε επίπεδο συστήματος και σε επίπεδο αποδοχής από τους χρήστες [63]. Τα επίπεδα αυτά φαίνονται στο Σχήμα 2.9.

<sup>42</sup>Στην πραγματικότητα, υπάρχουν ακόμη και τεχνικές που χρησιμοποιούν απευθείας πληροφορίες από τις απαιτήσεις, προκειμένου να δημιουργήσουν αυτόματα περιπτώσεις ελέγχου (*test cases*). Οι τεχνικές αυτές ανήκουν στην περιοχή του Model-Based Testing (MBT) [15].



Σχήμα 2.9: Επίπεδα Ελέγχου και αντίστοιχα Αντικείμενα που Ελέγχονται

Ο έλεγχος μονάδας (unit testing) αφορά μεμονωμένα τμήματα λογισμικού, δηλαδή κλάσεις ή μεθόδους, και εστιάζεται στην αξιολόγηση της λειτουργικότητάς τους. Ο έλεγχος συνένωσης ή ολοκλήρωσης (integration testing) χρησιμοποιείται για να εκτιμηθεί κατά πόσον τα τμήματα λογισμικού είναι σωστά ενωμένα μεταξύ τους και επικεντρώνεται κυρίως στις διεπαφές (interfaces) των διαφόρων τμημάτων. Ο έλεγχος συστήματος (system testing) αξιολογεί τη συνένωση των τμημάτων του συστήματος στο υψηλότερο επίπεδο και επικεντρώνεται σε ελαττώματα που προκύπτουν σε αυτό το επίπεδο. Τέλος, ο έλεγχος αποδοχής (acceptance testing) αξιολογεί το λογισμικό από τη μεριά του χρήστη, υποδεικνύοντας εάν η επιθυμητή λειτουργικότητα έχει καλυφθεί σωστά.

Συνεπώς, ανάλογα με τα τμήματα του λογισμικού υπό έλεγχο, μπορεί κανείς να επιλέξει να σχεδιάσει διαφορετικές περιπτώσεις ελέγχου σε διαφορετικά επίπεδα. Επιπλέον, μια ομάδα ανάπτυξης μπορεί να επιλέξει να τρέξει διαφορετικούς ελέγχους σε κάθε φάση της ανάπτυξης λογισμικού. Μια σχετικά απλή προσέγγιση είναι να περιμένει κανείς μέχρι να αναπτυχθεί πλήρως το σύστημα και στη συνέχεια να διεξαχθούν οι απαιτούμενοι έλεγχοι. Ωστόσο, αυτή η προσέγγιση αποδεικνύεται στις περισσότερες περιπτώσεις αναποτελεσματική [15]. Μια προσέγγιση που συνήθως προτιμάται είναι ο διαδοχικός έλεγχος (incremental testing), που είναι ουσιαστικά η σειρά ενεργειών που παρουσιάζεται στο Σχήμα 2.9 αν διαβαστεί από πάνω προς τα κάτω. Στην περίπτωση αυτή, οι έλεγχοι μονάδας κατασκευάζονται μαζί με την ανάπτυξη των τμημάτων λογισμικού, ενώ οι έλεγχοι συνένωσης και συστήματος προστίθενται κάθε φορά που συνενώνονται αυτά τα τμήματα. Οι έλεγχοι αποδοχής μπορούν επίσης να πραγματοποιηθούν σε διάφορα στάδια ανάπτυξης, ωστόσο, ιδανικά, θα πρέπει να συνδέονται με τις απαιτήσεις των χρηστών και θα πρέπει να ενημερώνονται μαζί με αυτές. Από τη σκοπιά της ευέλικτης (agile) ανάπτυξης λογισμικού, ο σχεδιασμός των ελέγχων αποδοχής κατά με την υλοποίηση των τμημάτων που καλύπτουν τις σχετικές απαιτήσεις είναι μία από τις πιο αποτελεσματικές πρακτικές για την ανάπτυξη λογισμικού υψηλής ποιότητας.

Μια ακόμα πιο ακραία προσέγγιση είναι ο σχεδιασμός των περιπτώσεων ελέγχου πριν ακόμη και από την κατασκευή του ίδιου του λογισμικού. Αυτή η προσέγγιση είναι γνωστή ως *Ανάπτυξη Λογισμικού Οδηγούμενη από Ελέγχους (Test-Driven Development - TDD)*. Παρόλο που αυτός ο τρόπος ανάπτυξης εισήχθη ταυτόχρονα με έννοιες όπως ο ακραίος προγραμματισμός (extreme programming) και οι ευέλικτες μέθοδοι ανάπτυξης (agile development) [64, 65], σημειώνουμε ότι είναι επίσης πλήρως εφαρμόσιμος και σε πιο παραδοσιακά μοντέλα



ανάπτυξης λογισμικού. Γενικώς το TDD θεωρείται ως μία από τις πιο επωφελείς πρακτικές λογισμικού, καθώς έχει πολλαπλά πλεονεκτήματα [66]. Κατ' αρχάς, η συγγραφή των περιπτώσεων ελέγχου πριν από τη συγγραφή του κώδικα υποχρεώνει ουσιαστικά τον προγραμματιστή να αποσαφηνίσει τις απαιτήσεις και να καθορίσει την επιθυμητή λειτουργικότητα για κάθε τμήμα κώδικα. Επιπλέον, η σύνδεση των περιπτώσεων ελέγχου με τις απαιτήσεις μπορεί να βοηθήσει στην ανακάλυψη προβλημάτων στις απαιτήσεις σε πρώιμο στάδιο, αποφεύγοντας έτσι τη διάδοση αυτών των προβλημάτων σε μεταγενέστερα στάδια. Τέλος, η συνολική ποιότητα του πηγαίου κώδικα βελτιώνεται σημαντικά, καθώς τα ελαττώματα γενικά εντοπίζονται νωρίτερα.

## 2.4 Ανάλυση Δεδομένων Τεχνολογίας Λογισμικού

Τα υποκεφάλαια αυτού του κεφαλαίου έχουν μέχρι στιγμής εισάγει τον αναγνώστη στην τέχνη της μηχανικής λογισμικού. Αυτή η εισαγωγή, αν και κάπως γενική, καλύπτει τα δομικά στοιχεία της ανάπτυξης λογισμικού και εξυπηρετεί δύο σκοπούς: (α) να καταγράψει τις μεθοδολογίες που ακολουθούνται για την κατασκευή προϊόντων λογισμικού που είναι επαναχρησιμοποιήσιμα και υψηλής ποιότητας, και (β) να προσδιορίσει τα δεδομένα που μπορούν να αξιοποιηθούν για να υποστηρίξουν περαιτέρω τον κύκλο ανάπτυξης του λογισμικού. Σε αυτό το υποκεφάλαιο, ο στόχος μας είναι να επαναδιατυπώσουμε την πρόκληση που θέτει αυτή η διατριβή και, δεδομένου πλέον του απαραίτητου υπόβαθρου, να διερευνήσουμε πώς μπορεί να αντιμετωπιστεί καθώς παράλληλα θα σημειώνεται πρόοδος πέρα από την τρέχουσα βιβλιογραφία.

Μέχρι αυτό το σημείο, έχουμε εξηγήσει ποιες είναι οι περιοχές της ανάπτυξης λογισμικού που μπορούν να βελτιωθούν και έχουμε υπογραμμίσει επίσης τις δυνατότητες που προκύπτουν από την τεράστια ποσότητα των διαθέσιμων δεδομένων τεχνολογίας λογισμικού. Η πρόκληση που θέλουμε τώρα να αντιμετωπίσουμε είναι να αναλύσουμε αυτά τα δεδομένα για να βελτιώσουμε πρακτικά τις διάφορες φάσεις του κύκλου ζωής της ανάπτυξης λογισμικού (software development lifecycle). Ο σχετικός τομέας έρευνας που συστάθηκε για να αντιμετωπίσει αυτή την πρόκληση είναι ευρέως γνωστός ως “Εξόρυξη Δεδομένων Τεχνολογίας Λογισμικού (Data Mining for Software Engineering)”. Έχουμε ήδη αναλύσει τα στοιχεία που σχετίζονται με την Τεχνολογία Λογισμικού, έτσι στην επόμενη ενότητα θα επικεντρωθούμε στην Εξόρυξη Δεδομένων (Data Mining), ενώ στην ενότητα 2.4.2 θα εξηγήσουμε πώς συνδυάζονται αυτά τα πεδία έτσι ώστε οι μέθοδοι εξόρυξης δεδομένων να ενισχύσουν ουσιαστικά τη διαδικασία ανάπτυξης λογισμικού. Τέλος, στην ενότητα 2.4.3, αναλύουμε τις δυνατότητες της Εξόρυξης Δεδομένων για την υποστήριξη της επαναχρησιμοποίησης και υποδεικνύουμε τις συνεισφορές μας σε αυτό τον άξονα.

### 2.4.1 Εξόρυξη Δεδομένων

Η έννοια της ανάλυσης δεδομένων δεν είναι νέα: τον τελευταίο καιρό, όμως, με την έλευση της εποχής της πληροφορίας (Information Age), βρισκόμαστε σε μια δύσκολη θέση όπου το χάσμα ανάμεσα στη δημιουργία νέων δεδομένων και στην κατανόηση αυτών διαρκώς μεγαλώνει [67]. Ως εκ τούτου, η αποτελεσματική ανάλυση δεδομένων έχει γίνει πιο απαραίτητη από ποτέ. Η κατανόηση και η αποτελεσματική χρήση των δεδομένων για την εξαγωγή γνώσης (knowledge extraction) είναι το αποτέλεσμα μιας διαδικασίας που ονομάζεται

*ανάλυση δεδομένων (data analysis)*. Η ανάλυση δεδομένων εφάπτεται κυρίως σε επιστημονικούς τομείς όπως η στατιστική (statistics), που μπορούν φυσικά να αποδειχθούν ιδιαίτερα χρήσιμοι σε ορισμένα σενάρια, ωστόσο το πεδίο εφαρμογής τους είναι συνήθως περιορισμένο. Ειδικότερα, όπως σημειώνεται από τον Tan και τους συνεργάτες του [68], οι παραδοσιακές τεχνικές ανάλυσης δεδομένων δεν μπορούν εύκολα να χειριστούν ετερογενή και σύνθετα δεδομένα (heterogeneous and complex data), δεν κλιμακώνονται (scaling) σε δεδομένα μεγάλου όγκου (ή ακόμη και σε δεδομένα μεγάλης διαστασιμότητας - dimensionality) και, το σημαντικότερο, περιορίζονται στον παραδοσιακό κύκλο της κατασκευής και της απόδειξης/κατάρριψης υποθέσεων χειροκίνητα (manual hypotheses proving/disproving), κάτι που δεν επαρκεί για την ανάλυση πραγματικών δεδομένων και την εξαγωγή γνώσης.

Έτσι, αυτό που διαφοροποιεί την εξόρυξη δεδομένων από τις παραδοσιακές τεχνικές ανάλυσης δεδομένων είναι η χρήση αλγορίθμων από άλλα επιστημονικά πεδία για την υποστήριξη της εξαγωγής γνώσης από δεδομένα. Η διεπιστημονική φύση του πεδίου μας προτρέπει να ακολουθήσουμε τον ορισμό που προτείνουν ο Tan και οι συνεργάτες του [68]:

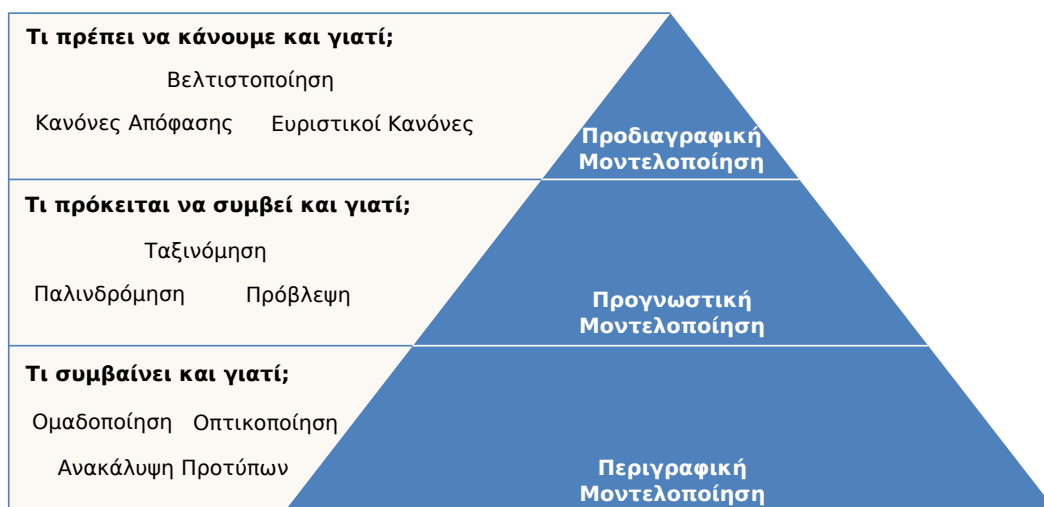
Η εξόρυξη δεδομένων είναι μια τεχνολογία που συνδυάζει τις παραδοσιακές μεθόδους ανάλυσης δεδομένων με προηγμένους αλγόριθμους για την επεξεργασία δεδομένων μεγάλου όγκου.

Ο ορισμός αυτός υιοθετείται διότι υπογραμμίζει την *προηγμένη (sophisticated)* φύση των τεχνικών εξόρυξης, μια πτυχή που συνήθως τονίζεται από διάφορους ερευνητές<sup>43</sup>. Και αυτή ακριβώς η προηγμένη φύση είναι αποτέλεσμα της ανάμιξης των παραδοσιακών μεθόδων ανάλυσης δεδομένων με τεχνικές από τα πεδία της Τεχνητής Νοημοσύνης (Artificial Intelligence - AI), της Μηχανικής Μάθησης (Machine Learning) και της Αναγνώρισης Προτύπων (Pattern Recognition). Ως εκ τούτου, το πεδίο της εξόρυξης δεδομένων περιλαμβάνει στατιστικές μεθόδους, όπως η δειγματοληψία (sampling), ο έλεγχος υποθέσεων (hypothesis testing) κ.α., καθώς και μεθόδους τεχνητής νοημοσύνης, όπως θεωρίες μάθησης (learning theories) [70], τεχνικές αναγνώρισης προτύπων (pattern recognition) [71], τεχνικές ανάκτησης πληροφοριών [72], ακόμα και εξελικτική υπολογιστική (evolutionary computing) [73].

Με βάση την παραπάνω ανάλυση, το πεδίο της Εξόρυξης Δεδομένων δείχνει αρκετά ευρύ. Ωστόσο, οι κύριες αρχές του μπορούν να συνοψιστούν στις εργασίες που συνήθως εκτελούνται, οι οποίες παρέχουν επίσης μια αποτελεσματική κατηγοριοποίηση των χρησιμοποιούμενων τεχνικών. Μια ευρέως αποδεκτή κατηγοριοποίηση [68, 74] είναι ο διαχωρισμός των εργασιών της εξόρυξης δεδομένων σε δύο κατηγορίες: σε εργασίες περιγραφικής μοντελοποίησης (descriptive modeling tasks) και σε εργασίες προγνωστικής μοντελοποίησης (predictive modeling tasks). Η πρώτη κατηγορία αναφέρεται στην εξαγωγή προτύπων (patterns) από δεδομένα προκειμένου να συνοψιστούν οι υποκείμενες σχέσεις, ενώ στόχος των εργασιών της δεύτερης κατηγορίας είναι η πρόβλεψη (prediction) των τιμών ορισμένων χαρακτηριστικών δεδομένων (data attributes) με βάση τη γνώση που εξάγεται από τα δεδομένα. Σε ορισμένες περιπτώσεις, η προγνωστική μοντελοποίηση μπορεί να θεωρηθεί ως επέκταση της περιγραφικής μοντελοποίησης (αν και αυτό δεν συμβαίνει πάντα), καθώς πρέπει πρώτα κάποιος να καταλάβει τα δεδομένα και στη συνέχεια να ζητήσει ορισμένες προβλέψεις. Μια επέκταση προς αυτήν την κατεύθυνση είναι η κατηγο-

<sup>43</sup>Ένας εναλλακτικός ορισμός που παρέχεται από τον Hand και τους συνεργάτες του [69] περιλαμβάνει τις έννοιες της “εύρεσης μη ορατών σχέσεων (finding unsuspected relationship)” και της “σύνοψης δεδομένων με νέους τρόπους (summarizing data in novel ways)”. Αυτές οι έννοιες υποδεικνύουν τις πραγματικές προκλήσεις που διαφοροποιούν την εξόρυξη δεδομένων από την παραδοσιακή ανάλυση δεδομένων.

ρία εργασιών προδιαγραφικής μοντελοποίησης (prescriptive modeling tasks), η οποία μπορεί να θεωρηθεί ως ο συνδυασμός των άλλων δύο κατηγοριών, καθώς δεν περιορίζεται στις προβλέψεις αλλά επιπλέον προτείνει συγκεκριμένες αποφάσεις με βάση τις προβλέψεις. Επομένως, θεωρείται πολύ χρήσιμη στη λήψη αποφάσεων σε επιχειρησιακό πλαίσιο (business intelligence-related decision-making). Από το πρίσμα της Επιχειρηματικής Αναλυτικής (Business Analytics) [75], η περιγραφική μοντελοποίηση απαντά στο ερώτημα του τι συμβαίνει, η προγνωστική μοντελοποίηση απαντά στα ερωτήματα του τι πρόκειται να συμβεί και για ποιο λόγο και, τέλος, η προδιαγραφική μοντελοποίηση απαντά στα ερωτήματα του τι πρέπει να κάνουμε και γιατί. Αυτές οι κατηγορίες εργασιών εξόρυξης δεδομένων παρουσιάζονται στο Σχήμα 2.10 μαζί με κάποια ενδεικτικά είδη τεχνικών για κάθε κατηγορία.



Σχήμα 2.10: Εργασίες Εξόρυξης Δεδομένων και Ενδεικτικές Τεχνικές

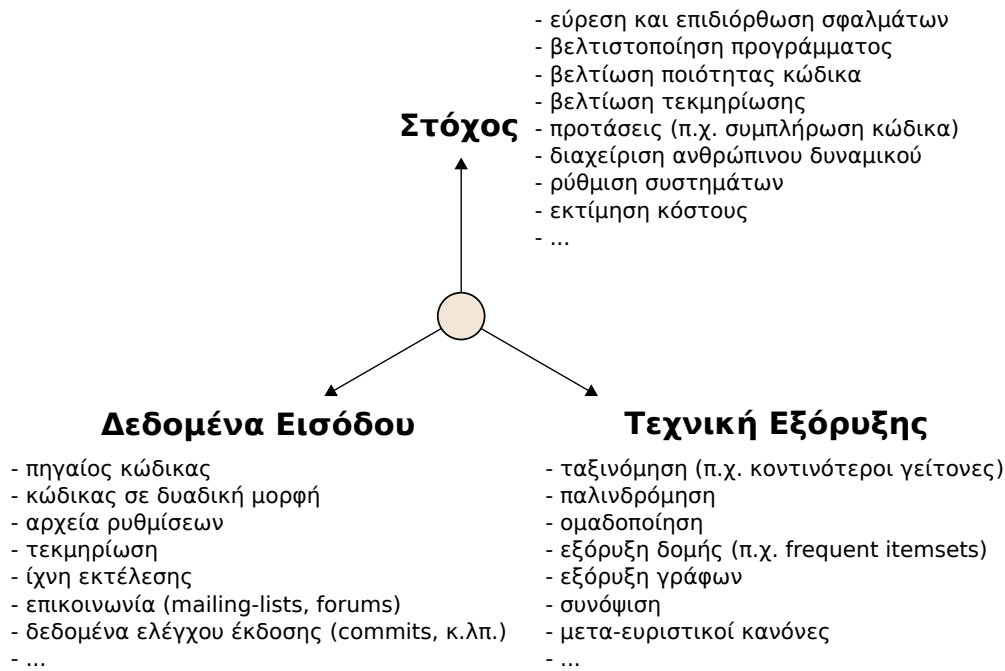
Οι τεχνικές που παρουσιάζονται στο αριστερό μέρος του Σχήματος 2.10 είναι μόνο ένα δείγμα αυτών που μπορούν να είναι χρήσιμες για την εξόρυξη δεδομένων. Παρόλο που δεν επιδιώκουμε (και δεν θεωρούμε απαραίτητο) να αναλύσουμε όλες αυτές τις τεχνικές εδώ, θα παρέχουμε μια γενική περιγραφή για ορισμένες ενδιαφέρουσες μεθόδους και θα παραπέμψουμε τον αναγνώστη που ενδιαφέρεται περαιτέρω σε πιο σχετικά κείμενα. Αρχικά, όσον αφορά την περιγραφική μοντελοποίηση, ο βασικός στόχος είναι η εξαγωγή γνώσης από τα δεδομένα. Ως εκ τούτου, μια πρώτη σκέψη θα ήταν η οπτικοποίηση (visualization) των δεδομένων, που μπορεί να είναι ιδιαίτερα χρήσιμη, αλλά πολλές φορές ενδέχεται να είναι και αρκετά δύσκολο να γίνει. Θα πρέπει κανείς να λάβει υπόψη του τη διαστασιμότητα (dimensionality) των δεδομένων και να παράγει οπτικοποιήσεις που απεικονίζουν χρήσιμη πληροφορία, ενδεχομένως απορρίπτοντας περιττά δεδομένα ή δεδομένα χωρίς κάποιο νόημα. Έτσι, μια άλλη πολύ σημαντική πρόκληση στην περιγραφική μοντελοποίηση είναι ο προσδιορισμός του τι αξίζει κανείς να δείξει στον αναλυτή. Η πρόκληση αυτή αντιμετωπίζεται με την *ανακάλυψη προτύπων (pattern discovery)*, που θεωρείται ως μία από τις πιο κρίσιμες εφαρμογές της εξόρυξης δεδομένων [67]. Σε αυτό το πλαίσιο, μπορούμε επίσης να τοποθετήσουμε μεθόδους μη επιβλεπόμενης μάθησης (unsupervised learning), όπως η ομαδοποίηση (clustering) [76], η οποία είναι μια τεχνική που μπορεί να οργανώσει τα δεδομένα σε ομάδες και έτσι να βοηθήσει στην κατανόησή τους και στον εντοπισμό συσχετίσεων.

Τέλος, για την εξαγωγή περιγραφικής γνώσης, μπορούν επίσης να χρησιμοποιηθούν διάφορες προσεγγίσεις μοντελοποίησης (modeling), συμπεριλαμβανομένων, για παράδειγμα, μεθόδων για την προσαρμογή κατανομών σε δεδομένα (distribution fitting), μεικτά μοντέλα (mixture models), κ.λπ. [71].

Η προσαρμογή κατανομών σε δεδομένα χρησιμοποιείται συχνότερα για εργασίες προγνωστικής μοντελοποίησης και είναι σχετική με τους αλγόριθμους παλινδρόμησης (regression), οι οποίοι είναι αλγόριθμοι επιβλεπόμενης μάθησης (supervised learning). Παρόμοια με την παλινδρόμηση, η οποία μπορεί να οριστεί ως η διαδικασία προσαρμογής ενός μοντέλου σε ορισμένα δεδομένα, μπορεί κανείς να ορίσει και την πρόβλεψη (forecasting), η οποία ωστόσο προϋποθέτει ότι τα δεδομένα είναι σε μορφή ακολουθίας (sequential) [71]. Μια άλλη αρκετά μεγάλη κατηγορία τεχνικών που σχετίζονται με την προγνωστική μοντελοποίηση είναι οι τεχνικές ταξινόμησης (classification). Τόσο στην παλινδρόμηση (και στην πρόβλεψη) όσο και στην ταξινόμηση, ο αρχικός στόχος είναι να κατασκευαστεί ένα μοντέλο με βάση υπάρχοντα δεδομένα, έτσι ώστε να μπορούν να ληφθούν ορισμένες αποφάσεις για νέες εγγραφές δεδομένων. Το αποτέλεσμα της παλινδρόμησης, ωστόσο, είναι μια τιμή που βρίσκεται σε ένα συνεχές εύρος (continuous range), ενώ αντίθετα σε ένα πρόβλημα ταξινόμησης οι νέες εγγραφές δεδομένων θα πρέπει να κατηγοριοποιηθούν σε μία συγκεκριμένη κλάση (class) από ένα προκαθορισμένο σύνολο κλάσεων [77]. Τέλος, η προδιαγραφική μοντελοποίηση περιλαμβάνει τεχνικές παρόμοιες με εκείνες που ήδη συζητήθηκαν, ωστόσο εμπεριέχει επιπλέον τη λήψη αποφάσεων. Ως εκ τούτου, οι αλγόριθμοι μηχανικής μάθησης που επιλέγονται σε αυτή την περίπτωση μπορεί ως έξοδο σύνολα κανόνων (sets of rules) ή ευριστικοί κανόνες (heuristics). Επίσης, επικεντρώνονται κυρίως στη βελτιστοποίηση μοντέλων. Έτσι, η προδιαγραφική μοντελοποίηση συνήθως περιλαμβάνει ερμηνεύσιμες (interpretable) τεχνικές ταξινόμησης/παλινδρόμησης, όπως π.χ. δέντρα αποφάσεων (decision trees), συστήματα υποστήριξης αποφάσεων βασισμένα σε κανόνες (rule-based decision support systems), όπως π.χ. μαθάνοντα συστήματα ταξινομητών (Learning Classifier Systems) [78], αλγορίθμους υπολογιστικής νοημοσύνης (computational intelligence algorithms), όπως π.χ. ασαφή μοντέλα (fuzzy inference models), κ.λπ.

#### 2.4.2 Εξόρυξη Δεδομένων Τεχνολογίας Λογισμικού

Έχοντας πλέον μια βασική κατανόηση των πεδίων της Τεχνολογίας Λογισμικού και της Εξόρυξης Δεδομένων, το επόμενο βήμα είναι να συνδυάσουμε αυτά τα δύο πεδία για να ενισχύσουμε τη διαδικασία ανάπτυξης λογισμικού. Η ενότητα αυτή εμπεριέχει τον ορισμό αυτής της ευρύτερης ερευνητικής περιοχής, υποδεικνύοντας τους διαφορετικούς άξονές της, καθώς και τις τρέχουσες προκλήσεις που έχουν προκύψει εντός αυτής και οι οποίες αποτελούν αντικείμενο έρευνας της παρούσας διατριβής. Πρακτικά, η περιοχή αυτή περιλαμβάνει την εφαρμογή τεχνικών εξόρυξης δεδομένων σε δεδομένα τεχνολογίας λογισμικού, με σκοπό την εξαγωγή γνώσης που μπορεί να αξιοποιηθεί για να βελτιώσει τη διαδικασία της ανάπτυξης λογισμικού. Από τον ορισμό αυτό αντιλαμβάνεται κανείς ότι υπάρχει πλήθος εφαρμογές στην εν λόγω περιοχή. Σύμφωνα με τον Xie και τους συνεργάτες του [79], μπορούμε να διακρίνουμε τις διάφορες ερευνητικές εργασίες με βάση τρεις άξονες: τον στόχο τους (goal), τα δεδομένα εισόδου που αξιοποιούν (input data), και την τεχνική εξόρυξης που εφαρμόζεται (mining technique). Αυτός ο διαχωρισμός οπτικοποιείται καλύτερα από τον Monperrus [80], όπως στο Σχήμα 2.11.



Σχήμα 2.11: Εξόρυξη Δεδομένων Τεχνολογίας Λογισμικού: Στόχοι, Δεδομένα Εισόδου και Τεχνικές Εξόρυξης

Όπως μπορεί κανείς να παρατηρήσει, το Σχήμα 2.11 περιλαμβάνει σχεδόν όλα όσα συζητήθηκαν μέχρι στιγμής σε αυτό το κεφάλαιο. Περιλαμβάνει τις διαφορετικές πηγές και τύπους δεδομένων τεχνολογίας λογισμικού, τους αλγορίθμους που χρησιμοποιούνται στον διεπιστημονικό τομέα της εξόρυξης δεδομένων και φυσικά ένα σύνολο εργασιών που αποτελούν μέρος της διαδικασίας ανάπτυξης λογισμικού. Για να αναλύσουμε περαιτέρω την εν λόγω περιοχή, επιλέξαμε να ακολουθήσουμε μια κατηγοριοποίηση παρόμοια με αυτή στο [81], σύμφωνα με την οποία οι διάφορες εργασίες διακρίνονται σύμφωνα με τις σχετικές φάσεις ανάπτυξης λογισμικού<sup>44</sup>. Από μια γενική σκοπιά, μπορούμε να επικεντρωθούμε στις δραστηριότητες του Καθορισμού Απαιτήσεων και της Εξαγωγής Προδιαγραφών (Requirements Elicitation and Specification Extraction), της Σχεδίασης και Ανάπτυξης Λογισμικού (Software Design and Development) και του Ελέγχου και της Διασφάλισης Ποιότητας (Testing and Quality Assurance). Η τρέχουσα βιβλιογραφία που είναι σχετική με αυτές τις κατηγορίες αναλύεται στις επόμενες παραγράφους προκειμένου να οριστεί καλύτερα η ερευνητική περιοχή στο σύνολό της και να προσδιοριστούν οι βασικοί άξονες της παρούσας διατριβής. Αναλύουμε ξεχωριστά κάθε τομέα έρευνας και επεξηγούμε τι μπορεί κανείς να κάνει (ή τι δεν μπορεί να κάνει) σήμερα χρησιμοποιώντας τα κατάλληλα δεδομένα μαζί με τις προτεινόμενες μεθόδους (ή με τις μεθόδους που δεν έχουν ακόμα προταθεί ή με τις μεθόδους που προτείνουμε εμείς στα επόμενα κεφάλαια). Σημειώστε, ωστόσο, ότι οι ακόλουθοι παράγραφοι δεν αποτελούν εξαντλητικές ανασκοπήσεις της βιβλιογραφίας σε κάθε έναν από τους τομείς που συζητήθηκαν. Αντί αυτού, παρέχουν αυτό που μπορούμε να ονομάσουμε την τρέχουσα ερευνητική πρακτική (current research practice)· για κάθε περιοχή, προσδιορίζεται ο σκοπός της και περιγράφεται ένα αντιπροσωπευτικό μέρος των σημερινών

<sup>44</sup>Ένας άλλος πιθανός διαχωρισμός είναι σύμφωνα με τον τύπο των δεδομένων εισόδου, τα οποία, σύμφωνα με τον Xie και τους συνεργάτες του [79], εμπίπτουν σε τρεις μεγάλες κατηγορίες: ακολουθίες (π.χ. ίχνη εκτέλεσης), γράφοι (π.χ. γράφοι κλήσεων - call graphs) και κείμενο (π.χ. τεκμηρίωση).

κατευθύνσεων της έρευνας, ενώ δίνεται βάση στον τρόπο με τον οποίο αντιμετωπίζονται οι βασικές προκλήσεις. Για μια εκτεταμένη ανασκόπηση της τρέχουσας βιβλιογραφίας σε κάθε έναν από αυτούς τους τομείς, ο αναγνώστης παραπέμπεται σε κάθε αντίστοιχο κεφάλαιο αυτής της διατριβής.

**Καθορισμός Απαιτήσεων και Εξαγωγή Προδιαγραφών** Ο καθορισμός των απαιτήσεων μπορεί να θεωρηθεί ως το πρώτο βήμα της διαδικασίας ανάπτυξης λογισμικού, καθώς το να γνωρίζει κανείς τι πρέπει να κατασκευάσει είναι πρακτικά απαραίτητο πριν ξεκινήσει η ανάλυση. Η έρευνα σε αυτόν τον τομέα επικεντρώνεται στη βελτιστοποίηση της διαδικασίας προσδιορισμού απαιτήσεων και στη συνέχεια στην εξαγωγή προδιαγραφών. Ως εκ τούτου, συνήθως περιλαμβάνει τη διαδικασία δημιουργίας ενός μοντέλου για την αποθήκευση και ευρετηριοποίηση (indexing) απαιτήσεων, καθώς και μια μεθοδολογία για την εξόρυξη τους για διάφορους σκοπούς. Έχοντας ένα επαρκές σύνολο απαιτήσεων και ένα μοντέλο που καταγράφει τη δομή και τη σημασιολογία τους (μέσω προσεκτικού annotation), μπορεί κανείς να εφαρμόσει τεχνικές εξόρυξης για την επικύρωση απαιτήσεων (requirements validation) [82, 83], την πρόταση νέων απαιτήσεων (requirements recommendation) [84–86], την αναγνώριση εξαρτήσεων μεταξύ απαιτήσεων (dependency detection) [87], κ.α. Αν υπάρχουν δεδομένα και από άλλες πηγές, όπως π.χ. οι προτιμήσεις των ενδιαφερόμενων (stakeholder preference), ο κώδικας της υλοποίησης, η τεκμηρίωση του κώδικα, κ.λπ., τότε μπορούν να αντιμετωπιστούν ακόμα περισσότερα προβλήματα, όπως π.χ. η σύνδεση των απαιτήσεων με τους ενδιαφερόμενους (requirements distribution to stakeholders) [87], ή ακόμα και η ανάκτηση εξαρτήσεων μεταξύ των απαιτήσεων και των τμημάτων λογισμικού (requirements tracing) [88].

Αν και οι προαναφερθείσες ιδέες είναι αρκετά ενδιαφέρουσες, συνήθως δεν διερευνούνται επαρκώς για δύο λόγους. Ο πρώτος λόγος είναι ότι οι απαιτήσεις συνήθως παρέχονται σε μορφές που είναι δύσκολο να εξορυχθούν (π.χ. κείμενο σε φυσική γλώσσα, διάφοροι τύποι διαγραμμάτων, κ.α.), ενώ ο δεύτερος λόγος είναι ότι στην παρούσα βιβλιογραφία δεν υπάρχουν πολλά “σημαδεμένα” (annotated) σύνολα δεδομένων. Ως εκ τούτου, οι περισσότερες σύγχρονες προσεγγίσεις περιορίζονται σε μοντέλα που αφορούν συγκεκριμένους τομείς (domain-specific) και δεν μπορούν να υποστηρίξουν πιο αφηρημένη σημασιολογία. Σε αυτή τη διατριβή, κατασκευάζουμε ένα νέο μοντέλο που δεν περιορίζεται στο λεξιλόγιο κάποιου τομέα (είναι δηλαδή domain-agnostic) και υποστηρίζει την αποθήκευση απαιτήσεων από διαφορετικές πηγές (multimodal requirements). Επιπλέον, το μοντέλο μας διευκολύνει τη χρήση τεχνικών εξόρυξης για ορισμένες από τις εφαρμογές που περιγράφηκαν παραπάνω.

**Σχεδίαση και Ανάπτυξη Λογισμικού** Η ανάπτυξη ενός προϊόντος είναι πρακτικά η φάση που έχει τη μεγαλύτερη προστιθέμενη αξία, επομένως είναι σημαντικό να εκτελείται σωστά. Οι μεθοδολογίες εξόρυξης σε αυτή τη φάση είναι πολυάριθμες, καλύπτουν αρκετά διαφορετικές κατευθύνσεις και έχουν πολλές εφαρμογές<sup>45</sup>. Τα δεδομένα εισόδου περιλαμβάνουν συνήθως τον πηγαίο κώδικα και ενδεχομένως περαιτέρω πληροφορίες σχετικές με το έργο (π.χ. τεκμηρίωση, αρχεία readme), ενώ τα μοντέλα εξόρυξης χρησιμοποιούν διάφορες αναπαραστάσεις, όπως Αφηρημένα Συνδετικά Δένδρα (Syntax Abstract Trees - ASTs), Γράφους Ελέγχου Ροής (Control Flow Graphs - CFGs), Γράφους Εξάρτησης Προγράμματος (Program

<sup>45</sup> Αναλούουμε εδώ ορισμένες δημοφιλείς κατευθύνσεις, ωστόσο προφανώς δεν τις εξαντλούμε. Ο αναγνώστης παραπέμπεται στο [89] για μια εκτεταμένη ανασκόπηση των μεθόδων ανάλυσης πηγαίου κώδικα.

Dependency Graphs - PDGs), κ.α. Καθώς κινούμαστε προς ένα μοντέλο ανάπτυξης λογισμικού προσανατολισμένο στην επαναχρησιμοποίηση με χρήση τμημάτων (reuse-oriented component-based software development), μία από τις πιο δημοφιλείς προκλήσεις σε αυτήν την περιοχή είναι η εύρεση επαναχρησιμοποιήσιμων λύσεων από διαδικτυακές πηγές. Έτσι, πολλές ερευνητικές προσπάθειες προσανατολίζονται στη σχεδίαση μηχανών αναζήτησης κώδικα (Code Search Engines - CSEs) [90–92] ή και πιο εξειδικευμένων συστημάτων προτάσεων (Recommendation Systems in Software Engineering - RSSEs) [93–95], τα οποία μπορούν να χρησιμοποιηθούν για τον εντοπισμό επαναχρησιμοποιήσιμων τμημάτων λογισμικού. Αυτές οι προσεγγίσεις εκτελούν εξόρυξη κώδικα με βάση τη σύνταξη (syntax-aware mining) και κατατάσσουν τα τμήματα κώδικα σύμφωνα με ορισμένα κριτήρια, έτσι ώστε ο προγραμματιστής να μπορέσει εύκολα να επιλέξει εκείνο που είναι πιο χρήσιμο στην περίπτωση του. Τέλος, υπάρχουν επίσης συστήματα που έχουν σχεδιαστεί με βάση το πρότυπο της ανάπτυξης λογισμικού οδηγούμενης από ελέγχους, τα οποία ελέγχουν περαιτέρω αν τα τμήματα που ανακτήθηκαν πληρούν τις σχετικές απαιτήσεις [96–98]. Τα συστήματα αυτά είναι γνωστά και ως συστήματα για επαναχρησιμοποίηση οδηγούμενη από ελέγχους (Test-Driven Reuse - TDR).

Αν και τα παραπάνω συστήματα μπορεί να είναι αρκετά αποτελεσματικά όσον αφορά την επαναχρησιμοποίηση λογισμικού, ωστόσο έχουν και αρκετούς περιορισμούς. Για παράδειγμα, οι περισσότερες CSEs δεν υποστηρίζουν τη διενέργεια ερωτημάτων με βάση τη σύνταξη (syntax-aware queries), ενώ επιπλέον δεν υποστηρίζουν την αυτοματοποιημένη ανάκτηση κώδικα από υπηρεσίες φιλοξενίας κώδικα (code hosting services), οπότε τα τμήματα κώδικα που επιστρέφουν περιορίζονται σε αυτά που είναι αποθηκευμένα στο ευρετήριο τους (index). Όσον αφορά τα RSSEs, οι αδυναμίες τους εντοπίζονται κυρίως στην έλλειψη σημασιολογίας (semantics) κατά την αναζήτηση τμημάτων, στην έλλειψη (ή στην ανεπάρκεια) των δυνατοτήτων μετασχηματισμού κώδικα (code transformation capabilities) και στον τρόπο κατάταξης των αποτελεσμάτων τους που λαμβάνει υπόψη μόνο την αξιολόγηση της λειτουργικής πλευράς τους [99]. Σε αυτή τη διατριβή, σχεδιάζουμε και αναπτύσσουμε μία CSE και δύο RSSEs, με σκοπό να αντιμετωπίσουμε τις παραπάνω αδυναμίες. Τα συστήματά μας χρησιμοποιούν εξόρυξη με βάση τη σύνταξη, εκτελούν μετασχηματισμούς κώδικα, ενώ επιπλέον παρέχουν χρήσιμες πληροφορίες σχετικά με τη δυνατότητα επαναχρησιμοποίησης των τμημάτων, έτσι ώστε ο προγραμματιστής να μπορεί να επιλέξει το βέλτιστο κώδικα για την περίπτωση του και να το ενσωματώσει εύκολα στον πηγαίο κώδικά του.

Η έρευνα σε αυτήν την περιοχή μπορεί επίσης να κατηγοριοποιηθεί με βάση τον τύπο και το μέγεθος του κώδικα προς εξόρυξη. Εκτός από τα τμήματα κώδικα (π.χ. κλάσεις, σύνολα κλάσεων), υπάρχει αυξανόμενο ενδιαφέρον για την εύρεση μικρών τμημάτων κώδικα (snippets) που αποτελούν λύσεις σε κοινά προγραμματιστικά ερωτήματα ή ως παραδείγματα χρήσης ενός API. Σε αυτό τον άξονα, υπάρχει σημαντική ερευνητική συνεισφορά, με τις διάφορες προσεγγίσεις να μπορούν να διακριθούν με βάση το πρόβλημα που αντιμετωπίζουν και τα δεδομένα που χρησιμοποιούν. Υπάρχουν προσεγγίσεις για την κατασκευή χρήσιμων παραδειγμάτων για APIs βιβλιοθηκών [100, 101], προσεγγίσεις που εξάγουν χρήσιμα snippets από υπηρεσίες ερωταπαντήσεων [102], καθώς και προσεγγίσεις που προτείνουν snippets για κοινά προγραμματιστικά ερωτήματα [103, 104]. Η κύρια πρόκληση με όλες τις παραπάνω προσεγγίσεις είναι εάν μπορούν πράγματι να είναι αποτελεσματικές και να προσφέρουν προστιθέμενη αξία στον προγραμματιστή. Ως εκ τούτου, σε αυτή τη διατριβή,

εστιάζουμε σε ορισμένες μετρικές που μπορούν να αξιολογήσουν ποσοτικά αν οι προτεινόμενες λύσεις είναι σωστές (π.χ. μέτρηση της συνάφειας των εξαγόμενων snippets), αποτελεσματικές (π.χ. διαδικασία αναζήτησης σε συστήματα ερωταπαντήσεων για τη βελτίωση υπάρχοντα κώδικα) και πρακτικά χρήσιμες (π.χ. διερεύνηση του ποιος είναι ο καλύτερος τρόπος παρουσίασης των αποτελεσμάτων σε ένα σύστημα προτάσεων snippets).

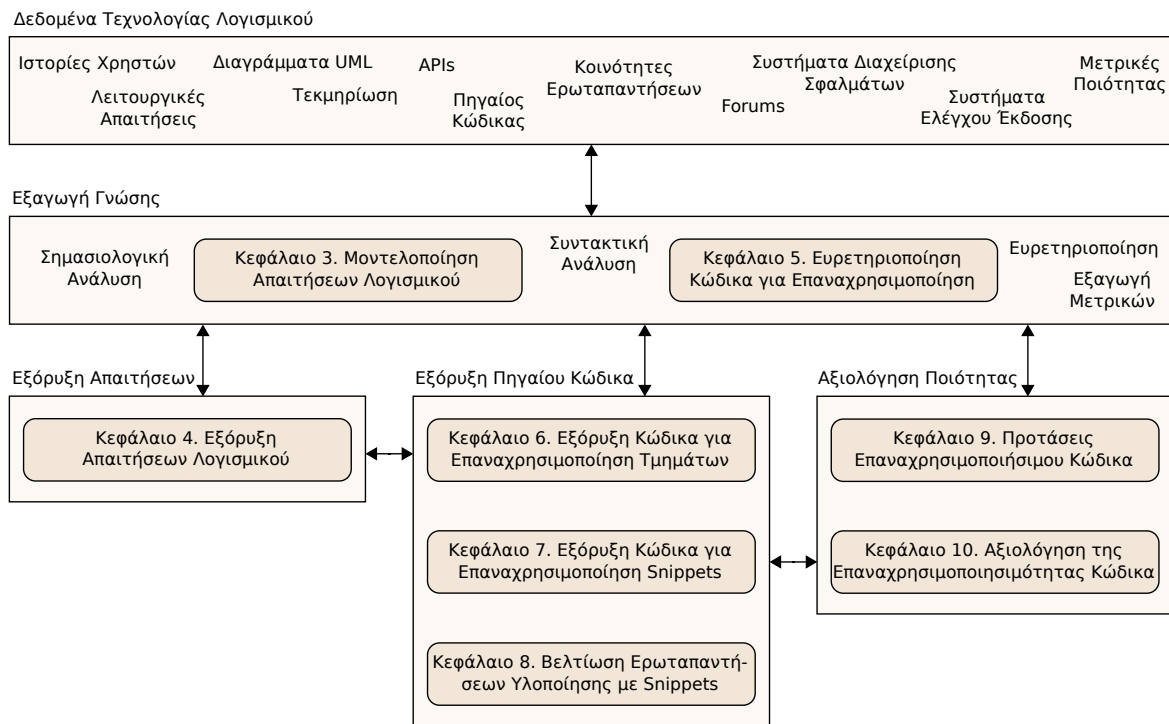
**Έλεγχος και Διασφάλιση Ποιότητας** Η τρίτη και τελευταία περιοχή έρευνας που αναλύουμε ως μέρος του ευρύτερου πεδίου της εξόρυξης λογισμικού είναι αυτή της διασφάλισης ποιότητας. Δεδομένου ότι η ποιότητα σχετίζεται με όλες τις φάσεις ανάπτυξης ενός έργου λογισμικού, η έρευνα σε αυτήν την περιοχή καλύπτει επίσης όλες αυτές τις φάσεις, όπως επίσης και την ίδια την εξέλιξη του λογισμικού (software evolution) [105]. Επικεντρωνόμαστε στις προσεγγίσεις στατικής ανάλυσης, όπως αυτή ορίστηκε στην ενότητα 2.3.2. Παρόλο που η στατική ανάλυση μπορεί να χρησιμοποιηθεί για διάφορους σκοπούς, όπως η αξιολόγηση της ποιότητας τμημάτων λογισμικού (component quality assessment), ο εντοπισμός σφαλμάτων (bug localization), κ.α., στην περίπτωση μας εστιάζουμε στην αξιολόγηση της ποιότητας τμημάτων λογισμικού. Για να αναδείξουμε τη σημασία της αξιολόγησης της ποιότητας κώδικα, κατασκευάσαμε αρχικά ένα σύστημα επαναχρησιμοποίησης τμημάτων (component reuse system) που χρησιμοποιεί ένα μοντέλο ποιότητας για να αξιολογεί τα προτεινόμενα τμήματα όχι μόνο από λειτουργική (functional) αλλά και από ποιοτική (quality) άποψη.

Σε αυτό το πλαίσιο, η πρόκληση είναι να κατασκευαστεί ένα μοντέλο που θα καθορίζει ένα σύνολο από όρια (metric thresholds) για τις δεδομένες μετρικές στατικής ανάλυσης. Το μοντέλο αυτό μπορεί στη συνέχεια να χρησιμοποιηθεί για την αξιολόγηση της ποιότητας τμημάτων κώδικα (code quality assessment) [106–108] ή για την εξεύρεση σημείων στον κώδικα που είναι πιθανό να έχουν σφάλματα (finding defect-prone locations) [109, 110], ελέγχοντας εάν οι τιμές των μετρικών υπερβαίνουν αυτά τα όρια. Αρκετές προσεγγίσεις σε αυτό τον άξονα βασίζονται στη γνώση εμπειρογνομόνων (expert help) για τον καθορισμό αποδεκτών ορίων [111, 112]. Ωστόσο, η βοήθεια από ειδικούς ποιότητας είναι συχνά υποκειμενική, εξαρτάται σε μεγάλο βαθμό από την κάθε περίπτωση έργου λογισμικού, ενώ μπορεί και να μην είναι πάντοτε διαθέσιμη [113]. Ως εκ τούτου, υπάρχουν αρκετές προσεγγίσεις που έχουν ως σκοπό τον αυτόματο προσδιορισμό των ορίων των μετρικών με βάση ένα σύνολο από εξακριβωμένες τιμές ποιότητας (ground truth) [106–108]. Δεδομένου όμως ότι είναι δύσκολο να βρεθεί ένα αντικειμενικό μέτρο της ποιότητας λογισμικού, αυτές οι προσεγγίσεις καταλήγουν επίσης συχνά σε βοήθεια εμπειρογνομόνων. Για να ξεπεράσουμε την ανάγκη για εξωτερική βοήθεια ειδικών ποιότητας, στην παρούσα διατριβή, χρησιμοποιούμε ως ground truth τη δημοτικότητα (popularity) και την επαναχρησιμοποιησιμότητα (reusability) των τμημάτων λογισμικού, όπως αυτές εκφράζονται από online υπηρεσίες, οι οποίες παρέχουν έτσι έναν δείκτη για την ποιότητα όπως γίνεται αντιληπτή από τους προγραμματιστές (developer-perceived quality). Με βάση αυτό το ground truth, κατασκευάζουμε ένα σύστημα που αξιολογεί την επαναχρησιμοποιησιμότητα (reusability) τμημάτων λογισμικού.



### 2.4.3 Δυνατότητες Επαναχρησιμοποίησης και Συνεισφορά

Οι δυνατότητες της χρήσης τεχνικών εξόρυξης δεδομένων για την υποστήριξη της επαναχρησιμοποίησης και η σχετική συνεισφορά της παρούσας διατριβής απεικονίζονται στο Σχήμα 2.12. Στο άνω μέρος αυτού του Σχήματος μπορεί κανείς να δει όλες τις πληροφορίες που είναι διαθέσιμες στους προγραμματιστές, οι οποίες μπορεί να προέρχονται από διάφορες πηγές, είτε online είτε τοπικές. Τα δεδομένα αυτά έχουν περιγραφεί στην ενότητα 2.2.2 και περιλαμβάνουν πηγαίο κώδικα (source code), τεκμηρίωση (documentation), απαιτήσεις λογισμικού (software requirements), δεδομένα από συστήματα ελέγχου έκδοσης (version control systems) ή συστήματα διαχείρισης σφαλμάτων (bug tracking systems), κ.λπ. Παρόλο που οι παραπάνω πηγές είναι χρήσιμες, οι πληροφορίες που παρέχουν είναι μη επεξεργασμένες (raw), συνεπώς ενδέχεται να μην είναι απευθείας κατάλληλες για την εφαρμογή τεχνικών εξόρυξης με σκοπό την επαναχρησιμοποίηση λογισμικού. Ως εκ τούτου, μπορούμε να ορίσουμε το λεγόμενο Επίπεδο Εξαγωγής Γνώσης (Knowledge Extraction Layer), το οποίο βρίσκεται στη μέση του Σχήματος 2.12. Το επίπεδο αυτό περιλαμβάνει συστήματα που λαμβάνουν τις πρωτογενείς πληροφορίες και τις επεξεργάζονται για να παρέχουν δεδομένα που είναι κατάλληλα για εξόρυξη.



Σχήμα 2.12: Επισκόπηση Εξόρυξης Δεδομένων για Επαναχρησιμοποίηση Λογισμικού και Σχετική Συνεισφορά της Διατριβής

Όσον αφορά τις απαιτήσεις λογισμικού, η πρόκληση είναι συνήθως η αποτελεσματική μοντελοποίησή τους. Με δεδομένο ένα σύνολο απαιτήσεων λογισμικού που μπορεί να εκφράζονται σε διαφορετικές μορφές (π.χ. ιστορίες χρηστών, διαγράμματα UML κ.α.), το πρώτο βήμα είναι η εξαγωγή των βασικών στοιχείων τους (π.χ. εξαγωγή των χρηστών, των ενεργειών, κ.λπ.) και η αποθήκευσή τους σε ένα μοντέλο που θα είναι κατάλληλο για επαναχρησιμοποίηση. Αυτό αποτελεί και το αντικείμενο του Κεφαλαίου 3 της παρούσας διατριβής. Έχοντας ένα κατάλληλο μοντέλο, μπορούμε στη συνέχεια να χρησιμοποιήσουμε

τεχνικές εξόρυξης δεδομένων για να διευκολύνουμε την επαναχρησιμοποίηση στον άξονα των απαιτήσεων. Παρέχουμε δύο τέτοια παραδείγματα εξόρυξης με σκοπό την επαναχρησιμοποίηση (reuse-oriented mining), ένα για λειτουργικές απαιτήσεις και ένα για μοντέλα UML (Κεφάλαιο 4).

Η επόμενη περιοχή ενδιαφέροντος είναι αυτή της επαναχρησιμοποίησης σε επίπεδο πηγαίου κώδικα. Αυτή η περιοχή ουσιαστικά αποτελεί λογική συνέχεια της προηγούμενης, καθώς οι απαιτήσεις λογισμικού καθορίζουν πρακτικά τη λειτουργικότητα του συστήματος και η συγγραφή του πηγαίου κώδικα χρησιμοποιείται για να την καλύψει. Αυτή η σχέση είναι επίσης αμφίδρομη, καθώς το λογισμικό θα πρέπει φυσικά να καλύπτει τις καθορισμένες απαιτήσεις, ενώ αντίστοιχα οι απαιτήσεις θα πρέπει να είναι ενημερωμένες και συνεπώς να συνδέονται με το λογισμικό που αναπτύσσεται. Για μια συνολική διερεύνηση αυτής της σύνδεσης, μπορούμε να κατευθύνουμε τον αναγνώστη στο ευρύ πεδίο της Αυτοματοποιημένης Μηχανικής Λογισμικού (Automated Software Engineering), που διερευνά τη σχέση μεταξύ των απαιτήσεων λογισμικού/προδιαγραφών και της ανάπτυξης του πηγαίου κώδικα της εφαρμογής, ενώ ταυτόχρονα προσπαθεί να την αυτοματοποιήσει ως διαδικασία. Στο πλαίσιο αυτής της διατριβής, η σύνδεση αυτή αναλύεται στο Κεφάλαιο 3, όπου παρουσιάζουμε ένα σενάριο ανάπτυξης μιας υπηρεσίας διαδικτύου (web service) και υποδεικνύουμε τον τρόπο με τον οποίο οι απαιτήσεις και ο πηγαίος κώδικας μπορούν να συνδεθούν αμφίδρομα και ανιχνεύσιμα (traceability).

Η περιοχή της εξόρυξης πηγαίου κώδικα είναι αρκετά ευρεία και περιλαμβάνει πλήθος εργαλείων για την εξαγωγή γνώσης από πηγαίο κώδικα, τεκμηρίωση κ.λπ. Οι πρωτογενείς πληροφορίες στην περίπτωση αυτή βρίσκονται σε αποθετήρια κώδικα (code repositories), συστήματα ελέγχου έκδοσης (version control systems) κ.λπ., ενώ τα σχετικά εργαλεία μπορεί να είναι CSEs ή άλλα συστήματα ευρετηριοποίησης (indexing systems), εργαλεία ανάλυσης για την εξαγωγή πληροφοριών από τη σύνταξη πηγαίου κώδικα (π.χ. ακολουθίες κλήσεων API), κ.λπ. Η συνεισφορά μας σε αυτό τον άξονα αναλύεται στο Κεφάλαιο 5, όπου σχεδιάζουμε και αναπτύσσουμε μια CSE που εξάγει συντακτικές πληροφορίες από τον πηγαίο κώδικα και διευκολύνει την επαναχρησιμοποίηση σε διαφορετικά επίπεδα (επίπεδο τμήματος κώδικα, επίπεδο snippet και επίπεδο έργου/πολλαπλών τμημάτων). Δεδομένου του εργαλείου που προτείνουμε, καθώς και παρόμοιων εργαλείων που είναι διαθέσιμα στο διαδίκτυο (π.χ. άλλες CSEs, μηχανισμούς ερωταπαντήσεων, κ.α.), είμαστε σε θέση να προσφέρουμε χρήσιμες λύσεις σε διάφορες πτυχές της επαναχρησιμοποίησης πηγαίου κώδικα. Ως εκ τούτου, κατασκευάζουμε τρία συστήματα, ένα για επαναχρησιμοποίηση οδηγούμενη από ελέγχους σε επίπεδο τμημάτων κώδικα (Κεφάλαιο 6), ένα για την εξαγωγή παραδειγμάτων χρήσης API για διαφορετικά σενάρια (Κεφάλαιο 7) και ένα για την περαιτέρω βελτίωση των υπαρχόντων snippets χρησιμοποιώντας δεδομένα από υπηρεσίες ερωταπαντήσεων (Κεφάλαιο 8).

Μέχρι αυτό το σημείο αναλύσαμε κυρίως τις λειτουργικές πτυχές της ανάπτυξης λογισμικού. Για να διασφαλιστεί, ωστόσο, ότι η επαναχρησιμοποίηση ενός τμήματος δεν απειλεί τη συνολική ποιότητα του υπό ανάπτυξη έργου, πρέπει να υπάρχουν και οι κατάλληλες ενδείξεις ότι το τμήμα είναι επαρκούς ποιότητας. Αναδεικνύουμε αυτή την αναγκαιότητα κατασκευάζοντας ένα σύστημα προτάσεων για τμήματα λογισμικού, τα οποία αξιολογούνται τόσο από λειτουργική σκοπιά όσο και από σκοπιά ποιότητας (Κεφάλαιο 9). Το επίπεδο εξαγωγής γνώσης σε αυτή την περίπτωση περιλαμβάνει εργαλεία για την εξαγωγή μετρικών από πηγαίο κώδικα, τεκμηρίωση, συστήματα ελέγχου έκδοσης κ.λπ. Εστιάζουμε στις μετρικές

στατικής ανάλυσης και προτείνουμε μια μεθοδολογία αξιολόγησης της επαναχρησιμοποιησιμότητας, που κατασκευάζεται χρησιμοποιώντας ως ground truth τη δημοτικότητα και την προτίμηση τμημάτων λογισμικού όπως αυτές εξάγονται από online υπηρεσίες (Κεφάλαιο 10).



**Μέρος**

**III**

**ΕΞΟΥΧΗ ΑΠΑΙΤΗΣΕΩΝ**



# 3

## Μοντελοποίηση Απαιτήσεων Λογισμικού

### 3.1 Επισκόπηση

Κατά τα αρχικά στάδια του κύκλου ζωής ανάπτυξης λογισμικού, οι προγραμματιστές και οι πελάτες συζητούν και συμφωνούν ως προς τη λειτουργικότητα του συστήματος που θα αναπτυχθεί. Η λειτουργικότητα αυτή καταγράφεται ως ένα σύνολο λειτουργικών απαιτήσεων (functional requirements), που αποτελούν τη βάση για το σχέδιο υλοποίησης (implementation plan) και τις εκτιμήσεις κόστους (cost estimations) του έργου [114]. Οι λειτουργικές απαιτήσεις μπορούν να εκφραστούν με διάφορους τρόπους, όπως π.χ. με διαγράμματα UML, με γραφικά σενάρια (storyboards) και συχνά με κείμενο σε φυσική γλώσσα [115].

Η εξαγωγή προδιαγραφών (specification extraction) από λειτουργικές απαιτήσεις είναι ένα από τα πιο σημαντικά βήματα της διαδικασίας ανάπτυξης λογισμικού. Ο πηγαίος κώδικας ενός έργου εξαρτάται συνήθως από ένα αρχικό μοντέλο (π.χ. από ένα διάγραμμα κλάσης) που πρέπει να σχεδιαστεί πολύ προσεκτικά ώστε να είναι λειτουργικά πλήρες. Ο σχεδιασμός ενός τέτοιου μοντέλου από το μηδέν δεν είναι μια απλή διαδικασία. Οι απαιτήσεις που εκφράζονται σε φυσική γλώσσα έχουν το πλεονέκτημα ότι είναι κατανοητές τόσο για τους πελάτες όσο και για τους προγραμματιστές. Ωστόσο, είναι εξίσου πιθανό να είναι διφορούμενες, ελλιπείς και ασυνεπείς. Για την αντιμετώπιση αυτών των προβλημάτων, έχουν προταθεί διάφορες εξειδικευμένες γλώσσες (formalized/specialized languages), ωστόσο οι πελάτες σπάνια διαθέτουν τις απαραίτητες τεχνικές ικανότητες για να κατασκευάσουν ή να κατανοήσουν απαιτήσεις σε κάποια εξειδικευμένη γλώσσα. Ως εκ τούτου, η προσπάθεια αυτοματοποίησης της διαδικασίας εξαγωγής προδιαγραφών από απαιτήσεις μπορεί να αποδειχθεί εξαιρετικά συμφέρουσα από πλευράς κόστους και αποτελεσματικότητας. Η αυτοματοποίηση αυτή καταργεί την ανάγκη κατανόησης πολύπλοκων μοντέλων σχεδίασης από τους πελάτες.

Επιπλέον, αυτοματοποιώντας αυτή τη διαδικασία γίνεται εφικτή η ανίχνευση ασαφειών και σφαλμάτων σε πρώιμα στάδια της διαδικασίας ανάπτυξης (π.χ. μέσω εργαλείων επαλήθευσης ή μέσω λογικού συμπερασμού - logical inference), αποφεύγοντας έτσι το σαφώς μεγαλύτερο κόστος εύρεσης και επίλυσης προβλημάτων σε μεταγενέστερα στάδια [116]. Παρόλο που υπάρχουν αρκετές τεχνικές για την εξαγωγή προδιαγραφών από απαιτήσεις,

οι τεχνικές αυτές βασίζονται κατά κύριο λόγο σε ευριστικούς κανόνες που αφορούν συγκεκριμένους τομείς (domain-specific heuristics) και/ή εξειδικευμένες γλώσσες. Στο παρόν κεφάλαιο, προτείνουμε μια μεθοδολογία που επιτρέπει στους προγραμματιστές να σχεδιάσουν το σύστημά τους χρησιμοποιώντας απαιτήσεις λογισμικού από διαφορετικές πηγές (multimodal requirements). Συγκεκριμένα, το σύστημά μας μοντελοποιεί τη στατική όψη και τη δυναμική όψη ενός έργου λογισμικού και χρησιμοποιεί τεχνικές επεξεργασίας φυσικής γλώσσας (Natural Language Processing - NLP) και σημασιολογικές τεχνικές (semantics) για τη μετατροπή των απαιτήσεων των χρηστών σε προδιαγραφές συστήματος. Επιπλέον, έχουμε κατασκευάσει ένα σύνολο από εργαλεία για τη δημιουργία μοντέλων από λειτουργικές απαιτήσεις, διαγράμματα UML και γραφικά σενάρια (storyboards) και την αποθήκευσή τους σε οντολογίες λογισμικού (software ontologies).

Αναφέρουμε, τέλος, ότι η μεθοδολογία που προτείνεται σε αυτό το κεφάλαιο μπορεί να εφαρμοστεί σε διαφορετικά μοντέλα ανάπτυξης λογισμικού και διαφορετικά σενάρια. Ενδεικτικά, παρουσιάζουμε μια εφαρμογή της μεθοδολογίας μας στην ανάπτυξη διαδικτυακών RESTful υπηρεσιών (RESTful web services). Στο συγκεκριμένο σενάριο δείχνουμε πώς η κατασκευή πρωτοτύπων υπηρεσιών μπορεί να διευκολυνθεί σημαντικά με την παροχή πλήρως ανιχνεύσιμων (traceable) προδιαγραφών.

## 3.2 Βιβλιογραφία για Καθορισμό Απαιτήσεων και Εξαγωγή Προδιαγραφών

Η μετάφραση των απαιτήσεων των χρηστών σε προδιαγραφές περιλαμβάνει ουσιαστικά τη σχεδίαση των κατάλληλων μοντέλων που θα έχουν ως είσοδο τη λειτουργικότητα που περιγράφεται από τις απαιτήσεις. Όσον αφορά τον τύπο της εισόδου, οι περισσότερες προσεγγίσεις προτείνουν εξειδικευμένες γλώσσες που μπορούν εύκολα να μεταφραστούν σε μοντέλα. Τα εργαλεία του έργου *ReDSeeDS* [117, 118] χρησιμοποιούν μια γλώσσα που ονομάζεται Requirements Specification Language (RSL) προκειμένου να εξαχθούν προδιαγραφές από σενάρια χρήσης (use cases). Το *Cucumber* [119] και το *JBehave* [120] είναι δύο άλλα δημοφιλή εργαλεία που υποστηρίζουν την *Ανάπτυξη Λογισμικού Οδηγούμενη από Συμπεριφορά* (Behavior-Driven Development - BDD). Τα εργαλεία αυτά επιτρέπουν τον ορισμό σεναρίων του τύπου given-then-when και επιδιώκουν την κατασκευή μοντέλων συμπεριφοράς (behavioral models) τα οποία στη συνέχεια μπορούν να περάσουν και από ελέγχους. Μια άλλη δημοφιλής προσέγγιση που χρησιμοποιεί εξειδικευμένη γλώσσα είναι το *Tropos* [121], μια μεθοδολογία που βασίζεται στις έννοιες των χρηστών (actors) και των στόχων (goals) του μηχανισμού μοντελοποίησης  $i^*$  [122].

Παρόλο που οι προαναφερθείσες προσεγγίσεις μπορούν να είναι πολύ χρήσιμες για την κατασκευή μοντέλων υψηλής ακρίβειας, η συμβατότητά τους με την UML και/ή με άλλα αντίστοιχα πρότυπα είναι συχνά περιορισμένη. Η χρήση τους απαιτεί την εκπαίδευση σε μια νέα γλώσσα, που μπορεί να αποτελέσει αποτρεπτικό παράγοντα για τους προγραμματιστές. Για το λόγο αυτό, η τρέχουσα βιβλιογραφία προτείνει τη εφαρμογή τεχνικών σημασιολογίας και τεχνικών NLP σε λειτουργικές απαιτήσεις που εκφράζονται σε φυσική (ή ημι-δομημένη - semi-structured) γλώσσα και σε διαγράμματα UML για την εξαγωγή προδιαγραφών. Μία από τις πρώτες μεθόδους που βασίζονται σε κανόνες (rule-based) για την εξαγωγή τύπων δεδομένων (data types), μεταβλητών (variables) και χειριστών (operators) από τις απαιτήσεις



προτάθηκε από την Abbott [123]. Σύμφωνα με τη μεθοδολογία του Abbott, τα ουσιαστικά προσδιορίζονται ως αντικείμενα (objects) και τα ρήματα ως αλληλεπιδράσεις (operations) μεταξύ τους. Μεταγενέστερα, η προσέγγιση αυτή επεκτάθηκε και στην αντικειμενοστραφή ανάπτυξη λογισμικού από τον Booch [124].

Ο Saeki και οι συνεργάτες τους [125] ήταν από τους πρώτους που δημιούργησαν ένα σύστημα για την εξαγωγή αντικειμενοστραφών μοντέλων από λειτουργικές απαιτήσεις. Το σύστημά τους χρησιμοποιεί μεθόδους NLP για να εξάγει ουσιαστικά και ρήματα από τις απαιτήσεις, τα οποία στη συνέχεια προσδιορίζεται αν είναι σχετικά με το μοντέλο μέσω ανθρώπινης παρέμβασης. Το έργο του Mich [126] περιλαμβάνει επιπλέον ένα σημασιολογικό μοντέλο που χρησιμοποιεί μια βάση γνώσεων (knowledge base) για να αξιολογήσει περαιτέρω τους συντακτικούς όρους που αναγνωρίστηκαν. Μια εναλλακτική πρόταση από τους Harmain και Gaizauskas [127] είναι το *CM-Builder*, ένα εργαλείο που έχει σκοπό να εξάγει κλάσεις αντικειμένων (object classes), ιδιότητες (attributes) και σχέσεις (relationships) από λειτουργικές απαιτήσεις.

Όσον αφορά τις αναπαραστάσεις, τα περισσότερα από τα προαναφερθέντα συστήματα χρησιμοποιούν είτε γνωστά πρότυπα της βιβλιογραφίας, όπως μοντέλα UML, είτε οντολογίες, οι οποίες χρησιμοποιούνται συχνά στη *Μηχανική Απαιτήσεων (Requirements Engineering - RE)* για την αποθήκευση και επικύρωση των πληροφοριών που εξάγονται από τις απαιτήσεις [128, 129]. Οι οντολογίες είναι επίσης χρήσιμες για την αποθήκευση γνώσης για συγκεκριμένους τομείς (domain-specific knowledge) ενώ είναι γνωστό ότι αποτελούν αποτελεσματικό τρόπο αποθήκευσης στοιχείων που προκύπτουν από γλώσσες μοντελοποίησης λογισμικού (software modeling languages) [130]<sup>1</sup>. Επιπλέον, οι οντολογίες παρέχουν έναν δομημένο τρόπο για την οργάνωση της πληροφορίας, ενώ είναι ιδιαίτερα χρήσιμες για την αποθήκευση συνδεδεμένων δεδομένων (linked data), επιτρέποντας την ανάκτηση αποθηκευμένων δεδομένων μέσω ερωτημάτων. Τέλος, υποστηρίζουν την εφαρμογή συλλογιστικών κανόνων (reasoning) για τη μοντελοποίηση υπονοούμενων σχέσεων μεταξύ των τιμών δεδομένων.

Παρόλο που οι παραπάνω μεθοδολογίες είναι αποτελεσματικές, συνήθως περιορίζονται σε εξειδικευμένες γλώσσες ή εξαρτώνται από ευριστικούς κανόνες (heuristics) και από πληροφορίες που αφορούν συγκεκριμένους τομείς (domain-specific). Για το λόγο αυτό, προτείνουμε τη χρήση σημασιολογικών τεχνικών ανάθεσης ρόλων (semantic role labeling) προκειμένου να αντιστοιχηθούν οι έννοιες των λειτουργικών απαιτήσεων σε σχετικές προδιαγραφές. Αν και η εξαγωγή σημασιολογικών σχέσεων μεταξύ οντοτήτων είναι ένα από τα πιο γνωστά προβλήματα του τομέα της επεξεργασίας φυσικής γλώσσας<sup>2</sup>, η εφαρμογή αυτών των μεθόδων για την ανάλυση των απαιτήσεων των χρηστών είναι μια σχετικά νέα ιδέα.

<sup>1</sup>Ο αναγνώστης παραπέμπεται στο [131] για μια εκτεταμένη ανασκόπηση όσον αφορά τη χρήση οντολογιών στη Μηχανική Απαιτήσεων.

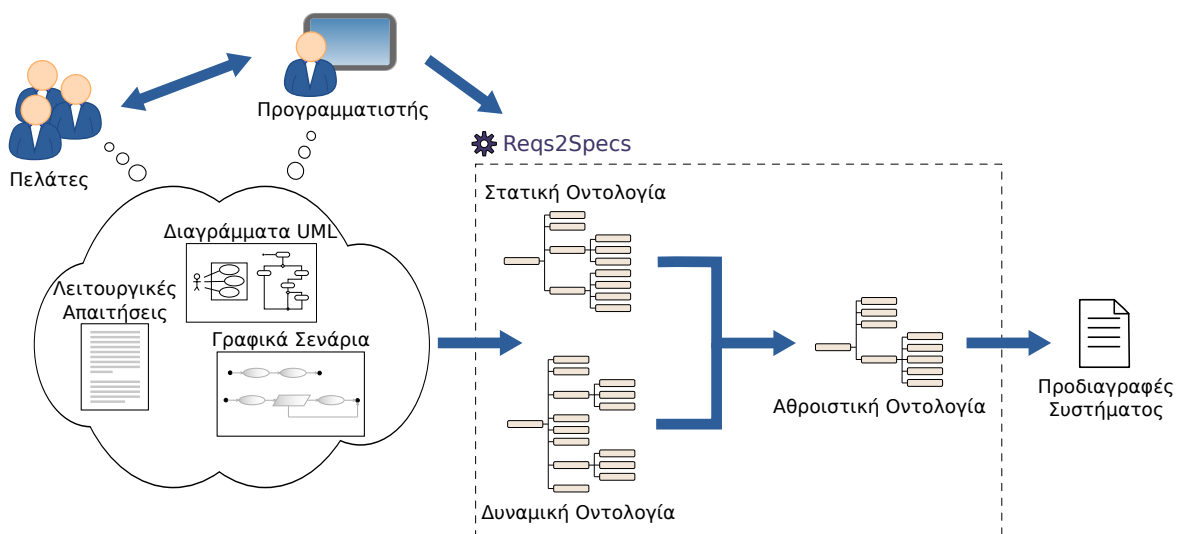
<sup>2</sup>Υπάρχουν πράγματι αρκετές προσεγγίσεις που μπορούν να διακριθούν σε μεθόδους βασισμένες σε χαρακτηριστικά (feature-based) και μεθόδους βασισμένες σε πυρήνες (kernel-based). Έχοντας ένα “σημαδεμένο” (annotated) σύνολο δεδομένων, οι μέθοδοι βασισμένες σε χαρακτηριστικά εξάγουν κάποια προκαθορισμένα συντακτικά και σημασιολογικά χαρακτηριστικά, και στη συνέχεια παρέχουν το σύνολο των χαρακτηριστικών σε έναν ταξινομητή (classifier) που εκπαιδεύεται για να προσδιορίζει τις σχέσεις [132–134]. Αντίθετα, οι μέθοδοι βασισμένες σε πυρήνες δεν απαιτούν τον προσδιορισμό του συνόλου χαρακτηριστικών καθώς αντιστοιχίζουν τις οντότητες και τις σχέσεις του κειμένου σε μια αναπαράσταση μεγαλύτερης διάστασης. Μετά την αντιστοίχιση αυτή με χρήση κάποιου πυρήνα, όπως π.χ. πυρήνα bag-of-words [135] ή πυρήνα δένδρου (tree kernel) [136–138], οι σχέσεις ταξινομούνται σύμφωνα στο χώρο που ορίζεται από τη νέα αναπαράσταση. Και οι δύο κατηγορίες μεθόδων είναι αποτελεσματικές σε διαφορετικά σενάρια [139].

Σε σχέση με την τρέχουσα βιβλιογραφία, επιδιώκουμε επίσης να παρέχουμε ένα ενιαίο μοντέλο (unified model) για την αποθήκευση απαιτήσεων από διαφορετικές πηγές (multi-modal), λαμβάνοντας υπόψη τόσο τη στατική όψη (static view) όσο και τη δυναμική όψη (dynamic view) των έργων λογισμικού. Το μοντέλο μας μπορεί να χειριστεί δεδομένα από λειτουργικές απαιτήσεις γραμμένες σε φυσική γλώσσα, από διαγράμματα σεναρίων χρήσης (use case diagrams) και διαγράμματα δραστηριοτήτων (activity diagrams) εκφρασμένα ως UML μοντέλα, καθώς και από γραφικά σενάρια (storyboards), για τα οποία έχουμε κατασκευάσει κατάλληλα εργαλεία. Οι εξαγόμενες πληροφορίες αποθηκεύονται σε οντολογίες, επιτρέποντας έτσι την ανάλυση και την επαναχρησιμοποίησή τους.

### 3.3 Εξαγωγή Προδιαγραφών από Απαιτήσεις

#### 3.3.1 Επισκόπηση Συστήματος

Το σύστημά μας περιλαμβάνει το τμήμα *Reqs2Specs* και η αρχιτεκτονική του φαίνεται στο Σχήμα 3.1. Το τμήμα αυτό περιέχει ένα σύνολο εργαλείων που οι προγραμματιστές μπορούν να χρησιμοποιήσουν για να εισάγουν τις απαιτήσεις ενός έργου λογισμικού σε διάφορες μορφές (λειτουργικές απαιτήσεις, γραφικά σενάρια), καθώς επίσης και μια μεθοδολογία για τη μετατροπή αυτών των αναπαραστάσεων σε προδιαγραφές, δηλαδή σε μοντέλα για την ανάπτυξη του έργου.



Σχήμα 3.1: Επισκόπηση της Αρχιτεκτονικής του Συστήματός μας

Έχουμε αναπτύξει κατάλληλα εργαλεία που επιτρέπουν στους προγραμματιστές να εισάγουν απαιτήσεις σε μορφή ημι-δομημένου κειμένου, σε μορφή UML διαγραμμάτων σεναρίων χρήσης και διαγραμμάτων δραστηριοτήτων, και σε μορφή γραφικών σεναρίων (storyboards), δηλαδή διαγραμμάτων που απεικονίζουν τη ροή των ενεργειών (action flow) στο σύστημα. Χρησιμοποιώντας τη μεθοδολογία μας, αυτές οι αναπαραστάσεις σχολιάζονται συντακτικά και σημασιολογικά και αποθηκεύονται σε δύο οντολογίες, στη στατική οντολογία και στη δυναμική οντολογία, που πρακτικά αποθηκεύουν τα στατικά στοιχεία και τα δυναμικά στοιχεία του συστήματος.

Έχοντας ολοκληρώσει τη διαδικασία καθορισμού των απαιτήσεων, το σύστημα συναθροίζει τις οντολογίες σε μια αθροιστική οντολογία και εξάγει λεπτομερείς προδιαγραφές συστήματος, οι οποίες είναι κατάλληλες είτε για άμεση χρήση από τους προγραμματιστές, είτε για να δοθούν ως είσοδος σε κάποιο μηχανισμό αυτόματης παραγωγής πηγαίου κώδικα (automated source code generation engine - όπως στο S-CASE [140]). Σημειώνουμε επιπλέον ότι η μεθοδολογία μας είναι πλήρως ανιχνεύσιμη (traceable): οποιεσδήποτε μεταβολές στις οντολογίες (π.χ. ως μέρος μιας διαδικασίας εξόρυξης, βλέπε επόμενο κεφάλαιο) μπορούν να μεταφερθούν πίσω στις λειτουργικές απαιτήσεις, στα διαγράμματα UML και στα γραφικά σενάρια. Ως αποτέλεσμα, οι προγραμματιστές και οι πελάτες έχουν πάντοτε μπροστά τους μια σαφή εικόνα των απαιτήσεων του συστήματος, ενώ οι προδιαγραφές είναι επίσης σαφώς καθορισμένες. Οι ακόλουθες ενότητες περιγράφουν τις οντολογίες που αναπτύχθηκαν, καθώς και τη διαδικασία με την οποία εισάγονται πληροφορίες από τις λειτουργικές απαιτήσεις, τα διαγράμματα UML και τα γραφικά σενάρια.

### 3.3.2 Μοντελοποίηση Απαιτήσεων

Σε αυτή την ενότητα περιγράφονται οι δύο οντολογίες που μοντελοποιούν τη στατική όψη (λειτουργικές απαιτήσεις, διαγράμματα σεναρίων χρήσης) και τη δυναμική όψη (γραφικά σενάρια, διαγράμματα δραστηριοτήτων) έργων λογισμικού. Αυτές οι οντολογίες επιτρέπουν την οργάνωση της πληροφορίας, ενώ υποστηρίζουν μεθόδους για την ανάκτηση των δεδομένων μέσω ερωτημάτων.

Για την αναπαράσταση των πληροφοριών που είναι σχετικές με τις απαιτήσεις επιλέγουμε το πρότυπο RDF (Resource Description Framework)<sup>3</sup>. Το μοντέλο δεδομένων του RDF έχει τρεις τύπους αντικειμένων: *πόρους (resources)*, *ιδιότητες (properties)* και *δηλώσεις (statements)*. Ένας πόρος είναι οποιοδήποτε αντικείμενο μπορεί να περιγραφεί από τη γλώσσα, ενώ οι ιδιότητες είναι δυαδικές σχέσεις. Μια δήλωση είναι μια τριπλέτα αποτελούμενη από έναν πόρο, μια ιδιότητα, και έναν πόρο ή ένα αλφαριθμητικό ως το αντικείμενο της ιδιότητας. Καθώς το πρότυπο RDF δεν ορίζει κάποια σύνταξη για τη γλώσσα, τα μοντέλα RDF μπορούν να εκφράζονται σε διαφορετικές μορφές, εκ των οποίων οι δύο πιο διαδεδομένες είναι η μορφή XML και η μορφή Turtle<sup>4</sup>. Τα μοντέλα RDF συχνά συνοδεύονται από ένα RDF σχήμα (RDF schema - RDFS) που ορίζει ένα λεξιλόγιο πόρων και ιδιοτήτων. Παρόλο που το πρότυπο RDF είναι επαρκές για την αποθήκευση πληροφοριών, η εφαρμογή συλλογιστικών κανόνων στα δεδομένα και η χρήση σημασιολογίας δεν υποστηρίζονται με εύκολο τρόπο. Το RDFS παρέχει κάποιες πράξεις λογικής, αλλά από τη σχεδιάσή του δεν είναι ιδιαίτερα εκφραστικό και δεν υποστηρίζει τη λογική έννοια της άρνησης. Για το σκοπό αυτό, σχεδιάστηκε η γλώσσα OWL (Web Ontology Language)<sup>5</sup>, στην οποία οι κλάσεις καθορίζονται αξιωματικά, ενώ επιπλέον υποστηρίζεται η εφαρμογή συλλογιστικών κανόνων που αφορούν τη συνέπεια (consistency) μεταξύ των κλάσεων. Η OWL είναι χτισμένη πάνω στο μοντέλο RDF, αλλά περιλαμβάνει μια πιο ισχυρή σύνταξη με χαρακτηριστικά όπως η πληθικότητα (cardinality) ή οι αντίστροφες σχέσεις (inverse relations) μεταξύ των ιδιοτήτων. Στο πλαίσιο της παρούσας διατριβής, χρησιμοποιούμε τη γλώσσα OWL για την ανα-

<sup>3</sup><http://www.w3.org/TR/1999/REC-rdf-syntax-19990222>

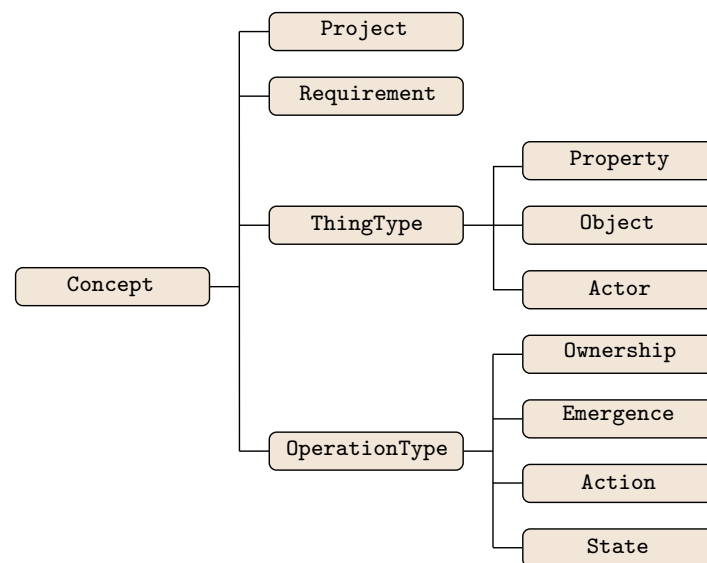
<sup>4</sup><http://www.w3.org/TR/turtle/>

<sup>5</sup><http://www.w3.org/TR/2004/REC-owl-guide-20040210/>

παράσταση της πληροφορίας, καθώς αποτελεί ένα ευρέως χρησιμοποιούμενο πρότυπο τόσο στην ερευνητική κοινότητα όσο και στη βιομηχανία<sup>6</sup>.

### 3.3.2.1 Στατική Όψη Έργων Λογισμικού

**Οντολογία για τη Στατική Όψη Έργων Λογισμικού** Όσον αφορά τους στατικούς τύπους απαιτήσεων, δηλαδή τις λειτουργικές απαιτήσεις και τα διαγράμματα σεναρίων χρήσης, η σχεδίαση της οντολογίας ακολουθεί τη λογική ότι κάποια υποκείμενα (π.χ. χρήστης - user, διαχειριστής - administrator, επισκέπτης - guest) εκτελούν ενέργειες τις οποίες τις δέχονται κάποια αντικείμενα. Η οντολογία σχεδιάστηκε για να αποθηκεύει πληροφορίες που εξάγονται από προτάσεις του τύπου Υποκείμενο-Ρήμα-Αντικείμενο (Subject-Verb-Object - SVO). Η ιεραρχία των κλάσεων της οντολογίας παρουσιάζεται στο Σχήμα 3.2.



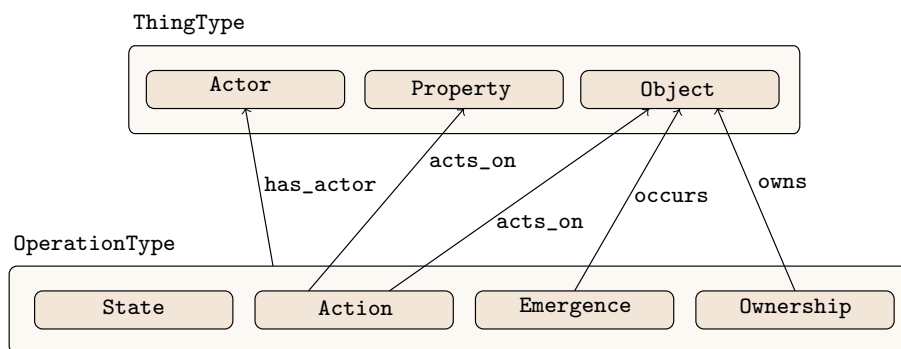
Σχήμα 3.2: Στατική Οντολογία Έργων Λογισμικού

Οτιδήποτε εισάγεται στην οντολογία είναι ένα **Concept**. Ένα αντικείμενο τύπου **Concept** μπορεί να είναι ένα έργο (**Project**), μια απαίτηση (**Requirement**), ένα πράγμα (**ThingType**) ή μια λειτουργία (**OperationType**). Οι κλάσεις **Project** και **Requirement** χρησιμοποιούνται για την αποθήκευση των απαιτήσεων του συστήματος, έτσι ώστε για κάθε αντικείμενο να γνωρίζουμε από ποια απαίτηση προήλθε. Οι κλάσεις **ThingType** και **OperationType** περιέχουν τους βασικούς τύπους αντικειμένων για οποιαδήποτε απαίτηση. Η κλάση **ThingType** αναφέρεται στα υποκείμενα/αντικείμενα του συστήματος που ενεργούν ή λαμβάνουν το αποτέλεσμα μιας ενέργειας, ενώ η κλάση **OperationType** περιλαμβάνει τους τύπους ενεργειών που εκτελούνται από τα υποκείμενα/αντικείμενα που ενεργούν. Κάθε αντικείμενο τύπου **ThingType** μπορεί να είναι ένας δράστης (**Actor**), ένα αντικείμενο (**Object**) ή μια ιδιότητα (**Property**). Η κλάση **Actor** αναφέρεται στους δράστες του συστήματος, συμπεριλαμβανομένων των χρηστών (**users**), του ίδιου του συστήματος (**system**) ή και εξωτερικών συστημάτων (**external systems**). Τα αντικείμενα του τύπου

<sup>6</sup>Παρόλο που δεν αξιοποιούμε στο έπακρο τις δυνατότητες εξαγωγής συμπερασμάτων (*inference capabilities*) της OWL, μας είναι χρήσιμες για τον έλεγχο της ακεραιότητας (*integrity*) στην οντολογία, όπως για τη διασφάλιση ότι ορισμένες ιδιότητες έχουν αντίστροφη ιδιότητα (π.χ. μια ιδιότητα κτήσης *owns/owned\_by*).

Object περιλαμβάνουν όλα τα αντικείμενα ή τους πόρους του συστήματος που λαμβάνουν κάποια ενέργεια, ενώ οι ιδιότητες (Property) περιλαμβάνουν όλους τις βασικές ιδιότητες των αντικειμένων ή των ενεργειών. Η κλάση OperationType αφορά όλες τις πιθανές λειτουργίες, που περιλαμβάνουν λειτουργίες τύπου Ownership που εκφράζουν την κατοχή (π.χ. “each user has an account”), λειτουργίες τύπου Emergence που εκφράζουν κάποιο παθητικό μετασχηματισμό (π.χ. “the posts are sorted”), λειτουργίες τύπου State που περιγράφουν την κατάσταση ενός δράστη (π.χ. “the user is logged in”) και τέλος λειτουργίες τύπου Action που περιγράφουν ενέργειες που εκτελούνται από κάποιο δράστη (π.χ. “the user creates a profile”).

Οι πιθανές αλληλεπιδράσεις μεταξύ των διαφορετικών εννοιών της οντολογίας, δηλαδή οι σχέσεις μεταξύ των κλάσεων, καθορίζονται με την χρήση ιδιοτήτων (*properties*). Οι ιδιότητες της στατικής οντολογίας παρουσιάζονται στο Σχήμα 3.3. Σημειώστε ότι για κάθε ιδιότητα ορίζεται επίσης η αντίστροφή της (π.χ. η ιδιότητα *has\_actor* έχει την αντίστροφη *is\_actor\_of*). Για λόγους απλότητας, στο Σχήμα 3.3 περιλαμβάνεται μόνο μία ιδιότητα για κάθε ζεύγος ιδιότητας και αντίστροφης, ενώ επιπλέον δεν απεικονίζονται ιδιότητες που αφορούν τις κλάσεις Project και Requirement.

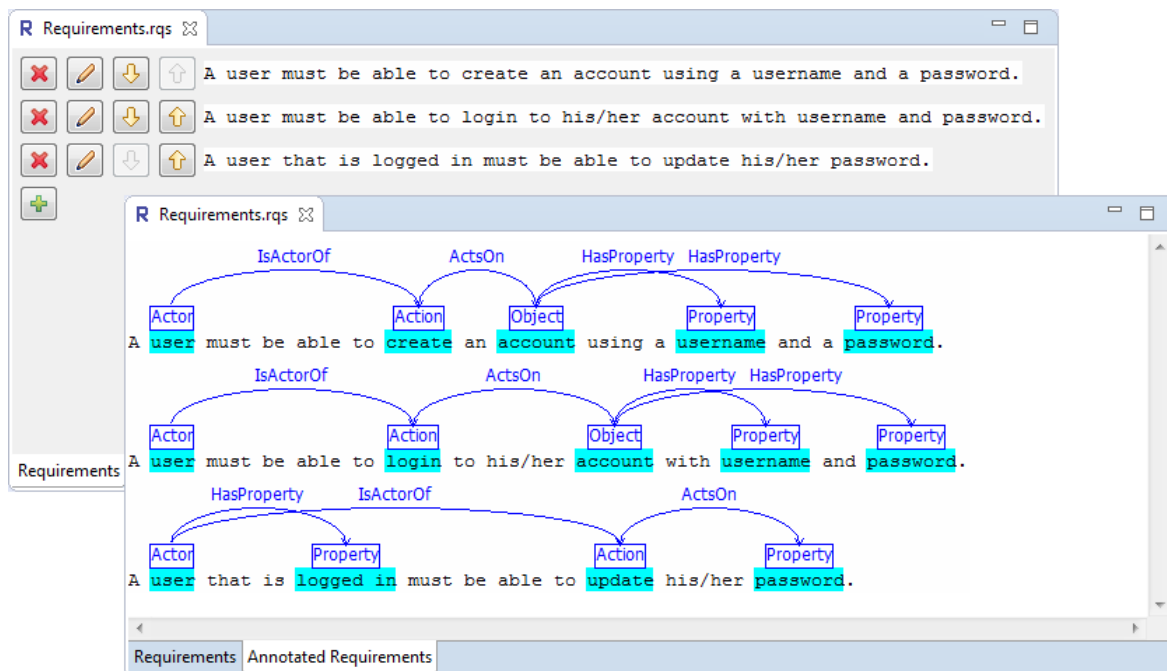


Σχήμα 3.3: Ιδιότητες Στατικής Οντολογίας

Ένα αντικείμενο τύπου Project μπορεί να έχει πολλές απαιτήσεις (αντικείμενα τύπου Requirement), ενώ κάθε απαίτηση συνδέεται με διάφορα αντικείμενα τύπου ThingType και OperationType. Οι σχέσεις μεταξύ των τελευταίων ορίζονται από τις υπόλοιπες ιδιότητες. Οι λειτουργίες (OperationType) συνδέονται με τους δράστες (Actor) μέσω της ιδιότητας *has\_actor*. Αν επιπλέον είναι μεταβατικές (transitive), συνδέονται και με τα σχετικά αντικείμενα. Έτσι, κάθε ενέργεια (Action) συνδέεται με αντικείμενα τύπου Object ή τύπου Property μέσω της ιδιότητας *acts\_on*, ενώ οι μετασχηματισμοί ουσιαστικά συμβαίνουν πάνω στα αντικείμενα, οπότε οι κλάσεις Emergence και Object συνδέονται με την ιδιότητα *occurs*. Η κατοχή ενός αντικειμένου (Ownership) δηλώνεται με την ιδιότητα *owns* ενώ για την αντίστροφη σχέση το αντίστοιχο αντικείμενο συνδέεται με την κλάση Ownership μέσω της ιδιότητας *owned\_by*. Τέλος, η περιγραφή μιας κατάστασης (State) που δεν είναι μεταβατική λειτουργία δεν συνδέεται με κανένα αντικείμενο τύπου Object ή Property.

**Εξαγωγή Αντικειμένων από Λειτουργικές Απαιτήσεις** Στην οντολογία που περιγράφηκε στις προηγούμενες παραγράφους αποθηκεύονται δεδομένα από λειτουργικές απαιτήσεις. Οι απαιτήσεις πρέπει να είναι “σημαδεμένες”/σχολιασμένες (*annotated*), ώστε στη συ-

νέχεια οι διάφορες οντότητες να αντιστοιχιστούν στις OWL κλάσεις της οντολογίας. Έτσι σχεδιάσαμε μια μεθοδολογία σχολιασμού των απαιτήσεων που περιλαμβάνει τέσσερις τύπους οντοτήτων και τρεις τύπους σχέσεων μεταξύ τους. Ειδικότερα, μια οντότητα από μια λειτουργική απαίτηση μπορεί να είναι ένας δράστης (*Actor*), μια ενέργεια (*Action*), ένα αντικείμενο (*Object*) ή μια ιδιότητα (*Property*). Αντίστοιχα, ορίζονται και οι σχέσεις *IsActorOf* (από έναν *Actor* σε ένα *Action*), *ActsOn* (από ένα *Action* σε ένα *Object* ή από ένα *Action* σε ένα *Property*) και *HasProperty* (από έναν *Actor* σε ένα *Property* ή από ένα *Object* σε ένα *Property* ή από ένα *Property* σε ένα *Property*). Με βάση τους παραπάνω κανόνες σχολιασμού, αναπτύξαμε ένα εργαλείο για την προσθήκη, τροποποίηση και σχολιασμό λειτουργικών απαιτήσεων. Το εργαλείο *Requirements Editor* είναι ένα SWT plugin του ενσωματωμένου περιβάλλοντος ανάπτυξης Eclipse και φαίνεται στο Σχήμα 3.4.



Σχήμα 3.4: Screenshot του Requirements Editor

Στην πρώτη οθόνη του εργαλείου (που φαίνεται πάνω αριστερά στο Σχήμα 3.4), ο χρήστης μπορεί να προσθέσει, να διαγράψει ή να τροποποιήσει λειτουργικές απαιτήσεις. Η δεύτερη οθόνη του εργαλείου (που φαίνεται κάτω δεξιά στο Σχήμα 3.4) αφορά το σχολιασμό απαιτήσεων. Σε αυτή την οθόνη ο χρήστης μπορεί να προσθέσει και να διαγράψει σχόλια (annotations) για τις οντότητες και τις σχέσεις.

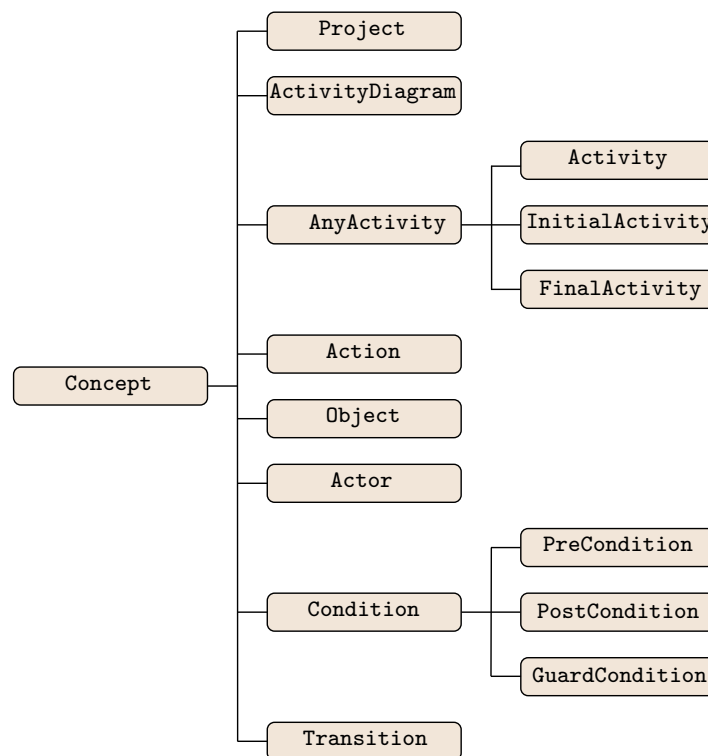
Δεδομένου ότι η δομή των λειτουργικών απαιτήσεων ακολουθεί συνήθως το μοτίβο SVO (όπως στο Σχήμα 3.4), ο σχολιασμός τους είναι σχετικά εύκολος. Ωστόσο, η διαδικασία σχολιασμού πολλών απαιτήσεων μπορεί να είναι κουραστική για τον χρήστη. Για το σκοπό αυτό, το εργαλείο μπορεί να συνδεθεί με έναν εξωτερικό αναλυτή φυσικής γλώσσας (NLP parser) για να υποστηρίξει τον ημι-αυτόματο σχολιασμό απαιτήσεων. Ως ένα παράδειγμα, δημιουργήσαμε έναν τέτοιο αναλυτή (βλέπε ενότητα 3.3.3) που λειτουργεί σε δύο στάδια. Το πρώτο στάδιο είναι η *συντακτική ανάλυση (syntactic analysis)* κάθε απαίτησης. Οι προτάσεις που δίνονται ως είσοδος αρχικά χωρίζονται σε λέξεις (tokens), στη συνέχεια προσδιορίζεται η γραμματική κατηγορία και εξάγεται ο βασικός τύπος (base type) κάθε λέ-

ξης, για να προσδιοριστούν τελικά οι γραμματικές σχέσεις μεταξύ των λέξεων. Το δεύτερο στάδιο περιλαμβάνει τη *σημασιολογική ανάλυση (semantic analysis)* των αναλυμένων προτάσεων. Κατά τη σημασιολογική ανάλυση εξάγονται τα σημασιολογικά χαρακτηριστικά των όρων (π.χ. μέρος του λόγου - part-of-speech) και χρησιμοποιείται ένας αλγόριθμος ταξινόμησης για την ανάθεση κάθε όρου στο σχετικό αντικείμενο ή σχέση της οντολογίας.

Μετά τον σχολιασμό, ο χρήστης μπορεί να επιλέξει την αποθήκευση των δεδομένων στη στατική οντολογία. Η αντιστοίχιση από τους κανόνες σχολιασμού στις έννοιες της στατικής οντολογίας είναι αρκετά απλή. Οι οντότητες *Actor*, *Action*, *Object* και *Property* αντιστοιχίζονται στις ομώνυμες OWL κλάσεις. Όσον αφορά τις σχέσεις, το *IsActorOf* αντιστοιχίζεται στις ιδιότητες *is\_actor\_of* και *has\_actor*, το *ActsOn* αντιστοιχίζεται στις ιδιότητες *acts\_on* και *receives\_action* και τέλος το *HasProperty* αντιστοιχίζεται στις ιδιότητες *has\_property* και *is\_property\_of*. Οι υπόλοιπες ιδιότητες της οντολογίας (π.χ. η *project\_has* ή η *consists\_of*) αποθηκεύουν πληροφορίες σχετικές τις λειτουργικές απαιτήσεις και το όνομα του έργου λογισμικού.

### 3.3.2.2 Δυναμική Όψη Έργων Λογισμικού

**Οντολογία για τη Δυναμική Όψη Έργων Λογισμικού** Στην υποενότητα αυτή, παρουσιάζεται η οντολογία που καταγράφει τη δυναμική όψη ενός συστήματος. Τα βασικά στοιχεία των δυναμικών απαιτήσεων είναι οι ροές των ενεργειών (action flows) μεταξύ των αντικειμένων του συστήματος. Χρησιμοποιώντας τη γλώσσα OWL, οι ενέργειες μπορούν να αναπαρασταθούν ως κλάσεις και οι ροές μπορούν να περιγραφούν χρησιμοποιώντας ιδιότητες. Η ιεραρχία των κλάσεων της οντολογίας παρουσιάζεται στο Σχήμα 3.5.



Σχήμα 3.5: Δυναμική Οντολογία Έργων Λογισμικού



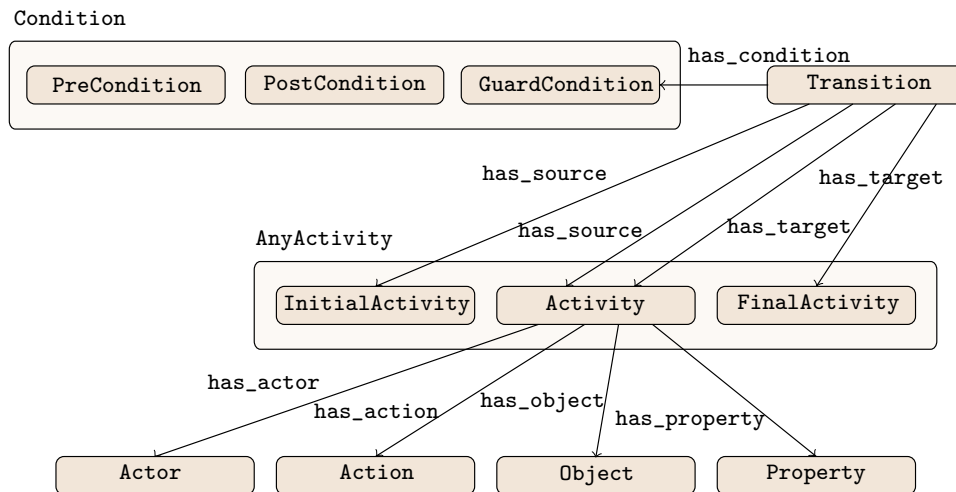
Οτιδήποτε εισάγεται στην οντολογία είναι ένα **Concept**. Ένα αντικείμενο τύπου **Concept** μπορεί να είναι ένα έργο (**Project**), ένα διάγραμμα (**ActivityDiagram**), μια δραστηριότητα (**AnyActivity**), ένας δράστης (**Actor**), μια ενέργεια (**Action**), ένα αντικείμενο (**Object**), μια συνθήκη (**Condition**), μια μετάβαση (**Transition**) ή μια ιδιότητα (**Property**). Η κλάση **Project** αναφέρεται στο έργο που αναλύεται, ενώ το **ActivityDiagram** αποθηκεύει κάθε διάγραμμα του συστήματος, συμπεριλαμβανομένων όχι μόνο διαγραμμάτων δραστηριοτήτων (**activity diagrams**), αλλά και γραφικών σεναρίων (**storyboards**) και γενικά διαγραμμάτων με δυναμικές ροές ενεργειών. Όπως και στην στατική οντολογία, οι κλάσεις **Project** και **ActivityDiagram** μπορούν να χρησιμοποιηθούν για να διασφαλίσουν ότι οι έννοιες που εξάγονται από την οντολογία συνδέονται με τα αντικείμενα των αρχικών διαγραμμάτων, επιτρέποντας έτσι την αναδόμηση τους.

Οι δραστηριότητες αποτελούν τις βασικές δομικές μονάδες των δυναμικών αναπαραστάσεων του συστήματος. Οι δραστηριότητες ενός διαγράμματος αποθηκεύονται στην OWL κλάση **AnyActivity**. Οι δραστηριότητες αυτής της κλάσης χωρίζονται περαιτέρω σε αρχικές δραστηριότητες (**InitialActivity**), τελικές δραστηριότητες (**FinalActivity**), καθώς και απλές δραστηριότητες (**Activity**). Οι αρχικές και οι τελικές δραστηριότητες αναφέρονται στην αρχική και στην τελική κατάσταση του συστήματος, αντίστοιχα, ενώ οι δραστηριότητες της κλάσης **Activity** είναι όλες οι ενδιάμεσες δραστηριότητες του συστήματος. Κάθε δραστηριότητα του συστήματος μπορεί επίσης να απαιτεί μία ή περισσότερες ιδιότητες εισόδου, που αποθηκεύονται στην κλάση **Property**. Για παράδειγμα, για την εκτέλεση της δραστηριότητας “**Create account**” μπορεί να απαιτούνται ένα “**username**” και ένα “**password**”. Σε αυτήν την περίπτωση, τα “**username**” και “**password**” είναι αντικείμενα της κλάσης **Property**.

Η ροή των δραστηριοτήτων σε γραφικά σενάρια ή διαγράμματα δραστηριοτήτων περιγράφεται χρησιμοποιώντας μεταβάσεις. Η κλάση **Transition** περιγράφει τη μετάβαση από μια δραστηριότητα (**Activity**) σε μία άλλη όπως αυτή προκύπτει από το αντίστοιχο διάγραμμα. Κάθε μετάβαση (**Transition**) μπορεί επίσης να έχει μια συνθήκη (**Condition**). Η κλάση **Condition** έχει τις υποκλάσεις **PreCondition**, **PostCondition** και **GuardCondition**. Οι πρώτες δύο αναφέρονται σε συνθήκες που πρέπει να πληρούνται πριν (**PreCondition**) ή μετά (**PostCondition**) την εκτέλεση της ροής δραστηριοτήτων του διαγράμματος, ενώ αντικείμενο **GuardCondition** είναι μια συνθήκη που επιτρέπει ή όχι την εκτέλεση μιας δραστηριότητας του συστήματος και εκφράζεται μαζί με την αντίστοιχη απάντηση. Για παράδειγμα, η δραστηριότητα “**Create account**” μπορεί να επιτρέπεται από τη συνθήκη “**is the username unique? Yes**”, ενώ η αντίθετη συνθήκη “**is the username unique? No**” να μην επιτρέπει την εκτέλεση της δραστηριότητας “**Create account**”.

Οι ιδιότητες της οντολογίας καθορίζουν τις αλληλεπιδράσεις μεταξύ των διαφόρων κλάσεων, που αφορούν αλληλεπιδράσεις σε επίπεδο διαγραμμάτων και σχέσεις μεταξύ των στοιχείων ενός διαγράμματος. Οι ιδιότητες της δυναμικής οντολογίας απεικονίζονται στο Σχήμα 3.6, όπου, για λόγους απλότητας, δεν περιλαμβάνονται οι αντίστροφες σχέσεις, ενώ επιπλέον δεν απεικονίζονται και οι ιδιότητες που αφορούν τις κλάσεις **Project** και **ActivityDiagram**. Κάθε έργο μπορεί να έχει ένα ή περισσότερα διαγράμματα και κάθε διάγραμμα πρέπει να ανήκει σε ένα έργο. Επιπλέον, κάθε διάγραμμα μπορεί να έχει μια αρχική συνθήκη (**PreCondition**) και/ή μια τελική συνθήκη (**PostCondition**). Ένα αντικείμενο τύπου **ActivityDiagram** έχει αντικείμενα των κλάσεων **Actor**, **AnyActivity**, **Transition**, **Property** και **Condition**.





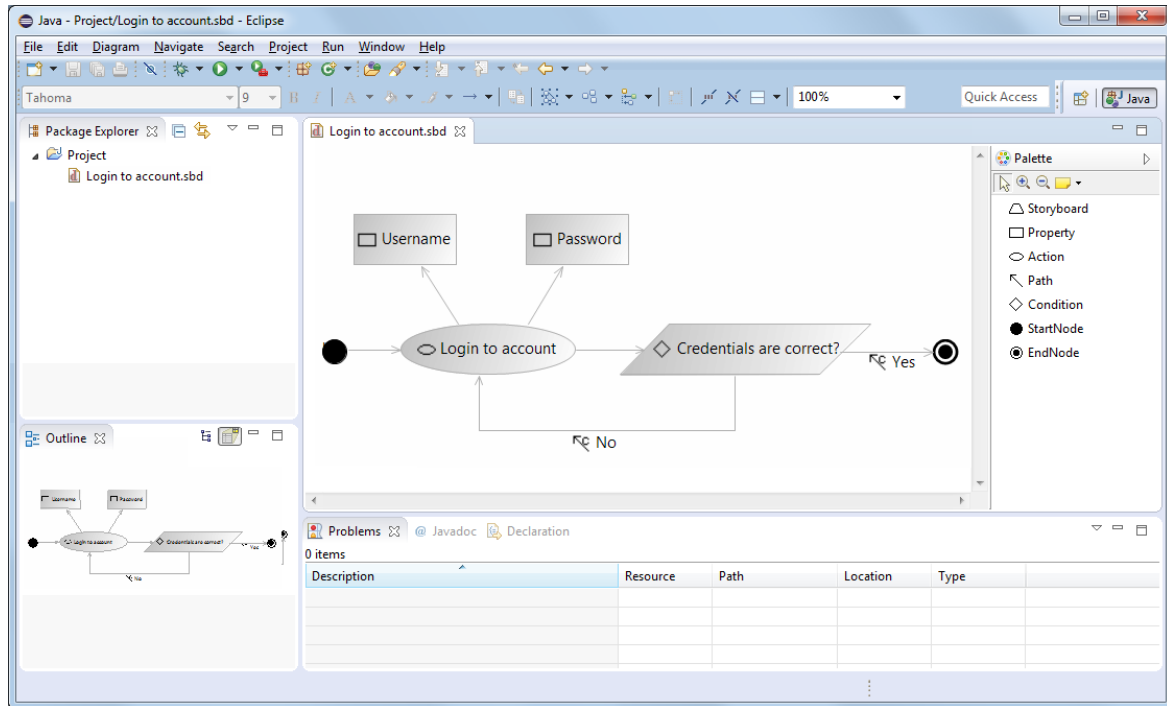
Σχήμα 3.6: Ιδιότητες Δυναμικής Οντολογίας

Οι σχέσεις των κλάσεων αποτελούν τη βασική ροή που περιγράφεται από τα στοιχεία του διαγράμματος. Έτσι, οι δραστηριότητες (**Activity**) συνδέονται μεταξύ τους μέσω μεταβάσεων (**Transition**). Κάθε μετάβαση έχει μια δραστηριότητα από την οποία προέρχεται και μια δραστηριότητα στην οποία καταλήγει (ιδιότητες `has_source` και `has_target`, αντίστοιχα), ενώ μπορεί επιπλέον να έχει και μια συνθήκη τύπου **GuardCondition** (ιδιότητα `has_condition`). Επίσης, μια δραστηριότητα (**Activity**) μπορεί να έχει ιδιότητες (**Property**), ενώ μια συνθήκη τύπου **GuardCondition** έχει και μια αντίστροφη, όπου οι συνθήκες συνδέονται μεταξύ τους μέσω της αμφίδρομης (bidirectional) ιδιότητας `is_opposite_of`. Τέλος, κάθε δραστηριότητα (**Activity**) συνδέεται με ένα δράστη (**Actor**), μια ενέργεια (**Action**) και ένα αντικείμενο (**Object**) μέσω των αντίστοιχων ιδιοτήτων (`has_actor`), (`activity_has_action`) και (`activity_has_object`).

**Εξαγωγή Ροής Δραστηριοτήτων από Γραφικά Σενάρια** Τα γραφικά σενάρια (storyboards) είναι διαγράμματα που περιγράφουν ροές ενεργειών σε συστήματα λογισμικού. Ένα διάγραμμα τέτοιου τύπου είναι δομημένο ως μια ροή από έναν *αρχικό κόμβο* (*start node*) σε έναν *τελικό κόμβο* (*end node*). Ανάμεσα στον αρχικό και στον τελικό κόμβο υπάρχουν *ενέργειες* (*actions*) με τις *ιδιότητες* (*properties*) τους και *συνθήκες* (*conditions*). Όλοι οι κόμβοι συνδέονται με ακμές (*edges*). Για τη δημιουργία και επεξεργασία γραφικών σεναρίων σχεδιάσαμε και κατασκευάσαμε το εργαλείο *Storyboard Creator*, ένα plugin του Eclipse που βασίζεται στο Graphical Modeling Framework (GMF). Το εργαλείο φαίνεται στο Σχήμα 3.7, όπου διακρίνεται και ένα παράδειγμα γραφικού σεναρίου.

Το εργαλείο *Storyboard Creator* περιλαμβάνει έναν καμβά (*canvas*) για τη σχεδίαση γραφικών σεναρίων και μια παλέτα (*palette*) για τη δημιουργία κόμβων και ακμών. Υποστηρίζεται επίσης η επικύρωση (*validation*) διαγραμμάτων χρησιμοποιώντας διάφορους κανόνες, π.χ. μπορεί να διασφαλιστεί ότι κάθε διάγραμμα έχει τουλάχιστον έναν κόμβο ενέργειας (*action*), κάθε ιδιότητα (*property*) συνδέεται με ακριβώς μία ενέργεια κ.α.

Το γραφικό σενάριο του Σχήματος 3.7 περιλαμβάνει μια ενέργεια πρόσβασης σε κάποιο λογαριασμό (“Login to account”), η οποία έχει επίσης δύο ιδιότητες που ορίζουν τα βασικά στοιχεία του λογαριασμού: “username” και “password”. Οι συνθήκες έχουν δύο πιθανές διαδρομές. Για παράδειγμα, η συνθήκη “Credentials are correct?”, που ελέγχει την ορθότητα



Σχήμα 3.7: Screenshot του Storyboard Creator

των στοιχείων του λογαριασμού, έχει τη διαδρομή “Yes” που οδηγεί στον τελικό κόμβο και τη διαδρομή “No” που οδηγεί πίσω στη ενέργεια “Login to account” όπου μπορούν να δοθούν νέα διαπιστευτήρια.

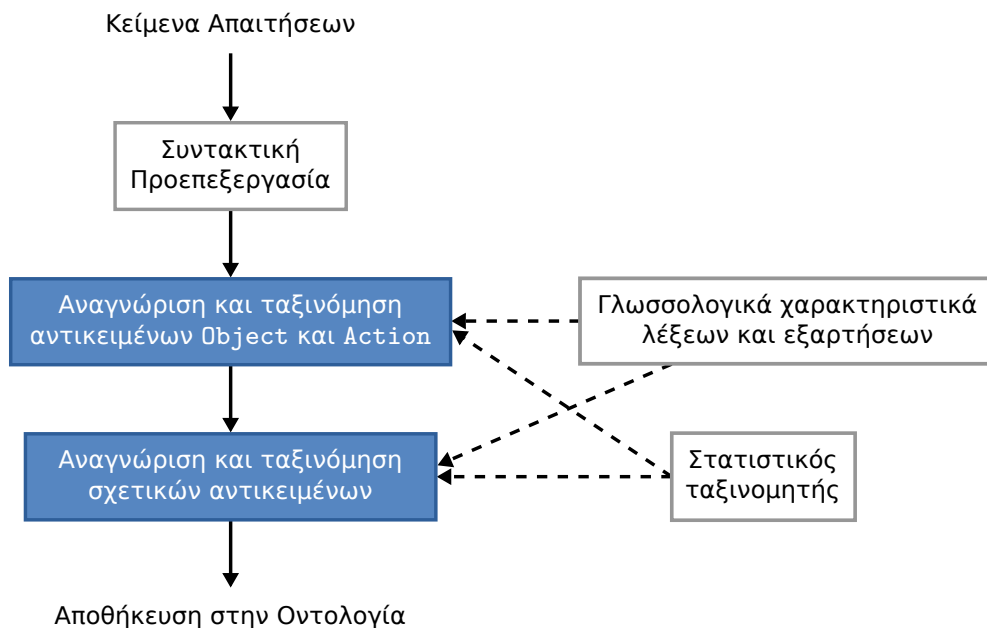
Η εισαγωγή διαγραμμάτων στη δυναμική οντολογία είναι απλή διαδικασία. Οι ενέργειες των σεναρίων αντιστοιχίζονται στην OWL κλάση **Activity** και διαιρούνται περαιτέρω σε αντικείμενα τύπου **Action** και **Object**. Οι ιδιότητες αποθηκεύονται ως αντικείμενα της κλάσης **Property** και συνδέονται με τις αντίστοιχες δραστηριότητες (**Activity**) μέσω της *has\_property*. Οι ακμές και οι ακμές των συνθηκών αποθηκεύονται στην κλάση **Transition**, ενώ κάθε συνθήκη αποθηκεύεται ως ένα ζεύγος αντικειμένων **GuardCondition** που είναι μεταξύ τους αντίστροφα. Ένα παράδειγμα για την αποθήκευση του γραφικού σεναρίου του Σχήματος 3.7 στην οντολογία φαίνεται στον Πίνακα 3.1.

Πίνακας 3.1: Παράδειγμα Αποθήκευσης του Γραφικού Σεναρίου του Σχήματος 3.7

Κλάση OWL	Αντικείμενα
Activity	Login_to_account
Property	Username,Password
Transition	FROM_StartNode_TO_Login_to_account, FROM_Login_to_account_TO_EndNode, FROM_Login_to_account_TO_Login_to_account
GuardCondition	Credentials_are_correct_PATH_Yes, Credentials_are_correct_PATH_No
Action	login
Object	account

### 3.3.3 Αναλυτής για Αυτόματο Σχολιασμό Απαιτήσεων

Με βάση τη μοντελοποίηση που περιγράφηκε στην ενότητα 3.3.2, αναπτύξαμε έναν αναλυτή που μαθαίνει να αντιστοιχεί αυτόματα τις απαιτήσεις λογισμικού γραμμένες σε φυσική γλώσσα σε αντικείμενα της οντολογίας. Αναγνωρίζονται οι έννοιες **Actor**, **Action**, **Object** και **Property**, καθώς και οι σχέσεις μεταξύ τους (**is\_actor\_of**, **acts\_on** και **has\_property**). Οι έννοιες αυτές επιλέχθηκαν καθώς αντιστοιχίζονται με τα βασικά συντακτικά στοιχεία των προτάσεων και είναι αρκετά διαισθητικές ώστε να επιτρέπουν το σχολιασμό χωρίς την ανάγκη βοήθειας από κάποιο ειδικό. Ωστόσο, ο αναλυτής μας θα μπορούσε να προσδιορίσει και πιο ειδικές έννοιες. Στην πράξη, η ανάλυση (parsing) περιλαμβάνει τα εξής βήματα: αρχικά, οι έννοιες πρέπει να εντοπιστούν και να αντιστοιχιστούν στη σχετική κλάση, και, στη συνέχεια, οι σχέσεις μεταξύ εννοιών πρέπει να αναγνωριστούν και να επισημανθούν ανάλογα. Χρησιμοποιούμε μια μεθοδολογία εξαγωγής σχέσης με βάση τα χαρακτηριστικά (feature-based relation extraction), και εστιάζουμε στον τομέα των απαιτήσεων έργων λογισμικού. Αυτό μας επιτρέπει να εφαρμόσουμε τη μεθοδολογία που φαίνεται στο Σχήμα 3.8. Στις ακόλουθες παραγράφους παρέχουμε μια περίληψη της κατασκευής, της εκπαίδευσης (training) και της αξιολόγησης (evaluation) του αναλυτή, ενώ ο αναγνώστης που ενδιαφέρεται για περισσότερες λεπτομέρειες παραπέμπεται στο [141] και στο [142].



Σχήμα 3.8: Μεθοδολογία για την Ανάλυση Απαιτήσεων Λογισμικού

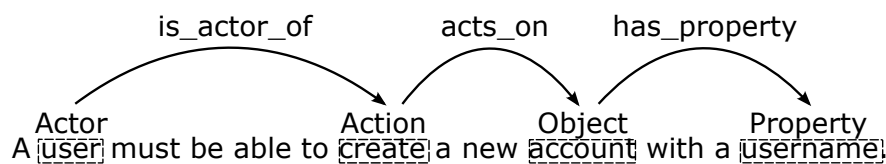
Ο αναλυτής μας αποτελείται από δύο υποσυστήματα, ένα για συντακτική ανάλυση (syntactic analysis) και ένα για σημασιολογική ανάλυση (semantic analysis). Η συντακτική ανάλυση μας επιτρέπει να προσδιορίσουμε την γραμματική κατηγορία κάθε λέξης σε μια πρόταση, ενώ η σημασιολογική ανάλυση χρησιμοποιείται για να αντιστοιχιστούν οι λέξεις μιας φράσης σε αντικείμενα της οντολογίας.

Η φάση της συντακτικής ανάλυσης της μεθοδολογίας μας περιλαμβάνει τα ακόλουθα βήματα: τμηματοποίηση σε λέξεις (tokenization), αναγνώριση μέρους του λόγου για κάθε λέξη (part-of-speech tagging), εύρεση λημμάτων των λέξεων (lemmatization) και ανάλυση εξαρτήσεων (dependency parsing). Για κάθε πρόταση εισόδου, ο αναλυτής μας χωρίζει την πρόταση σε λέξεις, προσδιορίζει την γραμματική κατηγορία κάθε λέξης (π.χ. “user” →

noun - ουσιαστικό, “create” → verb - ρήμα) και προσδιορίζει τον αρχικό τύπο κάθε λέξης (π.χ. “users” → “user”). Τέλος, προσδιορίζονται οι γραμματικές σχέσεις που υπάρχουν ανάμεσα σε δύο λέξεις (π.χ. ⟨“user”, “must”⟩ → subject-of - υποκείμενο του, ⟨“create”, “account”⟩ → object-of - αντικείμενο του). Η μεθοδολογία μας χρησιμοποιεί τα Mate Tools [143, 144]<sup>7</sup>, που περιλαμβάνουν προ-εκπαιδευμένα μοντέλα για συντακτική ανάλυση.

Η έξοδος της συντακτικής προεπεξεργασίας χρησιμοποιείται από το υποσύστημα σημασιολογικής ανάλυσης για την εξαγωγή αντικειμένων για τις κλάσεις της οντολογίας. Η ανάλυση πραγματοποιείται σε τέσσερα βήματα: (1) αναγνώριση εννοιών τύπου Action και Object; (2) κατανομή αυτών στη σωστή έννοια (είτε Action είτε Object); (3) αναγνώριση των υπόλοιπων σχετικών εννοιών (δηλαδή εννοιών τύπου Actor και Property) και (4) προσδιορισμός των σχέσεων των τελευταίων τις έννοιες που αναγνωρίστηκαν στο βήμα (1). Αυτά τα τέσσερα βήματα αντιστοιχούν στις φάσεις *predicate identification*, *predicate disambiguation*, *argument identification* και *argument classification* των Mate Tools [143]. Η μέθοδος μας βασίζεται στο σύστημα εξαγωγής σημασιολογικών ρόλων (semantic role labeling) των Mate Tools και χρησιμοποιεί τον ενσωματωμένο re-ranker για να βρει την καλύτερη δυνατή συνδυαστική έξοδο των βημάτων (3) και (4). Κάθε βήμα στη μεθοδολογία μας υλοποιείται ως ένα μοντέλο λογιστικής παλινδρόμησης (logistic regression) χρησιμοποιώντας το LIBLINEAR toolkit [145]. Το μοντέλο χρησιμοποιεί γλωσσολογικές ιδιότητες (linguistic properties) ως χαρακτηριστικά (features), για τα οποία υπολογίζονται τα κατάλληλα βάρη με βάση τα σχολιασμένα δεδομένα εκπαίδευσης (περισσότερες πληροφορίες για την κατασκευή του αναλυτή είναι διαθέσιμες στο [141] και στο [142]).

Για την εκπαίδευση (training) του αναλυτή χρειάζεται ένα σύνολο από σχολιασμένες λειτουργικές απαιτήσεις. Η διαδικασία του σχολιασμού είναι αρκετά απλή και περιλαμβάνει την επισήμανση των αντικειμένων τύπου Actor, Object, Action και Property που εκφράζονται ρητά σε μια συγκεκριμένη απαίτηση. Για παράδειγμα στην πρόταση του Σχήματος 3.9, ο χρήστης “user” σημειώνεται ως Actor, το ρήμα “create” ως Action, το “account” ως Object και το “username” ως Property<sup>8</sup>.



Σχήμα 3.9: Παράδειγμα Σχολιασμού Πρότασης χρησιμοποιώντας την Ιεραρχική Μεθοδολογία Σχολιασμού

Καθώς ο σχολιασμός μπορεί να είναι μια δύσκολη διαδικασία, ειδικά για άπειρους χρήστες, δημιουργήσαμε ένα εργαλείο σχολιασμού, το οποίο ονομάζεται *Requirements Annotation Tool*, για να διευκολυνθούν οι χρήστες. Το εργαλείο είναι μια εφαρμογή διαδικτύου (web application) όπου οι χρήστες μπορούν να δημιουργήσουν έναν λογαριασμό, να εισάγουν ένα ή περισσότερα από τα έργα τους και να τα σχολιάσουν. Πρακτικά το εργαλείο είναι παρόμοιο με τον Requirements Editor που παρουσιάστηκε στην υποενότητα 3.3.2.1, ωστόσο ο βασικός σκοπός του Requirements Annotation Tool είναι ο σχολιασμός για την

<sup>7</sup><http://code.google.com/p/mate-tools/>

<sup>8</sup>Θα μπορούσε επίσης να υπάρξει πιο εξειδικευμένος σχολιασμός χρησιμοποιώντας την ιεραρχική μεθοδολογία που παρουσιάζεται στο [142].

εκπαίδευση του αναλυτή, οπότε παρέχει ένα βολικό περιβάλλον με drag n’ drop δυνατότητες. Τα δύο εργαλεία είναι συμβατά καθώς επιτρέπουν την εισαγωγή/εξαγωγή σχολιασμών χρησιμοποιώντας τους ίδιους τύπους αρχείων.

Ως παράδειγμα χρήσης του εργαλείου σχολιασμού μας, χρησιμοποιούμε τις απαιτήσεις του έργου *Restmarks*. Το *Restmarks* είναι ένα παράδειγμα μιας υπηρεσίας (service) κοινωνικού δικτύου για σελιδοδείκτες (bookmarks). Οι χρήστες της υπηρεσίας μπορούν να αποθηκεύουν και να ανακτούν τους σελιδοδείκτες τους, να τους μοιράζονται με την κοινότητα και να αναζητούν σελιδοδείκτες χρησιμοποιώντας ετικέτες (tags). Στο Σχήμα 3.10 παρουσιάζεται ένα screenshot του εργαλείου με το σχολιασμό για το έργο *Restmarks*.



Σχήμα 3.10: Σχολιασμένες Απαιτήσεις του Έργου Restmarks

Οι σχολιασμοί είναι αρκετά κατανοητοί· ακόμη και οι χρήστες χωρίς εμπειρία θα πρέπει να είναι σε θέση να προσδιορίσουν και να επισημάνουν τις κατάλληλες οντότητες και σχέσεις. Το εργαλείο εξάγει σχολιασμούς σε διάφορες μορφές, συμπεριλαμβανομένων των OWL και TTL. Ο Πίνακας 3.2 απεικονίζει τις οντότητες του Restmarks που εξήχθησαν.

Πίνακας 3.2: Αντικείμενα της Οντολογίας για τις Οντότητες του Restmarks

Κλάση OWL	Αντικείμενα
Project	Restmarks
Requirement	FR1, FR2, FR3, FR4, FR5, FR6, FR7, FR8, FR9, FR10, FR11, FR12, FR13
Actor	user, users
Action	create, retrieve, mark, search, delete, add, providing, login, update
Object	account, bookmark, bookmarks, password_1, tags
Property	RESTMARKS, password, tag, public, private, logged_in_to_his_account, logged_in, username, user_1

Το εργαλείο εξάγει επίσης τις ιδιότητες της οντολογίας. Για παράδειγμα, οι ιδιότητες για την απαίτηση FR4 του Restmarks φαίνονται στον Πίνακα 3.3.

Πίνακας 3.3: Ιδιότητες της Οντολογίας για την Απαίτηση FR4 του Restmarks

Αντικείμενο	Ιδιότητα OWL	Αντικείμενο
user	is_actor_of	add
add	has_actor	user
add	acts_on	bookmark
bookmark	receives_action	add

Τέλος, χρησιμοποιώντας το εργαλείο που αναπτύξαμε, καταφέραμε να κατασκευάσουμε ένα σύνολο δεδομένων σχολιασμένων απαιτήσεων που χρησιμοποιήθηκαν για την εκπαίδευση του αναλυτή μας. Συγκεντρώσαμε απαιτήσεις από διάφορες πηγές, όπως απαιτήσεις από έγγραφα που έχουν γραφτεί από φοιτητές ως μέρος πανεπιστημιακών εργασιών<sup>9</sup>, απαιτήσεις από πραγματικά έργα λογισμικού, καθώς και απαιτήσεις RESTful υπηρεσιών από το έργο S-CASE<sup>10</sup>. Συνολικά συλλέξαμε 325 προτάσεις, ενώ μετά τη διαδικασία τμηματοποίησης σε λέξεις (tokenization) προέκυψαν 4057 λέξεις. Κατόπιν, οι προτάσεις σχολιάστηκαν από δύο διαφορετικούς σχολιαστές (annotators) και επανεξετάστηκαν προσεκτικά για να δημιουργηθεί τελικά ένα σχολιασμένο σύνολο δεδομένων καλής ποιότητας (περισσότερες πληροφορίες για τη διαδικασία σχολιασμού είναι διαθέσιμες στο [142]). Το τελικό σύνολο δεδομένων περιλαμβάνει 2051 οντότητες (435 ενέργειες, 305 δράστες, 613 αντικείμενα και 698 ιδιότητες) και 1576 σχέσεις μεταξύ τους (355 σχέσεις has\_actor, 531 σχέσεις acts\_on και 690 σχέσεις has\_property).

Για την αξιολόγηση του αναλυτή μας, χρησιμοποιούμε τις μετρικές της ακρίβειας (precision) και της ανάκλησης (recall). Ορίζουμε την *ακρίβεια* ως το ποσοστό των αντικειμένων που επισημάνθηκαν σωστά σε σχέση με όλα τα αντικείμενα που επισημάνθηκαν. Η *ανάκληση* ορίζεται ως το ποσοστό των αντικειμένων που επισημάνθηκαν σωστά από τον

<sup>9</sup>Η πλειοψηφία των απαιτήσεων από φοιτητές προέρχονται από ένα μάθημα για την ανάπτυξη λογισμικού που οργανώθηκε από κοινού από πολλά ευρωπαϊκά πανεπιστήμια, το οποίο είναι διαθέσιμο στην ηλεκτρονική διεύθυνση <http://www.fer.unizg.hr/rasip/dsd>.

<sup>10</sup><https://s-case.github.io>

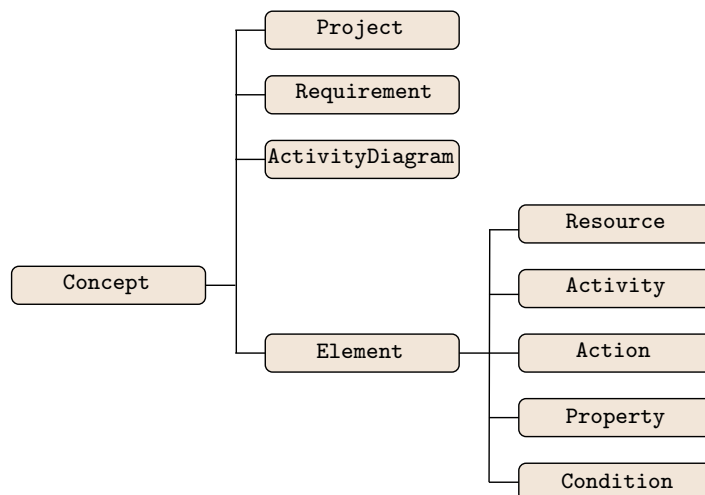
αναλυτή σε σχέση με όλα τα αντικείμενα που θα έπρεπε να έχουν επισημανθεί. Για να εκπαιδύσουμε (training) και να ελέγξουμε (testing) τα στατιστικά μοντέλα που πλαισιώνουν τη σημασιολογική ανάλυσή μας, χρησιμοποιούμε cross-validation με 5 folds. Δηλαδή, λαμβάνοντας υπόψη τις 325 προτάσεις από το σύνολο δεδομένων, δημιουργούμε τυχαία πέντε σύνολα (folds) ίδιου μεγέθους (με 65 προτάσεις το καθένα) και κρατάμε κάθε φορά ένα σύνολο για έλεγχο (testing), ενώ εκπαιδύουμε το μοντέλο χρησιμοποιώντας τα υπόλοιπα σύνολα. Στο πλαίσιο αυτής της διαδικασίας αξιολόγησης, το μοντέλο μας επιτυγχάνει ακρίβεια και ανάκληση 77.9% και 74.5%, αντίστοιχα. Όπως φαίνεται από αυτές τις τιμές, ο αναλυτής μας είναι αρκετά αποτελεσματικός όσον αφορά την αυτοματοποίηση της εξαγωγής αντικειμένων από απαιτήσεις. Συγκεκριμένα, η τιμή της ανάκλησης δείχνει ότι αναγνωρίζεται σωστά το 75% περίπου όλων των σχολιασμένων εννοιών και σχέσεων. Επιπλέον, η τιμή της ακρίβειας δείχνει ότι περίπου 4 στους 5 σχολιασμούς είναι σωστοί. Έτσι, ο αναλυτής μας μειώνει σημαντικά την προσπάθεια (και το χρόνο) που απαιτείται για την αναγνώριση των αντικειμένων σε λειτουργικές απαιτήσεις και τον αντίστοιχο σχολιασμό τους.

### 3.3.4 Εξαγωγή Προδιαγραφών από Αντικείμενα Λογισμικού

Οι οντολογίες που αναπτύχθηκαν στην ενότητα 3.3.2 αποτελούν από μόνες τους ένα μοντέλο κατάλληλο για την αποθήκευση απαιτήσεων, καθώς και για ενέργειες όπως η επικύρωση απαιτήσεων και η εξόρυξη απαιτήσεων (βλέπε επόμενο κεφάλαιο). Ο συνδυασμός των δύο αυτών οντολογιών μπορεί να οδηγήσει σε ένα τελικό μοντέλο που θα περιγράφει το υπό εξέλιξη σύστημα τόσο από στατική όσο και από δυναμική πλευρά. Αυτό το μοντέλο μπορεί στη συνέχεια να χρησιμοποιηθεί για τη σχεδίαση ή ακόμα και για την αυτοματοποιημένη ανάπτυξη του τελικού συστήματος (όπως γίνεται στο [140]). Χωρίς βλάβη της γενικότητας, σχεδιάζουμε ένα αθροιστικό μοντέλο για διαδικτυακές RESTful υπηρεσίες (RESTful web services) και κατασκευάζουμε μια αναπαράσταση σε YAML που λειτουργεί ως το τελικό μοντέλο της υπηρεσίας που αναπτύσσεται.

#### 3.3.4.1 Αθροιστική Οντολογία Έργων Λογισμικού

Η ιεραρχία των κλάσεων της αθροιστικής οντολογίας παρουσιάζεται στο Σχήμα 3.11. Τα αντικείμενα τύπου `Concept` ανήκουν σε μία από τις κλάσεις `Project`, `Requirement`, `ActivityDiagram` και `Element`. Η κλάση `Project` αναφέρεται στο έργο λογισμικού που αποθηκεύεται στην οντολογία, ενώ στις κλάσεις `Requirement` και `ActivityDiagram` αποθηκεύονται οι απαιτήσεις και τα διαγράμματα της στατικής και της δυναμικής οντολογίας, αντίστοιχα. Τα υπόλοιπα αντικείμενα τύπου `Concept` αποτελούν κάποια στοιχεία (`Element`) του έργου λογισμικού. Ένα αντικείμενο τύπου `Element` μπορεί να είναι ένας πόρος (`Resource`), μια δραστηριότητα (`Activity`), μια ενέργεια (`Action`), μια ιδιότητα (`Property`) ή μια συνθήκη (`Condition`). Οι πόροι (αντικείμενα τύπου `Resource`) είναι τα δομικά στοιχεία οποιασδήποτε RESTful υπηρεσίας, ενώ στα αντικείμενα τύπου `Action` αποθηκεύονται οι CRUD ενέργειες (Create-Read-Update-Delete) που εφαρμόζονται σε αυτούς. Η κλάση `Activity` αναφέρεται σε μια δραστηριότητα του συστήματος (π.χ. “create bookmark”) που συνδέεται με κάποιο `Resource` (π.χ. “bookmark”) και κάποιο CRUD `Action` (π.χ. “create”). Στην κλάση `Property` αποθηκεύονται οι παράμετροι που μπορεί να απαιτούνται για κάποια δραστηριότητα (π.χ. η παράμετρος “bookmark name” μπορεί να απαιτείται για τη δραστηριότητα “create bookmark”). Τέλος, τα αντικείμενα `Condition` αφορούν κριτήρια που πρέπει να πληρούνται για να εκτελεστεί μια δραστηριότητα.



Σχήμα 3.11: Αθροιστική Οντολογία Έργων Λογισμικού

Οι ιδιότητες της αθροιστικής οντολογίας φαίνονται στον Πίνακα 3.4. Οι ιδιότητες που είναι σχετικές με τις κλάσεις **Project**, **Requirement**, **ActivityDiagram** και **Element** χρησιμοποιούνται για να διασφαλίσουν ότι όλες οι έννοιες της οντολογίας συνδέονται με τα αντίστοιχα αντικείμενα στις άλλες δύο οντολογίες.

Πίνακας 3.4: Ιδιότητες Αθροιστικής Οντολογίας

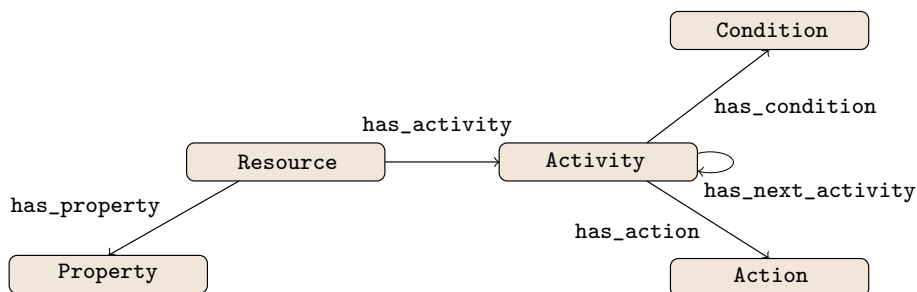
Κλάση OWL	Ιδιότητα	Κλάση OWL
Project	has_requirement	Requirement
Requirement	is_requirement_of	Project
Project	has_activity_diagram	ActivityDiagram
ActivityDiagram	is_activity_diagram_of	Project
Project	has_element	Element
Element	is_element_of	Project
Requirement/ ActivityDiagram	contains_element	Element
Element	element_is_contained_in	Requirement/ ActivityDiagram
Resource	has_activity	Activity
Activity	is_activity_of	Resource
Resource	has_property	Property
Property	is_property_of	Resource
Activity	has_action	Action
Action	is_action_of	Activity
Activity	has_condition	Condition
Property	is_condition_of	Activity
Activity	has_next_activity	Activity
Activity	has_previous_activity	Activity

Οι ιδιότητες μεταξύ των κλάσεων της οντολογίας περιστρέφονται γύρω από τις δύο βασικές υποκλάσεις του **Element**, την κλάση **Resource** και την κλάση **Activity**, κα-



θώς αυτές είναι και οι δύο βασικές κλάσεις κάθε RESTful υπηρεσίας. Κάθε πόρος του συστήματος (**Resource**) μπορεί να συνδέεται με ιδιότητες (**Property**) και δραστηριότητες (**Activity**) μέσω των σχέσεων `has_property/is_property_of` and `has_activity/is_activity_of`, αντίστοιχα. Η κλάση **Activity** συνδέεται με αντικείμενα τύπου **Action** (μέσω των `has_action/is_action_of`) και αντικείμενα του τύπου **Condition** (μέσω των `has_condition/is_condition_of`), επειδή είναι απαραίτητο να αποθηκεύονται οι CRUD ενέργειες καθώς επίσης και οι συνθήκες που πρέπει να πληρούνται ώστε η δραστηριότητα να είναι έγκυρη. Οι μεταβάσεις μοντελοποιούνται χρησιμοποιώντας τις ιδιότητες `has_next_activity/has_previous_activity`.

Οι ιδιότητες παρουσιάζονται επίσης στο Σχήμα 3.12 (όπου, για λόγους απλότητας, δεν απεικονίζονται οι ιδιότητες που είναι σχετικές με τις κλάσεις **Project**, **Requirement** και **ActivityDiagram**, ενώ επιπλέον περιλαμβάνεται μόνο μία ιδιότητα για κάθε ζεύγος ιδιότητας και αντίστροφης), όπου είναι σαφές ότι οι κλάσεις **Resource** και **Activity** έχουν κεντρικούς ρόλους στην αθροιστική οντολογία.



Σχήμα 3.12: Ιδιότητες Αθροιστικής Οντολογίας

Τα αντικείμενα που εισάγονται στην αθροιστική οντολογία έχουν προηγουμένως εξαχθεί από τη στατική και τη δυναμική οντολογία. Η στατική οντολογία περιέχει διάφορες κλάσεις που αναφέρονται στη στατική όψη του συστήματος. Από τις κλάσεις αυτές, εστιάζουμε σε ενέργειες που εκτελούνται σε αντικείμενα και στις ιδιότητες αυτών των αντικειμένων. Στη στατική οντολογία, τα στοιχεία αυτά αποθηκεύονται στις OWL κλάσεις **OperationType**, **Object** και **Property**. Όσον αφορά τα δυναμικά στοιχεία ενός συστήματος λογισμικού, η αντίστοιχη οντολογία καλύπτει όχι μόνο τις ενέργειες, τα αντικείμενα και τις ιδιότητες, αλλά και τις συνθήκες των ενεργειών. Οι αντίστοιχες OWL κλάσεις είναι οι κλάσεις **Action**, **Object**, **Property** και **GuardCondition**. Εκτός από τις παραπάνω κλάσεις, αποθηκεύουμε επιπλέον το **Project**, καθώς και τα αντικείμενα τύπου **Requirement** και **ActivityDiagram** που προέρχονται από τη στατική και τη δυναμική οντολογία αντίστοιχα. Αυτές οι τρεις κλάσεις διασφαλίζουν ότι οι οντολογίες μας είναι ανιχνεύσιμες (traceable) και ότι συνδέονται στενά μεταξύ τους. Η αντιστοίχιση των κλάσεων OWL της στατικής και της δυναμικής οντολογίας στις κλάσεις OWL της αθροιστικής οντολογίας παρουσιάζεται στον Πίνακα 3.5.

Όπως φαίνεται σε αυτόν τον Πίνακα, τα αντικείμενα τύπου **Requirement** και **ActivityDiagram** μεταφέρονται στην αθροιστική οντολογία, ενώ η κλάση **Project** διασφαλίζει ότι οι οντολογίες αναφέρονται όλες στο ίδιο έργο λογισμικού. Όσον αφορά τις υπόλοιπες κλάσεις της αθροιστικής οντολογίας, αρκετές από αυτές απαιτούν το συνδυασμό στοιχείων από τις δύο οντολογίες. Έτσι, κάθε **Object** της στατικής οντολογίας και κάθε **Object** της δυναμικής οντολογίας εισάγονται στην αθροιστική οντολογία το ένα μετά το άλλο. Στην

Πίνακας 3.5: Αντιστοίχιση Κλάσεων από τη Στατική και τη Δυναμική Οντολογία στην Αθροιστική Οντολογία

Κλάση OWL της Στατικής Οντολογίας	Κλάση OWL της Δυναμικής Οντολογίας	Κλάση OWL της Αθροιστικής Οντολογίας
Project	Project	Project
Requirement	-	Requirement
-	ActivityDiagram	ActivityDiagram
OperationType	Action	Activity
-	GuardCondition	Condition
Object	Object	Resource
Property	Property	Property

περίπτωση που υπάρχει ήδη ένα αντικείμενο στην αθροιστική οντολογία, τότε απλά δεν εισάγεται. Ωστόσο, οι εκάστοτε ιδιότητες του αντικειμένου εισάγονται κανονικά (αν δεν έχουν ήδη εισαχθεί): έτσι διασφαλίζεται ότι η οντολογία είναι πλήρης, χωρίς όμως να περιλαμβάνει περιττές πληροφορίες. Η αντιστοίχιση για τις ιδιότητες OWL παρουσιάζεται στον Πίνακα 3.6.

Πίνακας 3.6: Αντιστοίχιση Ιδιοτήτων από τη Στατική και τη Δυναμική Οντολογία στην Αθροιστική Οντολογία

Ιδιότητα OWL της Στατικής Οντολογίας	Ιδιότητα OWL της Δυναμικής Οντολογίας	Ιδιότητα OWL της Αθροιστικής Οντολογίας
project_has	-	has_requirement
is_of_project	-	is_requirement_of
-	project_has_diagram	has_activity_diagram
-	is_diagram_of_project	is_activity_diagram_of
consists_of	diagram_has	contains_element
consist	is_of_diagram	element_is_contained_in
receives_action	is_object_of	has_activity
acts_on	has_object	is_activity_of
has_property	has_property*	has_property
is_property_of	is_property_of*	is_property_of
-	has_action	has_action
-	is_action_of	is_action_of
-	has_condition*	has_condition
-	is_condition_of*	is_condition_of
-	has_target	has_next_activity
-	has_source	has_previous_activity

\* εξαγόμενη ιδιότητα

Σημειώστε ότι ορισμένες ιδιότητες δεν αντιστοιχίζονται άμεσα μεταξύ των οντολογιών. Στην περίπτωση αυτή, οι ιδιότητες μπορούν να προέρχονται έμμεσα από ενδιάμεσα αντικείμενα. Για παράδειγμα, η ιδιότητα `has_property` της αθροιστικής οντολογίας κατευθύνεται από ένα `Resource` σε ένα `Property`. Ωστόσο, στη δυναμική οντολογία, οι ιδιότητες

(Property) συνδέονται με δραστηριότητες (Activity). Επομένως, για κάθε Activity (π.χ. “Create bookmark”) χρειάζεται να βρεθεί το αντίστοιχο Object (“bookmark”) και, εφόσον το τελευταίο προστεθεί στην αθροιστική οντολογία, στη συνέχεια πρέπει να βρεθούν και τα αντίστοιχα αντικείμενα Property του Activity (π.χ. “bookmark name”) και να προστεθούν και αυτά στην οντολογία μαζί με τις αντίστοιχες συνδέσεις. Με αντίστοιχο τρόπο παίρνουν τιμές και οι ιδιότητες των συνθηκών (has\_condition και is\_condition\_of) από τις συνθήκες (GuardCondition) της προηγούμενης μετάβασης (Transition).

### 3.3.4.2 Εξαγωγή Αναπαράστασης YAML από Αντικείμενα Οντολογίας

Μετά την αποθήκευση των αντικειμένων στην αθροιστική οντολογία, το επόμενο βήμα είναι η μετατροπή τους στις προδιαγραφές της προς ανάπτυξη υπηρεσίας. Οι προδιαγραφές (specifications) μπορούν να καθοριστούν και να αποθηκευτούν σε διάφορες μορφές, π.χ. ως μοντέλα CIM (Computationally Independent Models), ως αντικείμενα UML, κ.α. Στην περίπτωση μας σχεδιάζουμε μια αναπαράσταση σε γλώσσα YAML<sup>11</sup>, η οποία θα περιγράφει αποτελεσματικά τα βασικά στοιχεία της υπό ανάπτυξη υπηρεσίας και θα παρέχει στον χρήστη τη δυνατότητα να τροποποιήσει (fine-grain) το μοντέλο. Κατόπιν, το μοντέλο YAML μπορεί να χρησιμοποιηθεί ως σχεδιαστικό πρότυπο ή ακόμα και για να παραχθεί απευθείας ένα CIM, επιτρέποντας έτσι στους προγραμματιστές να έχουν μια σαφή εικόνα των προδιαγραφών και σε ορισμένες περιπτώσεις ακόμη και να αυτοματοποιήσουν την κατασκευή της υπηρεσίας (π.χ. όπως στο [140]). Το πρότυπο YAML υποστηρίζει αρκετές γνωστές δομές δεδομένων, καθώς έχει σχεδιαστεί για να αντιστοιχίζεται εύκολα σε γλώσσες προγραμματισμού. Στην περίπτωσή μας, χρησιμοποιούμε λίστες και συσχετιζόμενους πίνακες (associative arrays, key-value pairs) για να δημιουργήσουμε μια δομή για τους πόρους, τις ιδιότητές τους και τους διάφορους τύπους πληροφοριών που πρέπει να αποθηκευτούν για κάθε πόρο. Το σχήμα (schema) της αναπαράστασής μας παρουσιάζεται στο Σχήμα 3.13, όπου το βασικό στοιχείο είναι ο RESTful πόρος (Resource).

---

```

- !!Resource
  Name: String
  IsAlgorithmic: Boolean
  CRUDActivities: List of Create, Read, Update, and/or Delete
  Properties:
  - Name: String
    Type: Integer/Float/String/Boolean/null
    Unique: Boolean
    NamingProperty: Boolean
  - ...
  RelatedResources: List of String

```

---

Σχήμα 3.13: Σχήμα της Αναπαράστασης YAML

Ένα έργο αποτελείται από μια λίστα από πόρους. Για κάθε πόρο ορίζονται διάφορα πεδία, το καθένα με δικό του τύπο δεδομένων και δικές του επιτρεπόμενες τιμές. Κατ’ αρχάς,

<sup>11</sup><http://yaml.org/>

κάθε πόρος πρέπει να έχει ένα όνομα, το οποίο πρέπει επίσης να είναι μοναδικό. Επιπρόσθετα, κάθε πόρος μπορεί να είναι είτε αλγοριθμικός (algorithmic resource, δηλαδή πόρος που απαιτεί τη συγγραφή κώδικα που σχετίζεται με κάποιο αλγόριθμο ή με την κλήση κάποιας εξωτερικής υπηρεσίας) ή μη αλγοριθμικός (non-algorithmic resource). Η δυνατότητα αυτή εκφράζεται με το πεδίο `IsAlgorithmic`. Τα ρήματα/ενέργειες CRUD (ή συνώνυμα που μπορούν να μεταφραστούν σε CRUD ρήματα) συνήθως δεν εφαρμόζονται σε αλγοριθμικές πηγές. Υπάρχουν τέσσερις τύποι δραστηριοτήτων που μπορούν να εφαρμοστούν στους πόρους, που συνάδουν με τις CRUD ενέργειες (Create, Read, Update και Delete). Κάθε πόρος μπορεί να υποστηρίζει μία ή περισσότερες από αυτές τις ενέργειες, δηλαδή οποιοδήποτε συνδυασμό τους, ως μια λίστα (`CRUDActivities`).

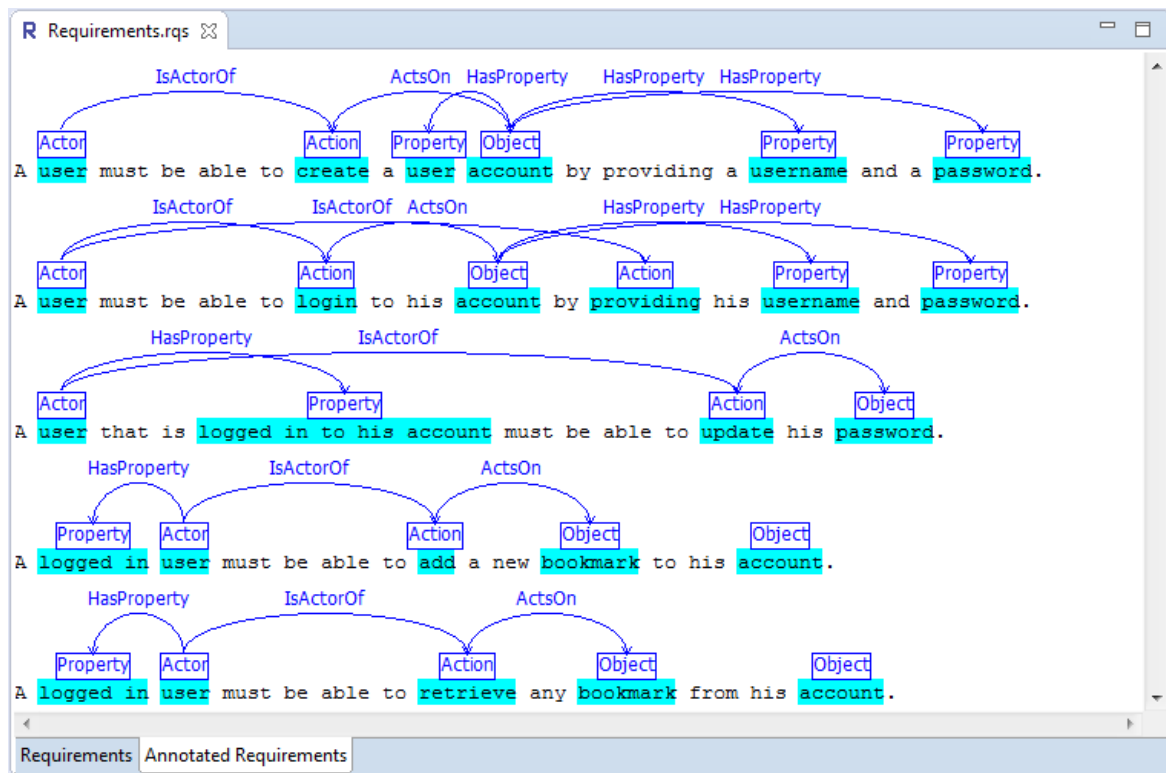
Οι πόροι έχουν επίσης ιδιότητες, οι οποίες ορίζονται ως μια λίστα αντικειμένων. Κάθε ιδιότητα έχει ένα όνομα (`Name`), που είναι αλφαριθμητικό, καθώς και έναν τύπο (`Type`), που αντιστοιχεί στους συνήθεις τύπους δεδομένων των γλωσσών προγραμματισμού, δηλαδή ακέραιους (integers), δεκαδικούς (floats), αλφαριθμητικούς (strings) και λογικές μεταβλητές (booleans). Επιπλέον, κάθε ιδιότητα έχει δύο λογικά (boolean) πεδία: `Unique` και `NamingProperty`. Το πρώτο εξ αυτών δηλώνει εάν η ιδιότητα έχει μια μοναδική τιμή για κάθε εμφάνιση του πόρου, ενώ το δεύτερο δηλώνει εάν ο πόρος παίρνει το όνομά του από την τιμή αυτής της ιδιότητας. Για παράδειγμα, ένας πόρος “user” θα μπορούσε να έχει τις ιδιότητες “username” και “email account”. Σε αυτήν την περίπτωση, το όνομα χρήστη (username) θα ήταν ενδεχομένως μοναδικό, ενώ ο λογαριασμός ηλεκτρονικού ταχυδρομείου (email account) θα μπορούσε είτε να είναι μοναδικός είτε όχι (π.χ. ένας χρήστης μπορεί να επιτρέπεται να δηλώσει περισσότερα από ένα email). Ωστόσο, κάθε πόρος τύπου user θα πρέπει να αναγνωρίζεται με μοναδικό τρόπο στο σύστημα. Έτσι, εάν δεν επιτρέψουμε σε δύο χρήστες να έχουν το ίδιο όνομα χρήστη, θα μπορούσαμε να δηλώσουμε την ιδιότητα username ως ιδιότητα από την οποία ο πόρος παίρνει το όνομά του, θέτοντας κατάλληλα το λογικό πεδίο `NamingProperty`. Τέλος, κάθε πόρος μπορεί να έχει σχετικούς πόρους (related resources). Το πεδίο `RelatedResources` είναι μια λίστα με αλφαριθμητικές τιμές που αντιστοιχούν στα ονόματα άλλων πόρων.

Η εξαγωγή πληροφοριών από την αθροιστική οντολογία και η δημιουργία του αντίστοιχου αρχείου YAML είναι μια σχετικά απλή διαδικασία. Αρχικά, τα αντικείμενα της OWL κλάσης `Resource` μπορούν να αντιστοιχιστούν άμεσα σε αντικείμενα YAML τύπου `Resource`. Κάθε πόρος αρχικά θεωρείται ως μη αλγοριθμικός. Η ροή των δραστηριοτήτων και των συνθηκών χρησιμοποιείται για να βρει τους τύπους των ρημάτων που χρησιμοποιούνται σε κάθε πόρο καθώς και τους σχετικούς πόρους (related resources). Έτσι, για παράδειγμα, αν έχουμε μια δραστηριότητα “Add bookmark” ακολουθούμενη από μια δραστηριότητα “Add tag”, μπορούμε να προσδιορίσουμε δύο πόρους, “bookmark” και “tag”, όπου ο πόρος “tag” επίσης ένας σχετικός πόρος για τον πόρο “bookmark”. Επιπλέον, και για τους δύο πόρους, “bookmark” και “tag”, θα πρέπει να επιτρέπεται η CRUD ενέργεια “Create” CRUD, καθώς το ρήμα “add” συνεπάγεται τη δημιουργία ενός νέου αντικειμένου. Ο τύπος ενέργειας για κάθε ρήμα αναγνωρίζεται χρησιμοποιώντας ένα λεξικό (lexicon). Κάθε φορά που ένα ρήμα δεν μπορεί να ταξινομηθεί σε οποιονδήποτε από τους τέσσερις τύπους ενεργειών CRUD, δημιουργείται ένας νέος αλγοριθμικός πόρος. Έτσι, για παράδειγμα, μια δραστηριότητα “Search user” θα δημιουργούσε το νέο αλγοριθμικό πόρο “userSearch” και θα τον συνέδεε ως σχετικό πόρο με τον πόρο “user”. Τέλος, οι ιδιότητες των πόρων αντιστοιχίζονται ως αντικείμενα στο πεδίο `Properties`.

### 3.4 Μελέτη Περίπτωσης

Στο υποκεφάλαιο αυτό, παρέχουμε μια μελέτη περίπτωσης (case study) για το έργο Restmarks, το οποίο παρουσιάστηκε στην ενότητα 3.3.3. Θεωρούμε το Restmarks ως μια υπηρεσία που επιτρέπει στους χρήστες να αποθηκεύουν και να ανακτούν τους σελιδοδείκτες (bookmarks) τους, να τους μοιράζονται με άλλους χρήστες και να αναζητούν σελιδοδείκτες χρησιμοποιώντας ετικέτες (tags). Πρακτικά, είναι μια υπηρεσία (service) κοινωνικού δικτύου για σελιδοδείκτες. Στις επόμενες παραγράφους, δείχνουμε πώς μπορούμε να χρησιμοποιήσουμε το σύστημα μας βήμα προς βήμα, όπως στο Σχήμα 3.1, για να δημιουργήσουμε μια τέτοια υπηρεσία εύκολα, ενώ παράλληλα να διασφαλίσουμε ότι είναι πλήρως λειτουργική και ανιχνεύσιμη (traceable).

Το πρώτο βήμα της διαδικασίας περιλαμβάνει την εισαγωγή και το σχολιασμό των λειτουργικών απαιτήσεων. Ένα ενδεικτικό τμήμα των απαιτήσεων του Restmarks, με τους σχετικούς σχολιασμούς χρησιμοποιώντας το Requirements Editor, απεικονίζεται στο Σχήμα 3.14.



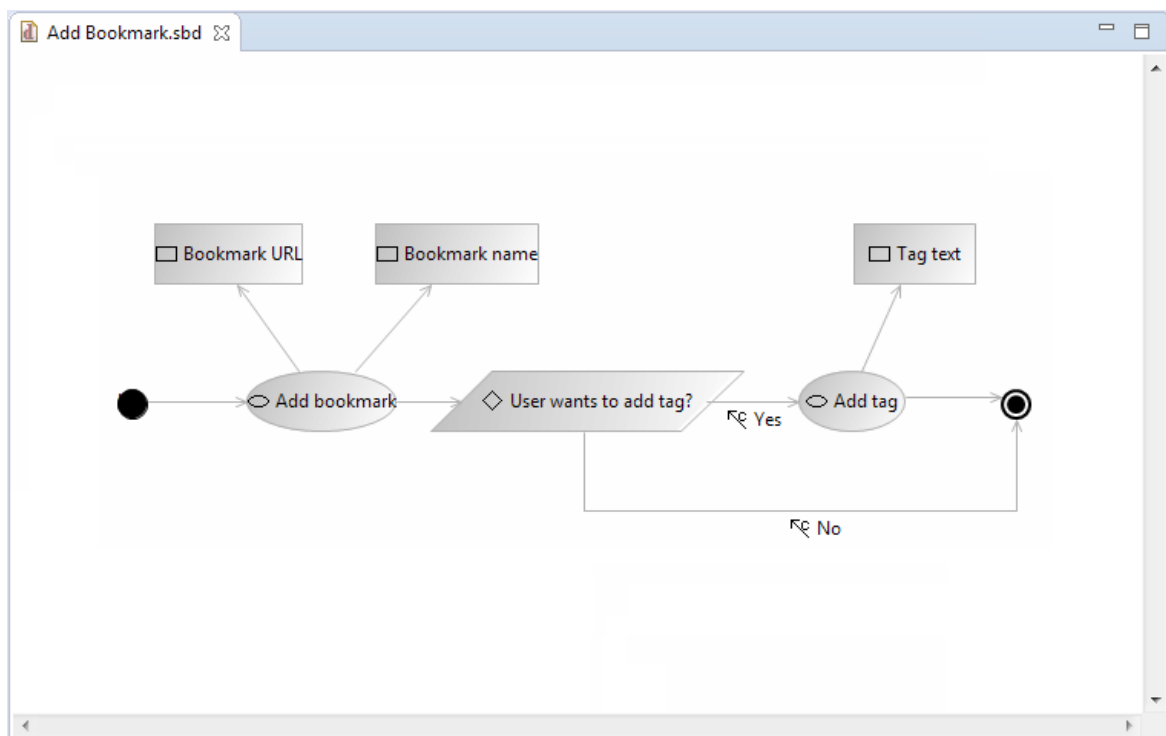
Σχήμα 3.14: Τμήμα των Σχολιασμένων Λειτουργικών Απαιτήσεων του Έργου Restmarks

Μετά την εισαγωγή των απαιτήσεων, ο χρήστης πρέπει να εισάγει πληροφορίες σχετικά με τη δυναμική όψη του συστήματος. Στο παράδειγμα αυτό, η δυναμική αναπαράσταση του συστήματος δίνεται με τη μορφή γραφικών σεναρίων (storyboards). Υποθέτουμε ότι το Restmarks έχει τα ακόλουθα δυναμικά σενάρια:

1. Add Bookmark: Ο χρήστης προσθέτει ένα σελιδοδείκτη στη συλλογή του και προαιρετικά προσθέτει και μια ετικέτα στον σελιδοδείκτη.
2. Create Account: Ο χρήστης δημιουργεί ένα νέο λογαριασμό.

3. Delete Bookmark: Ο χρήστης διαγράφει έναν από τους σελιδοδείκτες του.
4. Login to Account: Ο χρήστης συνδέεται στο λογαριασμό του.
5. Search Bookmark by Tag System Wide: Ο χρήστης αναζητά σελιδοδείκτες δίνοντας το όνομα μιας ετικέτας. Η αναζήτηση περιλαμβάνει όλους τους δημόσιους σελιδοδείκτες.
6. Search Bookmark by Tag User Wide: Ο χρήστης αναζητά σελιδοδείκτες δίνοντας το όνομα μιας ετικέτας. Η αναζήτηση περιλαμβάνει τους δημόσιους σελιδοδείκτες και τους ιδιωτικούς σελιδοδείκτες του χρήστη.
7. Show Bookmark: Το σύστημα εμφανίζει ένα συγκεκριμένο σελιδοδείκτη στον χρήστη.
8. Update Bookmark: Ο χρήστης ενημερώνει τις πληροφορίες σε έναν από τους σελιδοδείκτες του.

Τα storyboards των παραπάνω σεναρίων δημιουργούνται χρησιμοποιώντας το εργαλείο Storyboard Creator. Ένα παράδειγμα γραφικού σεναρίου χρησιμοποιώντας αυτό το εργαλείο φαίνεται στο Σχήμα 3.15. Το storyboard αυτού του Σχήματος αναφέρεται στο πρώτο σενάριο της προσθήκης ενός νέου σελιδοδείκτη. Τα υπόλοιπα σεναρία του έργου εισάγονται αντίστοιχα.



Σχήμα 3.15: Διάγραμμα Σεναρίου “Add bookmark” του Έργου Restmarks

Μετά τη κατασκευή των απαιτήσεων και των γραφικών σεναρίων, το επόμενο βήμα είναι η δημιουργία της στατικής και της δυναμικής οντολογίας. Οι δύο οντολογίες συνδυάζονται για να παρέχουν την αθροιστική οντολογία. Τα αντικείμενα των κλάσεων **Resource**, **Property** και **Activity** της αθροιστικής οντολογίας παρουσιάζονται στον Πίνακα 3.7.

Πίνακας 3.7: Αντικείμενα Κλάσεων Resource, Property και Activity για το έργο Restmarks

Κλάση OWL	Αντικείμενα
Resource	bookmark, tag, account, password
Property	username, password, private, public, user
Activity	search_bookmark, add_tag, update_bookmark, delete_bookmark, mark_bookmark, retrieve_bookmark, delete_tag, add_bookmark, update_tag, update_password, login_account, create_account, get_bookmark

Τέλος, από την αθροιστική οντολογία εξάγεται η αναπαράσταση YAML που περιγράφει το έργο. Το αρχείο YAML για το έργο Restmarks φαίνεται στο Σχήμα 3.16.

```

- !!Resource
  Name: account
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [username, password]
  RelatedResources: [bookmark]
- !!Resource
  Name: tagSearch
  IsAlgorithmic: true
  CRUDActivities: []
  Properties: []
  RelatedResources: []
- !!Resource
  Name: tag
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [name, description]
  RelatedResources: [tagSearch]
- !!Resource
  Name: bookmark
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [url, scope]
  RelatedResources: [tag]

```

Σχήμα 3.16: Παράδειγμα Αρχείου YAML για το Έργο Restmarks

Μόλις παραχθεί η αναπαράσταση YAML του συστήματος, ο προγραμματιστής κατέχει πλέον πλήρεις και ανιχνεύσιμες προδιαγραφές που μπορούν να χρησιμοποιηθούν για διάφορους σκοπούς, συμπεριλαμβανομένων της επικύρωσης απαιτήσεων (requirements validation), της ανάπτυξης του συστήματος (system development) και της επαναχρησιμοποίησης

λογισμικού (software reuse). Η χρήση των μοντέλων μας για την εξόρυξη και επαναχρησιμοποίηση απαιτήσεων παρουσιάζεται στο επόμενο κεφάλαιο, ενώ για περισσότερες πληροφορίες σχετικά με τον μετασχηματισμό τους σε κώδικα (model-to-code transformation), ο ενδιαφερόμενος αναγνώστης παραπέμπεται στο [140].

### 3.5 Συμπεράσματα

Τον τελευταίο καιρό, το πρόβλημα της αυτοματοποίησης των διαδικασιών του καθορισμού απαιτήσεων και της εξαγωγής προδιαγραφών έχει απασχολήσει εκτεταμένα την ερευνητική κοινότητα. Ωστόσο, οι τρέχουσες προσεγγίσεις δεν υποστηρίζουν διαφορετικούς τύπους απαιτήσεων και συνήθως βασίζονται σε ευριστικούς κανόνες που αφορούν συγκεκριμένους τομείς (domain-specific heuristics) και/ή εξειδικευμένες γλώσσες. Στο παρόν κεφάλαιο, σχεδιάσαμε μια μεθοδολογία που επιτρέπει στους προγραμματιστές να μοντελοποιούν το σύστημα προς ανάπτυξη χρησιμοποιώντας απαιτήσεις λογισμικού από διάφορες πηγές. Η είσοδος του συστήματός μας δίνεται σε μορφή λειτουργικών απαιτήσεων και ροών ενεργειών με τη χρήση γραφικών σεναρίων, ούτως ώστε να καλύπτονται τόσο η στατική όψη (static view), όσο και η δυναμική όψη (dynamic view) του συστήματος. Η χρήση σημασιολογικών τεχνικών και τεχνικών επεξεργασίας φυσικής γλώσσας διευκολύνει την εξαγωγή των προδιαγραφών από τις απαιτήσεις, ενώ οι οντολογίες που σχεδιάστηκαν παράγουν ένα ανιχνεύσιμο μοντέλο.

Δείξαμε τα πλεονεκτήματα της προσέγγισής μας στο πεδίο της ανάπτυξης διαδικτυακών RESTful υπηρεσιών και συγκεκριμένα στο πώς μπορούν να παραχθούν εύκολα οι προδιαγραφές μιας υπηρεσίας. Το παραγόμενο μοντέλο (αρχείο YAML) περιλαμβάνει όλα τα απαιτούμενα στοιχεία της υπηρεσίας, συμπεριλαμβανομένων των πόρων, των ενεργειών CRUD και των ιδιοτήτων, ενώ υποστηρίζει και συνδέσεις hypermedia μέσω της καταγραφής των σχετικών πόρων. Η μελλοντική εργασία σχετικά με τη μεθοδολογία μας μπορεί να ακολουθήσει διάφορες κατευθύνσεις. Η συνεχής βελτίωση των εργαλείων με τη λήψη σχολίων από τους χρήστες βρίσκεται στα άμεσα σχέδια μας, ενώ η μελλοντική έρευνα που σχεδιάζουμε περιλαμβάνει περαιτέρω την αξιολόγηση της μεθοδολογίας μας σε βιομηχανικό περιβάλλον.



# 4

## Εξόρυξη Απαιτήσεων Λογισμικού

### 4.1 Επισκόπηση

Έχοντας προτείνει ένα μοντέλο για την αποθήκευση στατικής και δυναμικής πληροφορίας συστημάτων λογισμικού στο προηγούμενο κεφάλαιο, σε αυτό το κεφάλαιο επικεντρωνόμαστε στο πρόβλημα του καθορισμού και της επαναχρησιμοποίησης απαιτήσεων (requirements identification and reuse). Είναι ευρέως γνωστό ότι οι ανακρίβειες ή ελλείψεις απαιτήσεις είναι ο συνηθέστερος λόγος αποτυχίας για έργα λογισμικού [146]. Επιπλέον, οι συνεχείς αλλαγές στις αρχικές απαιτήσεις μπορούν να οδηγήσουν σε σφάλματα, ενώ το κόστος αναδιοργάνωσης λόγω των ανεπαρκώς καθορισμένων απαιτήσεων είναι αρκετά υψηλό [147]. Σε αυτό το πλαίσιο, ο καθορισμός των κατάλληλων απαιτήσεων για το εκάστοτε έργο λογισμικού προς ανάπτυξη είναι εξαιρετικά σημαντικό.

Σε αυτό το κεφάλαιο, αναλύουμε την πρόκληση της επαναχρησιμοποίησης απαιτήσεων λογισμικού. Όπως ήδη αναφέρθηκε, οι διάφορες πρωτοβουλίες λογισμικού ανοιχτού κώδικα (open-source software initiatives), καθώς και η βασισμένη σε τμήματα (component-based) φύση του λογισμικού έχει οδηγήσει σε έναν νέο τρόπο ανάπτυξης λογισμικού, που βασίζεται όλο και περισσότερο στην επαναχρησιμοποίηση τμημάτων, εντός ενός πλαισίου ταχείας παραγωγής πρωτότυπων εφαρμογών (rapid prototyping). Η ανάγκη για επαναχρησιμοποίηση υπαρχόντων τμημάτων έχει γίνει πιο ξεκάθαρη από ποτέ, καθώς η επαναχρησιμοποίηση τμημάτων μπορεί να μειώσει το χρόνο και την ανθρωποπροσπάθεια που δαπανάται σε όλες τις φάσεις του κύκλου ζωής του λογισμικού, συμπεριλαμβανομένων των φάσεων του καθορισμού απαιτήσεων (requirements elicitation), της εξαγωγής προδιαγραφών (specification extraction), της συγγραφής του κώδικα (source code writing) και της συντήρησης και του ελέγχου λογισμικού (software maintenance and testing). Ως εκ τούτου, υπάρχουν διάφορες προσεγγίσεις σχετικά με την εφαρμογή τεχνικών εξόρυξης δεδομένων για την πρόταση τμημάτων λογισμικού που καλύπτουν την απαιτούμενη λειτουργικότητα και ταυτόχρονα είναι υψηλής ποιότητας. Ωστόσο, οι περισσότερες από αυτές τις προσεγγίσεις επικεντρώνονται στον πηγαίο κώδικα των τμημάτων [95, 96, 101], σε πληροφορίες/μετρικές ποιότητας [148] και σε δεδομένα από online αποθετήρια [149, 150].

Τα οφέλη από την επαναχρησιμοποίηση των απαιτήσεων είναι εμφανή, ανεξάρτητα από τον τύπο του προϊόντος και τη μεθοδολογία ανάπτυξης λογισμικού που χρησιμοποιείται. Το λογισμικό κατασκευάζεται με βάση τις απαιτήσεις που καθορίζονται, ανεξάρτητα από το εάν καθορίζονται όλες μαζί, με επαναληπτικό τρόπο ή με δημιουργία πρωτότυπων (throw-away prototyping). Επομένως, οι βασικές προκλήσεις είναι η *σχεδίαση ενός μοντέλου ικανού να αποθηκεύει απαιτήσεις λογισμικού και η κατασκευή μιας μεθοδολογίας που θα επιτρέπει την επαναχρησιμοποίηση απαιτήσεων*. Το μοντέλο που προτείνουμε είναι αυτό που σχεδιάσαμε στο προηγούμενο κεφάλαιο, το οποίο επιτρέπει την απρόσκοπτη καταγραφή απαιτήσεων ή ακόμα και τη μεταφορά υφιστάμενων απαιτήσεων και υποστηρίζει την επαναχρησιμοποίηση ανεξάρτητα από τη μεθοδολογία ανάπτυξης λογισμικού που εφαρμόζεται. Η μεθοδολογία εξόρυξης είναι μια άλλη σημαντική πρόκληση η οποία προφανώς εξαρτάται σε μεγάλο βαθμό από το μοντέλο που επιλέγεται.

Όσον αφορά τον έλεγχο της ποιότητας των απαιτήσεων, οι σύγχρονες ερευνητικές προσπάθειες περιλαμβάνουν την αποθήκευση των απαιτήσεων λογισμικού σε σαφώς καθορισμένα μοντέλα [117–119, 121], που επιτρέπουν στους μηχανικούς απαιτήσεων (requirements engineers) να έχουν τον πλήρη έλεγχο της σχεδίασης του συστήματος και να εντοπίζουν τα σφάλματα σε πρώιμο στάδιο του κύκλου ζωής του έργου, κάτι που είναι συνήθως πολύ πιο αποτελεσματικό από την εύρεση και διόρθωση σφαλμάτων σε μεταγενέστερο στάδιο [116]. Όσον αφορά τον καθορισμό λειτουργικών απαιτήσεων, οι περισσότερες προσεγγίσεις περιλαμβάνουν μοντέλα που αφορούν συγκεκριμένους τομείς (domain-specific). Τα μοντέλα αυτά μπορούν στη συνέχεια να χρησιμοποιηθούν για να την επικύρωση απαιτήσεων (requirements validation) [82, 83] ή την πρόταση νέων απαιτήσεων (requirements reommendation) [84, 85] εντοπίζοντας οντότητες και σχέσεις που μπορεί να λείπουν [86]. Άλλες ερευνητικές κατευθύνσεις περιλαμβάνουν τον προσδιορισμό των εξαρτήσεων μεταξύ των απαιτήσεων και το διαχωρισμό τους ανάλογα με το για ποιους ενδιαφερόμενους έχουν σημασία (stakeholder preference) [87], καθώς και την ανάκτηση των συνδέσεων (recovering traceability links) μεταξύ των απαιτήσεων και των αντικειμένων λογισμικού και τη χρήση τους για τον εντοπισμό απαιτήσεων που ενδέχεται να υφίστανται αλλαγές [88]. Παρόλο που πολλές από τις παραπάνω προσεγγίσεις μπορεί να είναι αποτελεσματικές, οι περισσότερες από αυτές περιορίζονται σε συγκεκριμένους τομείς, κυρίως λόγω της έλλειψης σχολιασμένων απαιτήσεων για διάφορους τομείς.

Οι τεχνικές εξόρυξης διαγραμμάτων/μοντέλων UML αντιμετωπίζουν παρόμοια ζητήματα με αυτά των τεχνικών εξόρυξης λειτουργικών απαιτήσεων. Οι περισσότερες μέθοδοι που βασίζονται στη σημασιολογία (semantics-enabled methods) [151, 152] απαιτούν πληροφορίες από συγκεκριμένους τομείς (είναι δηλαδή domain-specific). Από την άλλη πλευρά, οι τεχνικές που δεν περιορίζονται στο λεξιλόγιο κάποιου τομέα (είναι δηλαδή domain-agnostic) έχουν σημαντικά περιθώρια βελτίωσης αν γίνει κατάλληλος χειρισμός των δεδομένων από τα μοντέλα απαιτήσεων. Οι αδυναμίες τους συνήθως έγκεινται στην κατάλληλη μοντελοποίηση των απαιτήσεων (που συνήθως γίνεται με γράφους [153–156] ή με διατεταγμένα δένδρα - ordered trees [157–159]), και συγκεκριμένα στην εξαγωγή πληροφοριών δομής (structural) ή ροής (flow), π.χ. για διαγράμματα σεναρίων χρήσης (use case diagrams) ή διαγράμματα δραστηριοτήτων (activity diagrams).

Σε αυτό το κεφάλαιο, χρησιμοποιούμε τις οντολογίες που σχεδιάσαμε στο προηγούμενο κεφάλαιο για την αποθήκευση λειτουργικών απαιτήσεων και διαγραμμάτων UML και αναπτύσσουμε μια μεθοδολογία εξόρυξης με σκοπό την επαναχρησιμοποίηση απαιτήσεων.

Εφαρμόζουμε τεχνικές εξόρυξης δεδομένων σε δύο άξονες επαναχρησιμοποίησης: λειτουργικές απαιτήσεις σε φυσική γλώσσα και μοντέλα UML. Όσον αφορά τις λειτουργικές απαιτήσεις, εφαρμόζουμε τεχνικές εξόρυξης κανόνων συσχέτισης (association rule mining) και ευριστικούς κανόνες (heuristics) για να προσδιοριστεί εάν οι απαιτήσεις ενός έργου λογισμικού είναι πλήρεις, και για την πρόταση νέων απαιτήσεων (όπως συζητείται στο [160]). Όσον αφορά τα μοντέλα UML, εφαρμόζουμε τεχνικές αντιστοίχισης (matching techniques) για την εύρεση παρόμοιων διαγραμμάτων με σκοπό να μπορεί εύκολα ο μηχανικός απαιτήσεων να βελτιώσει την υπάρχουσα λειτουργικότητα και τη ροή δεδομένων/επιχειρηματική ροή (data flow/business flow) του έργου.

## 4.2 Βιβλιογραφία για την Εξόρυξη Απαιτήσεων

### 4.2.1 Βιβλιογραφία για την Εξόρυξη Λειτουργικών Απαιτήσεων

Οι πρώτες ερευνητικές προσπάθειες για συστήματα προτάσεων στην περιοχή του καθορισμού απαιτήσεων επικεντρώθηκαν στην ανάλυση συγκεκριμένων τομέων και έτσι χρησιμοποιούσαν γλωσσολογία (linguistics) για τον προσδιορισμό των πιθανών τομέων εφαρμογής ενός έργου και την αναγνώριση οντοτήτων και σχέσεων που μπορεί να λείπουν σε επίπεδο απαιτήσεων. Ένα σχετικό παράδειγμα είναι το εργαλείο DARE [86] που χρησιμοποιεί πληροφορίες από τις απαιτήσεις, την αρχιτεκτονική και τον πηγαίο κώδικα ενός έργου. Το εργαλείο εξάγει οντότητες (entities) και σχέσεις (relations) από διάφορα έργα λογισμικού και στη συνέχεια εφαρμόζει ομαδοποίηση (clustering) για τον εντοπισμό κοινών οντοτήτων, και επομένως την πρόταση παρόμοιων αντικείμενων και αρχιτεκτονικών για κάθε έργο. Από την άλλη πλευρά, ο Kumar και οι συνεργάτες του [82] χρησιμοποιούν οντολογίες για την αποθήκευση απαιτήσεων και τομέων εφαρμογής έργων (project domains) με σκοπό την εξαγωγή προδιαγραφών λογισμικού. Οι Ghaisas και Ajmeri [83] αναπτύσσουν περαιτέρω ένα αποθετήριο γνώσης βασισμένο σε οντολογίες, που ονομάζεται K-RE (Knowledge-Assisted Ontology-Based Requirements Evolution repository), και το οποίο χρησιμοποιούν για τον καθορισμό απαιτήσεων λογισμικού και για την επίλυση πιθανών συγκρούσεων (conflict resolution) μεταξύ των απαιτήσεων.

Υπάρχουν, επίσης, διάφορες προσεγγίσεις που επιδιώκουν να προσδιορίσουν τα χαρακτηριστικά (features) ενός συστήματος μέσω των απαιτήσεών του και να προτείνουν νέα χαρακτηριστικά. Ο Chen και οι συνεργάτες του [84] χρησιμοποίησαν απαιτήσεις από διάφορα έργα και δημιούργησαν γράφους σχέσεων (relationship graphs) μεταξύ των απαιτήσεων. Στη συνέχεια, χρησιμοποίησαν τεχνικές ομαδοποίησης για την εξαγωγή πληροφοριών για τους τομείς εφαρμογής (domain information). Το σύστημά τους μπορεί να αναγνωρίσει χαρακτηριστικά (όπως π.χ. η αποθήκευση δεδομένων σε ένα αρχείο) τα οποία μπορούν να χρησιμοποιηθούν για τη δημιουργία ενός μοντέλου χαρακτηριστικών έργων που ανήκουν στον ίδιο τομέα εφαρμογής. Παρόμοια είναι και η προσέγγιση του Alves και των συνεργατών του [85], που χρησιμοποίησαν το μοντέλο διανυσματικού χώρου (vector space model) για την κατασκευή ενός μοντέλου χαρακτηριστικών τομέα εφαρμογής και εφάρμοσαν λανθάνουσα σημασιολογική ανάλυση (latent semantic analysis) για την εύρεση παρόμοιων απαιτήσεων ομαδοποιώντας τις ανάλογα με τους τομείς εφαρμογής τους. Ο Dumitru και οι συνεργάτες του [161] εφάρμοσαν τεχνικές εξόρυξης κανόνων συσχέτισης (association rule mining) για την ανάλυση απαιτήσεων, τις οποίες στη συνέχεια ομαδοποίησαν σε ομάδες χαρακτηρι-

κών (feature groups). Οι ομάδες αυτές μπορούν να χρησιμοποιηθούν για την εύρεση έργων που είναι παρόμοια ή για την πρόταση νέων χαρακτηριστικών για ένα έργο.

Τέλος, υπάρχουν επίσης προσεγγίσεις που διερευνούν τη σχέση μεταξύ απαιτήσεων και ενδιαφερομένων μερών (stakeholders) [162–164]. Στα σχετικά συστήματα η είσοδος δίνεται με τη μορφή προτιμήσεων για τις απαιτήσεις από κάθε μεμονωμένο ενδιαφερόμενο, και στη συνέχεια εφαρμόζονται τεχνικές collaborative filtering [165] για την παροχή προτάσεων και την ιεράρχηση (prioritization) των απαιτήσεων σύμφωνα με τις προτιμήσεις των ενδιαφερομένων. Ωστόσο, οι προσεγγίσεις αυτές, όπως επίσης και οι προσεγγίσεις που επικεντρώνονται σε μη λειτουργικές απαιτήσεις [166], αποκλίνουν από το πλαίσιο αυτής της διατριβής.

Με βάση την παραπάνω συζήτηση, παρατηρούμε ότι οι περισσότερες προσεγγίσεις είναι επικεντρωμένες σε συγκεκριμένους τομείς [82–86], κάτι που είναι γενικώς αναμενόμενο καθώς η χρήση πληροφοριών για κάποιο συγκεκριμένο τομέα εφαρμογής μπορούν να ενισχύσουν την κατανόηση του υπό μελέτη λογισμικού. Από την άλλη πλευρά, όπως επισημαίνεται από τον Dumitru και τους συνεργάτες του [161], οι τεχνικές αυτές είναι συχνά μη εφαρμόσιμες λόγω της έλλειψης σχολιασμένων απαιτήσεων για έναν συγκεκριμένο τομέα. Οι προσεγγίσεις που επιδιώκουν να προσδιορίσουν τα χαρακτηριστικά ενός συστήματος (feature-based) [161, 166] δεν αντιμετωπίζουν τα ίδια ζητήματα: ωστόσο, το πεδίο εφαρμογής τους είναι διαφορετικό, καθώς εφαρμόζονται σε επίπεδο χαρακτηριστικών και όχι λεπτομερών απαιτήσεων. Τέλος, το πεδίο εφαρμογής των τεχνικών που αφορούν τα ενδιαφερόμενα μέρη [162–164] αφορά κατά κύριο λόγο στη διαδικασία του πώς καθορίζονται οι απαιτήσεις, και όχι του ποιες είναι οι κατάλληλες απαιτήσεις για το έργο.

Σε αυτό το κεφάλαιο, κατασκευάζουμε ένα σύστημα που επικεντρώνεται στις λειτουργικές απαιτήσεις και παρέχει προτάσεις σε επίπεδο απαιτήσεων, χωρίς να περιορίζεται σε κάποιον τομέα εφαρμογής (καθώς είναι domain-agnostic). Αρχικά, χρησιμοποιούμε τη στατική οντολογία και τον σημασιολογικό αναλυτή του προηγούμενου κεφαλαίου για να εξάγουμε και να αποθηκεύσουμε τους δράστες (actors), τις ενέργειες (actions), τα αντικείμενα (objects) και τις ιδιότητες (properties) από λειτουργικές απαιτήσεις. Στη συνέχεια, το μοντέλο μας επιτρέπει την εξαγωγή προτάσεων απαιτήσεων με περιεκτικό τρόπο χωρίς κάποιο περιορισμό ως προς τον τομέα εφαρμογής.

#### 4.2.2 Βιβλιογραφία για την Εξόρυξη Μοντέλων UML

Οι πρώτες ερευνητικές προσπάθειες για εξόρυξη μοντέλων UML χρησιμοποιούσαν τεχνικές Ανάκτησης Πληροφοριών (Information Retrieval) για την εξόρυξη σεναρίων χρήσης (use case scenarios) [167, 168]. Σύμφωνα με αυτές, τα σενάρια χρήσης μπορούν να οριστούν ως σύνολα από *γεγονότα* (events) που λαμβάνουν μέρος λόγω των *ενεργειών* (actions) κάποιων *δραστήων* (actors), ενώ επίσης τα σενάρια περιλαμβάνουν *δημιουργούς* (authors) και *στόχους* (goals). Έτσι, η έρευνα σχετικά με την επαναχρησιμοποίηση σεναρίων αφορά κυρίως την αναπαράσταση ροών συμβάντων (event flows) και τη σύγκρισή τους για την εύρεση παρόμοιων σεναρίων χρήσης [167]. Μια άλλη κατηγορία προσεγγίσεων περιλαμβάνει τις προσεγγίσεις που αναπαριστούν τα διαγράμματα UML ως γράφους (graphs) και στη συνέχεια εφαρμόζουν τεχνικές αντιστοίχισης γράφων (graph matching) για την εξεύρεση παρόμοιων γράφων. Σε τέτοιου τύπου αναπαραστάσεις γράφων, οι κορυφές (vertices) αναφέρονται στα αντικείμενα των UML διαγραμμάτων σεναρίων χρήσης (use case diagrams),

των διαγραμμάτων κλάσεων (class diagrams) και των διαγραμμάτων ακολουθιών (sequence diagrams), ενώ οι ακμές (edges) υποδηλώνουν τις συσχετισμούς μεταξύ αυτών των αντικειμένων. Οι τεχνικές αντιστοίχισης γράφων που χρησιμοποιούνται μπορούν να είναι ακριβείς [153–155] ή προσεγγιστικές [156]. Μια παρόμοια προσέγγιση από τους Park και Bae [169] περιλαμβάνει τη χρήση Message Object Order Graphs (MOOGs) για την αποθήκευση διαγραμμάτων ακολουθιών, όπου οι κόμβοι και οι ακμές αντιπροσωπεύουν τα μηνύματα και τη ροή μεταξύ τους, αντίστοιχα.

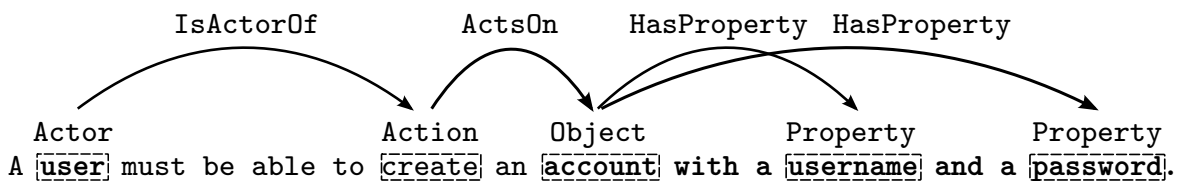
Παρά την αποτελεσματικότητα των μεθόδων που βασίζονται σε γράφους σε ορισμένα σενάρια (π.χ. για δομημένα μοντέλα), η εφαρμογή τους στα διαγράμματα UML στερείται σημασιολογίας. Όπως σημειώνεται από τον Kelter και τους συνεργάτες του [157], οι προσεγγίσεις εξόρυξης μοντέλων UML πρέπει να επικεντρωθούν στη δημιουργία ενός σημασιολογικού μοντέλου, αντί να εφαρμόζουν αυθαίρετα μετρικές ομοιότητας. Για το λόγο αυτό, τα διαγράμματα UML (και οι δομές XML) συχνά αναπαρίστανται ως διατεταγμένα δένδρα (ordered trees) [157–159]. Η βασική λογική των σχετικών προσεγγίσεων είναι να σχεδιάζεται πρώτα ένα μοντέλο δεδομένων (data model) σύμφωνα με τη δομή των διαγραμμάτων UML και στη συνέχεια να εφαρμόζονται μετρικές ομοιότητας δένδρων (tree similarity metrics). Οι προσεγγίσεις αυτές είναι αρκετά αποτελεσματικές για διαγράμματα στατικού τύπου (σεναρίων χρήσης, κλάσεων), ωστόσο δεν μπορούν να αναπαραστήσουν αποτελεσματικά τις ροές δεδομένων (data flows) ή τις ροές δράσεων (action flows), έτσι δεν αποτελούν τη βέλτιστη επιλογή για διαγράμματα δυναμικού τύπου (δραστηριοτήτων, ακολουθιών). Επιπλέον, συνήθως χρησιμοποιούν τεχνικές ομοιότητας συμβολοσειρών (string similarity), επομένως δεν είναι αποτελεσματικές σε περιπτώσεις όπου τα διαγράμματα προέρχονται από διαφορετικές πηγές. Για να αντιμετωπιστεί αυτή η αδυναμία, οι ερευνητικές προσπάθειες εστίασαν στην ενσωμάτωση σημασιολογίας (semantics) μέσω της χρήσης οντολογιών συγκεκριμένων τομέων (domain-specific ontologies) [151, 152, 170]. Η αντιστοίχιση των αντικειμένων των διαγραμμάτων (δραστών, σεναρίων χρήσης κ.λπ.) σύμφωνα με τη σημασιολογική τους ομοιότητα έχει πράγματι αποδειχθεί αποτελεσματική, εφόσον ο τομέας της εφαρμογής είναι καλώς ορισμένος, δηλαδή ορίζεται με επαρκείς πληροφορίες· τις περισσότερες φορές, ωστόσο, οι πληροφορίες για συγκεκριμένους τομείς είναι περιορισμένες.

Σε αυτό το κεφάλαιο, εστιάζουμε στη φάση του καθορισμού των απαιτήσεων και προτείνουμε δύο μεθοδολογίες εξόρυξης, μία για διαγράμματα σεναρίων χρήσης (use case diagrams) και μία για τα διαγράμματα δραστηριοτήτων (activity diagrams). Για την αντιστοίχιση διαγραμμάτων σεναρίων χρήσης, αναπαριστούμε τα σενάρια χρήσης ως διακριτούς κόμβους και χρησιμοποιούμε μετρικές σημασιολογική ομοιότητας (semantic similarity metrics), συνδυάζοντας έτσι τα πλεονεκτήματα των τεχνικών που βασίζονται σε γράφους με αυτά των τεχνικών ανάκτησης πληροφοριών και ταυτόχρονα αποφεύγοντας τους ενδεχόμενους περιορισμούς τους. Για την αντιστοίχιση διαγραμμάτων δραστηριοτήτων, κατασκευάζουμε ένα μοντέλο που αναπαριστά τα διαγράμματα δραστηριοτήτων ως ακολουθίες από ροές ενεργειών (sequences of action flows). Με αυτόν τον τρόπο μοντελοποιείται αποτελεσματικά ο δυναμικός χαρακτήρας των διαγραμμάτων. Ο αλγόριθμός μας είναι παρόμοιος με τις μεθόδους διατεταγμένων δένδρων [157–159], οι οποίες στην πραγματικότητα αποτελούν τη χρυσή τομή μεταξύ των μη δομημένων μεθόδων (π.χ. μεθόδων ανάκτησης πληροφοριών) και των υπερ-δομημένων μεθόδων (π.χ. μεθόδων βασισμένων σε γράφους). Οι μεθοδολογίες μας χρησιμοποιούν επίσης σημασιολογικές μετρικές για τη σύγκριση συμβολοσειρών, οι οποίες

ωστόσο δεν περιορίζονται σε συγκεκριμένους τομείς (είναι δηλαδή domain-agnostic), και επομένως μπορούν να χρησιμοποιηθούν σε περιπτώσεις όπου τα διαγράμματα προέρχονται από διάφορα έργα λογισμικού.

### 4.3 Εξόρυξη Λειτουργικών Απαιτήσεων

Όπως ήδη αναφέρθηκε, το μοντέλο της μεθοδολογίας εξόρυξης λειτουργικών απαιτήσεων περιλαμβάνει τη στατική οντολογία (και προφανώς όλα τα υποστηρικτικά εργαλεία, όπως το σημασιολογικό αναλυτή κ.λπ.). Έτσι, για την εξόρυξη υποθέτουμε ότι ο μηχανικός απαιτήσεων έχει ήδη δημιουργήσει μοντέλα για διάφορα έργα χρησιμοποιώντας τα εργαλεία μας. Για τη σχεδίαση και την αξιολόγηση του μοντέλου εξόρυξης απαιτήσεών μας, κατασκευάσαμε ένα σύνολο δεδομένων από απαιτήσεις 30 έργων λογισμικού, τα οποία περιλαμβάνουν απαιτήσεις από έγγραφα που έχουν γραφεί από φοιτητές, απαιτήσεις από πραγματικά έργα λογισμικού, καθώς και απαιτήσεις RESTful υπηρεσιών. Συνολικά, τα έργα αυτά έχουν 514 λειτουργικές απαιτήσεις, ενώ μετά τη μοντελοποίηση προκύπτουν 7234 έννοιες και 6626 σχέσεις μεταξύ εννοιών. Ένα παράδειγμα απαίτησης με σχολιασμούς (annotations) φαίνεται στο Σχήμα 4.1.



Σχήμα 4.1: Παράδειγμα απαίτησης με σχολιασμούς

Το σύστημά μας μπορεί να συσχετίσει όρους μεταξύ διαφορετικών έργων. Αρχικά, εξάγονται οι έννοιες και οι σχέσεις για κάθε απαίτηση (και συνεπώς για κάθε έργο). Για παράδειγμα, για την απαίτηση του Σχήματος 4.1, εξάγονται οι όροι `user`, `create`, `account`, `username` και `password`, καθώς και οι αντίστοιχες σχέσεις μεταξύ τους: `user_IsActorOf_create`, `create_ActsOn_account`, `account_HasProperty_username` και `account_HasProperty_password`. Η συσχέτιση δύο απαιτήσεων (η γενικά δύο έργων) απαιτεί τη σημασιολογική συσχέτιση των όρων τους.

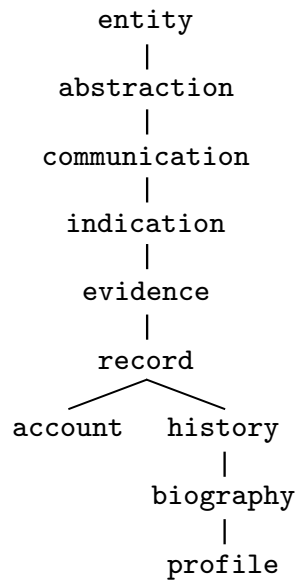
Για παράδειγμα, εάν έχουμε ένα άλλο έργο όπου κάθε χρήστης (`user`) έχει επίσης ένα λογαριασμό, ωστόσο ο όρος για το λογαριασμό χρήστη είναι `profile` και όχι `account`, τότε αυτοί οι όροι `profile` και `account` θα πρέπει να σημειωθούν ως σημασιολογικά όμοιοι. Για να επισημάνουμε τους σημασιολογικά όμοιους όρους, χρειαζόμαστε ένα ευρετήριο (index) όρων και ένα μέτρο ομοιότητας (similarity measure) μεταξύ τους. Χρησιμοποιούμε το WordNet [171] ως το ευρετήριό μας, και συνδεόμαστε με αυτό μέσω του MIT Java Wordnet Interface [172] και της βιβλιοθήκης `Java Wordnet::Similarity`<sup>1</sup>. Υπάρχουν διάφορες μέθοδοι για τον υπολογισμό της ομοιότητας μεταξύ δύο όρων [173]. Οι περισσότεροι, ωστόσο, είτε δεν χρησιμοποιούν σημασιολογία είτε δεν συμφωνούν με την ανθρώπινη κρίση. Ως εκ τούτου, χρησιμοποιούμε τη μετρική του περιεχομένου πληροφορίας

<sup>1</sup><http://users.sussex.ac.uk/~drh21/>

(information content) που προτάθηκε από τον Lin [174], η οποία συμφωνεί με την ανθρώπινη κρίση πιο συχνά από άλλες μετρικές. Ορίζουμε την ομοιότητα μεταξύ δύο όρων (δηλαδή κλάσεων του WordNet)  $C_1$  και  $C_2$  ως:

$$\text{sim}(C_1, C_2) = \frac{2 \cdot \log P(C_0)}{\log P(C_1) + \log P(C_2)} \quad (4.1)$$

όπου  $C_0$  είναι η πιο ειδική (*specific*) κλάση που περιέχει το  $C_1$  και το  $C_2$ . Για παράδειγμα, η πιο ειδική κλάση που περιγράφει τους όρους **account** και **profile** είναι η κλάση **record**, όπως φαίνεται και στο σχετικό τμήμα του WordNet στο Σχήμα 4.2.



Σχήμα 4.2: Παράδειγμα τμήματος του WordNet όπου η κλάση **record** είναι η πιο ειδική κλάση των **account** και **profile**

Έχοντας βρει το  $C_0$ , υπολογίζουμε την ομοιότητα μεταξύ των δύο όρων χρησιμοποιώντας την εξίσωση (4.1), που χρειάζεται την τιμή του *περιεχομένου πληροφορίας* (*information content*) για κάθε μία από τις τρεις κλάσεις του WordNet. Το περιεχόμενο πληροφορίας μιας κλάσης WordNet ορίζεται ως η λογαριθμική πιθανότητα ότι ένας όρος του corpus ανήκει σε αυτήν την κλάση<sup>2</sup>. Έτσι, για παράδειγμα, οι κλάσεις **record**, **account** και **profile** έχουν τιμές περιεχομένου πληροφορίας ίσες με 7.874, 7.874 και 11.766 αντίστοιχα, οπότε η ομοιότητα μεταξύ των όρων **account** και **profile** είναι  $2 \cdot 7.874 / (7.874 + 11.766) = 0.802$ . Τέλος, δύο όροι θεωρείται ότι είναι σημασιολογικά όμοιοι εάν η τιμή ομοιότητας μεταξύ τους (όπως καθορίζεται από την εξίσωση (4.1)) είναι μεγαλύτερη από ένα όριο  $t$ . Έτσι, εξασφαλίζεται ότι η σημασιολογία των όρων λαμβάνεται υπόψη, τόσο κατά την εκπαίδευση του συστήματος όσο και για τη χρήση του για προτάσεις απαιτήσεων. Ορίζουμε αυτό το όριο στο 0.5, και έτσι έχουμε τώρα πλέον 1512 όρους από τα 30 έργα λογισμικού, εκ των οποίων οι 1162 είναι διακριτοί.

Αφού ολοκληρωθεί η παραπάνω ανάλυση, τα δεδομένα μας πλέον είναι ένα σύνολο από *στοιχεία* (*items*) για κάθε έργο λογισμικού, από τα οποία μπορούμε να εξάγουμε χρήσιμους

<sup>2</sup>Χρησιμοποιήσαμε τις προϋπολογισμένες τιμές περιεχομένου πληροφορίας της βιβλιοθήκης Perl WordNet similarity [173], που είναι διαθέσιμη στο <http://www.d.umn.edu/~tpederse/>.

κανόνες συσχέτισης (*association rules*) χρησιμοποιώντας τεχνικές εξόρυξης κανόνων συσχέτισης (*association rule mining*) [175]. Έστω  $P = \{p_1, p_2, \dots, p_m\}$  το σύνολο των  $m$  έργων λογισμικού και  $I = \{i_1, i_2, \dots, i_n\}$  το σύνολο όλων των  $n$  στοιχείων. Τα *στοιχειοσύνολα* (*itemsets*) ορίζονται ως υποσύνολα του  $I$ . Για ένα στοιχειοσύνολο  $X$ , η υποστήριξη (*support*) του ορίζεται ως το πλήθος των έργων στα οποία εμφανίζονται όλα τα στοιχεία του:

$$\sigma(X) = |\{p_i | X \subset p_i, p_i \in P\}| \quad (4.2)$$

Οι κανόνες συσχέτισης εκφράζονται στη μορφή  $X \rightarrow Y$ , όπου τα  $X$  και  $Y$  είναι στοιχειοσύνολα που είναι ξένα μεταξύ τους (*disjoint itemsets*). Ένα παράδειγμα κανόνα που μπορεί να εξαχθεί από τα στοιχεία της απαίτησης του Σχήματος 4.1 είναι ο κανόνας  $\{\text{account\_HasProperty\_username}\} \rightarrow \{\text{account\_HasProperty\_password}\}$ .

Οι δύο μετρικές που χρησιμοποιούνται για να προσδιοριστεί η ισχύς ενός κανόνα είναι η *υποστήριξη* (*support*) και η *εμπιστοσύνη* (*confidence*). Για έναν κανόνα συσχέτισης  $X \rightarrow Y$ , η υποστήριξη (*support*) είναι το πλήθος των έργων λογισμικού για τα οποία ο κανόνας ισχύει (ή, όπως λέμε, ενεργοποιείται), και υπολογίζεται ως:

$$\sigma(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{|P|} \quad (4.3)$$

Η εμπιστοσύνη (*confidence*) του κανόνα υποδεικνύει πόσο συχνά τα στοιχεία του  $Y$  εμφανίζονται στο  $X$ , και υπολογίζεται ως:

$$c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \quad (4.4)$$

Χρησιμοποιούμε τον αλγόριθμο Apriori [176] για να εξάγουμε κανόνες συσχέτισης με υποστήριξη και εμπιστοσύνη πάνω από συγκεκριμένα όρια. Θέτουμε την ελάχιστη υποστήριξη (*minimum support*) σε 0.1, ούτως ώστε κάθε κανόνας να πρέπει να περιέχεται τουλάχιστον στο 10% των έργων. Επίσης θέτουμε την ελάχιστη εμπιστοσύνη (*minimum confidence*) σε 0.5, έτσι ώστε οι κανόνες που εξάγονται να ισχύουν τουλάχιστον τις μισές φορές που εμφανίζεται το πρότερο μέρος τους (*antecedent*). Η εκτέλεση του Apriori είχε ως αποτέλεσμα την εξαγωγή 1372 κανόνων, ένα τμήμα των οποίων φαίνεται στον Πίνακα 4.1.

Πίνακας 4.1: Δείγμα Κανόνων Συσχέτισης που Εξήχθησαν από το Σύνολο Δεδομένων

A/A	Κανόνας Συσχέτισης	$\sigma$	$c$
1	<i>provide_ActsOn_product</i> $\rightarrow$ <i>system_IsActorOf_provide</i>	0.167	1.0
2	<i>system_IsActorOf_validate</i> $\rightarrow$ <i>user_IsActorOf_login</i>	0.1	1.0
3	<i>user_IsActorOf_buy</i> $\rightarrow$ <i>system_IsActorOf_provide</i>	0.1	1.0
4	<i>administrator_IsActorOf_add</i> $\rightarrow$ <i>administrator_IsActorOf_delete</i>	0.167	0.833
5	<i>user_IsActorOf_logout</i> $\rightarrow$ <i>user_IsActorOf_login</i>	0.167	0.833
6	<i>user_IsActorOf_add</i> $\rightarrow$ <i>user_IsActorOf_delete</i>	0.133	0.8
7	<i>user_IsActorOf_access</i> $\rightarrow$ <i>user_IsActorOf_view</i>	0.1	0.75
8	<i>edit_ActsOn_product</i> $\rightarrow$ <i>add_ActsOn_product</i>	0.1	0.75
9	<i>administrator_IsActorOf_delete</i> $\rightarrow$ <i>administrator_IsActorOf_add</i>	0.167	0.714
10	<i>user_HasProperty_contact</i> $\rightarrow$ <i>user_IsActorOf_search</i>	0.133	0.5

$\sigma$ : Υποστήριξη (Support),  $c$ : Εμπιστοσύνη (Confidence)



Αρκετοί από αυτούς τους κανόνες είναι λογικοί. Για παράδειγμα, ο κανόνας 2 υποδεικνύει ότι για να μπορέσει κάποιος χρήστης να συνδεθεί, ο λογαριασμός του πρέπει πρώτα να επικυρωθεί από το σύστημα. Επίσης, ο κανόνας 5 υποδηλώνει ότι η δυνατότητα αποσύνδεσης ενός χρήστη από ένα σύστημα πρέπει να συνυπάρχει προφανώς με τη δυνατότητα σύνδεσης στο σύστημα.

Αφού εξάγουμε τους κανόνες, τους χρησιμοποιούμε για να προτείνουμε νέες απαιτήσεις σε έργα λογισμικού. Αρχικά, για κάθε έργο λογισμικού που προσθέτουμε ορίζουμε ένα νέο σύνολο στοιχείων  $p$ . Με βάση αυτό το σύνολο στοιχείων καθώς και τους κανόνες συσχέτισης, εξάγουμε ένα σύνολο ενεργοποιημένων (*activated*) κανόνων  $R$ . Ένας κανόνας  $X \rightarrow Y$  ενεργοποιείται για το έργο με το σύνολο στοιχείων  $p$  αν όλα τα στοιχεία του  $X$  περιέχονται επίσης στο  $p$  (δηλαδή αν ισχύει  $X \subset p$ ). Στη συνέχεια, το σύνολο των ενεργοποιημένων κανόνων  $R$  γίνεται επίπεδο (*flattened*) δημιουργώντας έναν νέο κανόνα για κάθε συνδυασμό των πρότερων στοιχείων (*antecedents*) και των επακόλουθων στοιχείων (*consequents*) ενός αρχικού κανόνα, έτσι ώστε στους νέους κανόνες κάθε πρότερο μέρος και κάθε επακόλουθο μέρος να αποτελείται από μόνο ένα στοιχείο. Για παράδειγμα, για τον κανόνα  $X \rightarrow Y$  όπου τα στοιχειοσύνολα  $X$  και  $Y$  περιέχουν τα στοιχεία  $\{i_1, i_2, i_3\}$  και  $\{i_4, i_5\}$  αντίστοιχα, οι νέοι επίπεδοι κανόνες είναι οι  $i_1 \rightarrow i_4, i_1 \rightarrow i_5, i_2 \rightarrow i_4, i_2 \rightarrow i_5, i_3 \rightarrow i_4$  και  $i_3 \rightarrow i_5$ . Η υποστήριξη (*support*) και η εμπιστοσύνη (*confidence*) των αρχικών κανόνων μεταφέρεται σε αυτούς τους νέους επίπεδους κανόνες, έτσι ώστε να χρησιμοποιηθούν ως κριτήρια σημαντικότητας. Τέλος, δεδομένου του συνόλου των στοιχείων ενός έργου  $p$  και των επίπεδων ενεργοποιημένων κανόνων για αυτό το έργο, το σύστημά μας παρέχει προτάσεις για νέες απαιτήσεις χρησιμοποιώντας τους ευριστικούς κανόνες (*heuristics*) του Πίνακα 4.2.

Πίνακας 4.2: Ενεργοποίηση Κανόνων για ένα Έργο Λογισμικού

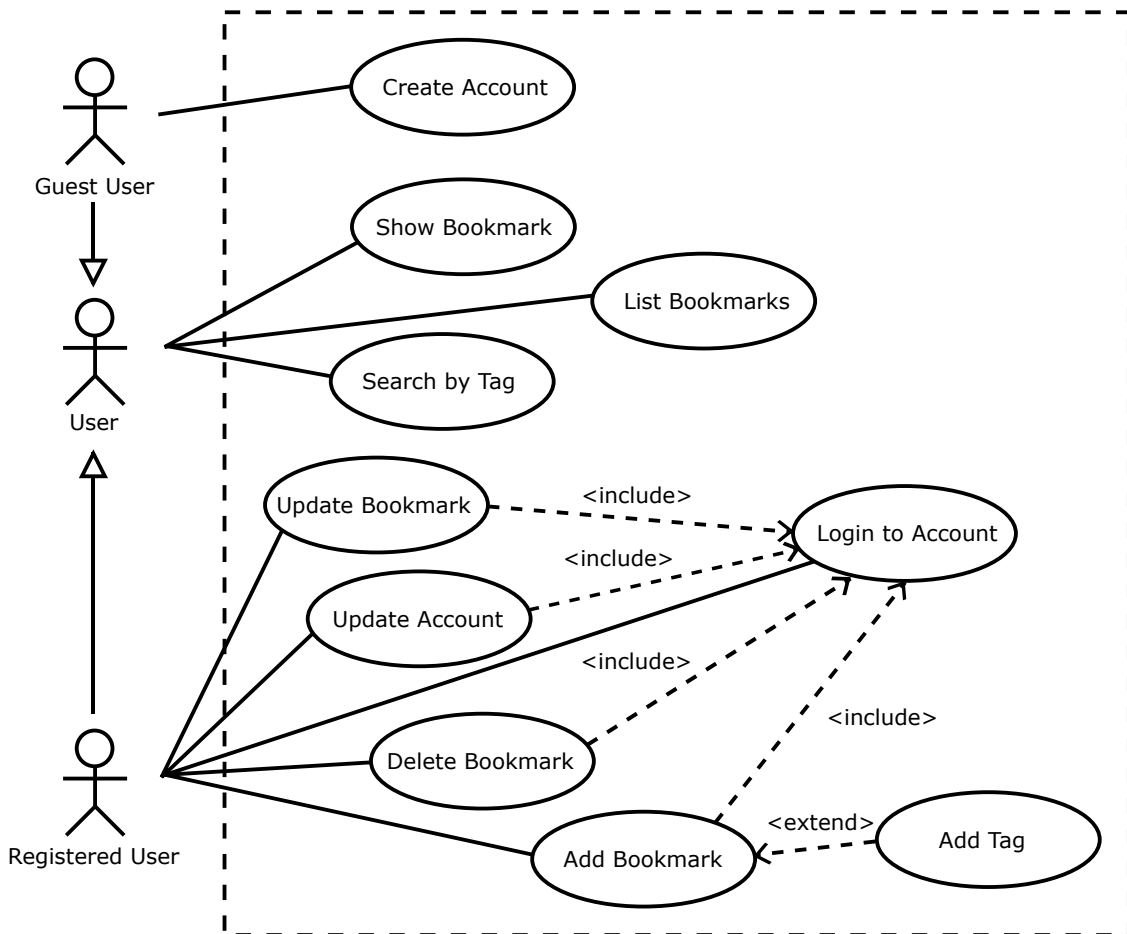
Πρότερο μέρος	Επακόλουθο μέρος	Συνθήκες	Αποτέλεσμα
$[Actor1, Action1]$	$[Actor2, Action2]$	$Actor2 \in p$	$[Actor2, Action2, Object],$ $\forall Object \in p :$ $[Actor1, Action1, Object]$
$[Action1, Object1]$	$[Actor2, Action2]$	$Actor2 \in p$	$[Actor2, Action2, Object1]$
$[Actor1, Action1]$	$[Action2, Object2]$	$Object2 \in p$	$[Actor1, Action2, Object2]$
$[Action1, Object1]$	$[Action2, Object2]$	$Object2 \in p$	$[Actor, Action2, Object2],$ $\forall Actor \in p :$ $[Actor, Action1, Object1]$
* (except for the above)	$[Action2, Object2]$	$Object2 \in p$	$[Actor, Action2, Object2],$ $\forall Actor, Action \in p :$ $[Actor, Action, Object2]$
*	$[Any2, Property2]$	$Any2 \in p$	$[Any2, Property2]$

Όσον αφορά το πρότερο μέρος (*consequent*)  $[Actor2, Action2]$ , που αντιστοιχεί σε ένα στοιχείο  $Actor2\_IsActorOf\_Action2$ , η προτεινόμενη απαίτηση περιλαμβάνει το δράστη (*actor*) και την ενέργεια (*action*) του επακόλουθου μέρους (*consequent*), καθώς επίσης και ένα αντικείμενο (*Object*) που προσδιορίζεται από το πρότερο μέρος. Για παράδειγμα, για το πρότερο μέρος  $[create, bookmark]$  και το επακόλουθο μέρος  $[user, edit]$ , η νέα προτεινόμενη απαίτηση θα δηλώνεται ως  $[user, edit, bookmark]$ . Όσον αφορά το πρότερο μέ-

ρος  $[Action2, Object2]$ , που αντιστοιχεί σε ένα στοιχείο  $Action2\_ActsOn\_Object2$ , η προτεινόμενη απαίτηση περιλαμβάνει την ενέργεια (action) και το αντικείμενο (object) του επακόλουθου μέρους καθώς επίσης και το δράστη (actor) που προσδιορίζεται από το πρότερο μέρος. Για παράδειγμα, για το πρότερο μέρος  $[user, profile]$  και το επακόλουθο μέρος  $[create, profile]$ , η νέα προτεινόμενη απαίτηση θα δηλώνεται ως  $[user, create, profile]$ . Τέλος, κάθε κανόνας που έχει το `HasProperty` στο πρότερο μέρος (και οποιοδήποτε επακόλουθο μέρος) παράγει προτάσεις απαιτήσεων της μορφής  $[Any, Property]$ . Ένα παράδειγμα τέτοιας απαίτησης δηλώνεται ως  $[user, profile]$ . Χρησιμοποιώντας τη στατική οντολογία, μπορούμε επιπλέον να αναδομήσουμε τις απαιτήσεις, χρησιμοποιώντας τα πρότυπα ‘The Actor must be able to Action Object.’ και ‘The Any must have Property.’.

## 4.4 Εξόρυξη Μοντέλων UML

Για να σχεδιάσουμε τη μεθοδολογία εξόρυξης διαγραμμάτων UML, χρησιμοποιούμε ένα σύνολο από διαγράμματα σεναρίων χρήσης (use case diagrams) και διαγράμματα δραστηριοτήτων (activity diagrams), στα οποία περιλαμβάνονται τα διαγράμματα του έργου Restmarks, της υπηρεσίας που παρουσιάστηκε στο προηγούμενο κεφάλαιο. Στο Σχήμα 4.3 παρουσιάζεται ένα παράδειγμα διαγράμματος σεναρίου χρήσης από το Restmarks που περιέχει 10 σενάρια χρήσης (use cases) και 3 δράστες (actors).



Σχήμα 4.3: Παράδειγμα διαγράμματος σεναρίων χρήσης για το έργο Restmarks

Καθώς τα σενάρια χρήσης αναφέρονται σε στατική πληροφορία, αποθηκεύονται στην στατική οντολογία με ένα μοντέλο που είναι ουσιαστικά επίπεδο. Συγκεκριμένα, το μοντέλο περιλαμβάνει τους δράστες και τα σενάρια χρήσης του διαγράμματος. Για παράδειγμα, το διάγραμμα  $D$  του Σχήματος 4.3 έχει ένα μοντέλο που αποτελείται από δύο σύνολα: το σύνολο των δραστών  $A = \{User, Registered\ User, Guest\ User\}$  και το σύνολο των σεναρίων χρήσης  $UC = \{Add\ Bookmark, Update\ Bookmark, Update\ Account, Show\ Bookmark, Search\ by\ Tag, Add\ Tag, Login\ to\ Account, Delete\ Bookmark\}$ . Για δύο διαγράμματα  $D_1$  και  $D_2$ , η μέθοδος αντιστοίχισης που σχεδιάσαμε περιέχει δύο σύνολα για κάθε διάγραμμα: ένα σύνολο για τους δράστες ( $A_1$  και  $A_2$  αντίστοιχα) και ένα σύνολο για τα σενάρια χρήσης ( $UC_1$  και  $UC_2$  αντίστοιχα). Η ομοιότητα μεταξύ δύο διαγραμμάτων υπολογίζεται από την παρακάτω εξίσωση:

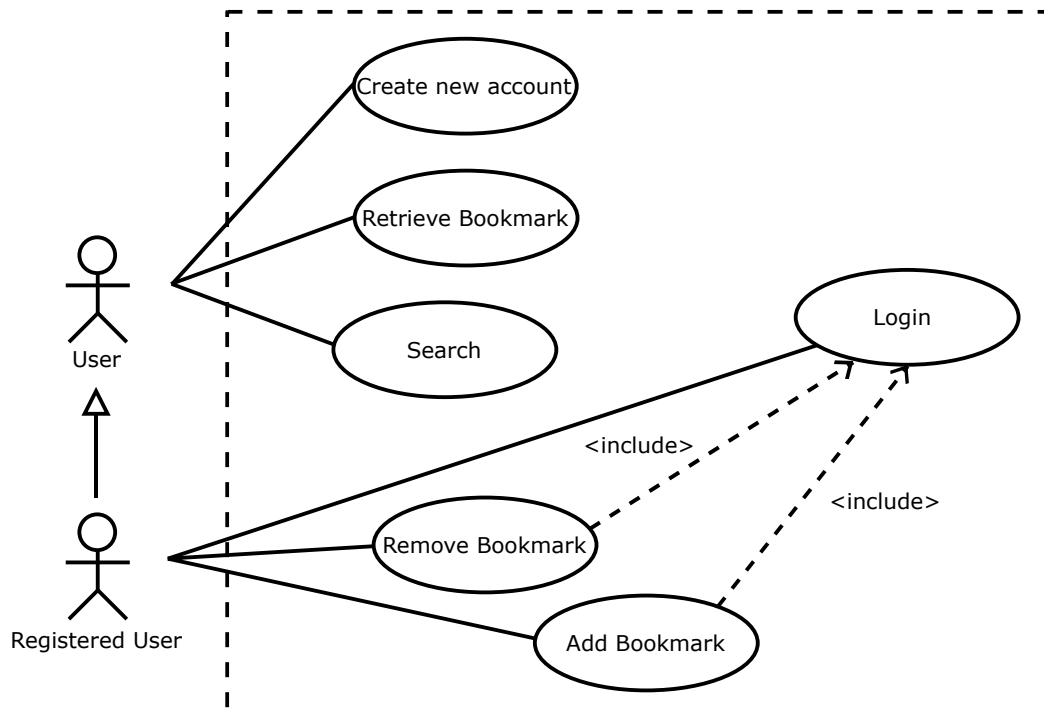
$$s(D_1, D_2) = \alpha \cdot s(A_1, A_2) + (1 - \alpha) \cdot s(UC_1, UC_2) \quad (4.5)$$

όπου το  $s$  υποδηλώνει την ομοιότητα μεταξύ δύο συνόλων (είτε των δραστών είτε των σεναρίων χρήσης) και η μεταβλητή  $\alpha$  υποδηλώνει τη σημασία της ομοιότητας των δραστών για τα διαγράμματα. Ορίζουμε το  $\alpha$  ως το ποσοστό του αριθμού των δραστών διαιρούμενο με τον αριθμό των σεναρίων χρήσης του διαγράμματος για το οποίο αναζητούμε παρόμοια. Για παράδειγμα για ένα διάγραμμα με 3 δράστες και 10 σενάρια χρήσης, το  $\alpha$  έχει την τιμή 0.3.

Η ομοιότητα μεταξύ δύο συνόλων, είτε των δραστών είτε των σεναρίων χρήσης, δίνεται από το συνδυασμό όλων των αντιστοιχισμένων στοιχείων με το υψηλότερο σκορ. Για παράδειγμα, για τα δύο σύνολα  $\{user, administrator, guest\}$  και  $\{administrator, user\}$ , ο καλύτερος δυνατός συνδυασμός είναι  $\{(user, user), (administrator, administrator), (guest, null)\}$ , και η αντιστοίχιση θα επέστρεφε τιμή ίση με  $2/3 = 0.66$ . Χρησιμοποιούμε τη μετρική σημασιολογικής απόστασης του προηγούμενου υποκεφαλαίου για να παρέχουμε μια τιμή για την ομοιότητα μεταξύ συμβολοσειρών (string similarity). Για δύο συμβολοσειρές  $S_1$  και  $S_2$ , αρχικά τις διαχωρίζουμε σε όρους (tokens), π.χ.  $tokens(S_1) = \{t_1, t_2\}$  και  $tokens(S_2) = \{t_3, t_4\}$ , και στη συνέχεια προσδιορίζουμε το συνδυασμό των όρων με τη μεγαλύτερη τιμή ομοιότητας. Η τελική τιμή της ομοιότητας μεταξύ των συμβολοσειρών καθορίζεται παίρνοντας το μέσο όρο των τιμών για όλους τους όρους. Για παράδειγμα, αν έχουμε τις συμβολοσειρές ‘Get bookmark’ and ‘Retrieve bookmarks’, ο καλύτερος συνδυασμός είναι (‘get’, ‘retrieve’) και (‘bookmark’, ‘bookmarks’). Εφόσον η σημασιολογική ομοιότητα μεταξύ των ‘get’ και ‘retrieve’ είναι 0.677 και η σημασιολογική ομοιότητα μεταξύ των ‘bookmark’ και ‘bookmarks’ είναι 1.0, η τελική ομοιότητα μεταξύ των συμβολοσειρών είναι  $(0.677 + 1)/2 = 0.8385$ .

Για παράδειγμα, έχοντας το διάγραμμα του Σχήματος 4.3 και το διάγραμμα του Σχήματος 4.4, η αντιστοίχιση μεταξύ των στοιχείων των διαγραμμάτων φαίνεται στον Πίνακα 4.3, ενώ η τελική τιμή για την ομοιότητά τους (με βάση την εξίσωση (4.5)) είναι 0.457.

Όσον αφορά τις προτάσεις, ο μηχανικός του δεύτερου διαγράμματος θα μπορούσε να εξετάσει την προσθήκη ενός χρήστη τύπου επισκέπτη (Guest User). Επιπλέον, θα μπορούσε να εξετάσει το ενδεχόμενο να προσθέσει σενάρια χρήσης για την επιστροφή λίστας σελιδοδεικτών (List Bookmarks) ή την ενημέρωση σελιδοδεικτών (Update Bookmark), την προσθήκη ετικετών στους σελιδοδείκτες (Add Tag) ή την ενημέρωση του λογαριασμού χρήστη (Update Account).



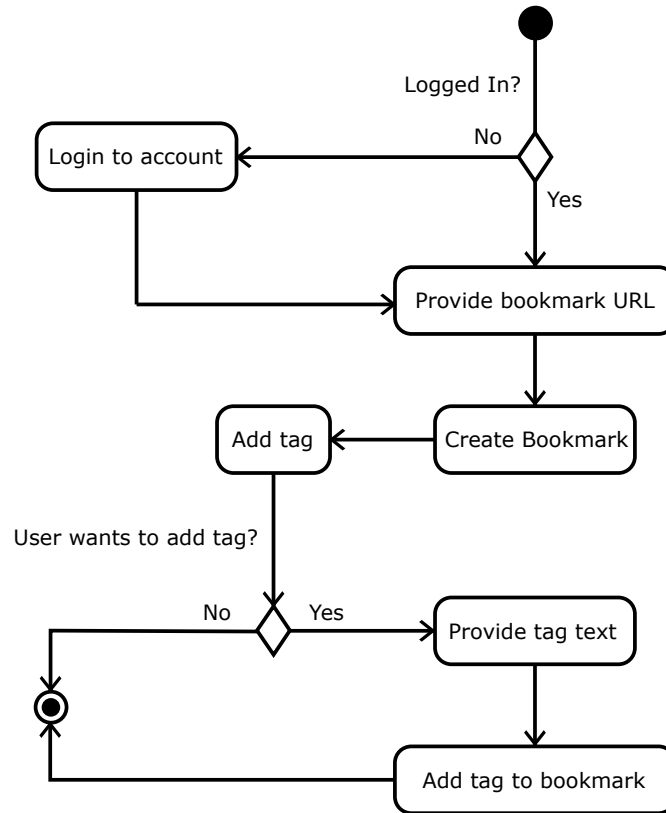
Σχήμα 4.4: Παράδειγμα διαγράμματος σεναρίων χρήσης για αντιστοίχιση με το διάγραμμα του Σχήματος 4.3.

Πίνακας 4.3: Αντιστοίχιση μεταξύ των Διαγραμμάτων των Σχημάτων 4.3 και 4.4.

Διάγραμμα 1	Διάγραμμα 2	Τιμή
User	User	1.00
Registered User	Registered User	1.00
Guest User	null	0.00
Delete Bookmark	Remove Bookmark	0.86
Show Bookmark	Retrieve Bookmark	0.50
Add Bookmark	Add Bookmark	1.00
Create Account	Create new account	0.66
Search by Tag	Search	0.33
Login to Account	Login	0.33
List Bookmarks	null	0.00
Update Bookmark	null	0.00
Update Account	null	0.00
Add Tag	null	0.00

Όσον αφορά τα διαγράμματα δραστηριοτήτων, χρειαζόμαστε μια αναπαράσταση που θα αποθηκεύει το διάγραμμα ως μοντέλο ροής. Στο Σχήμα 4.5 φαίνεται ένα παράδειγμα διαγράμματος του έργου Restmarks.

Τα διαγράμματα δραστηριοτήτων αποτελούνται κυρίως από ακολουθίες δραστηριοτήτων και συνθήκες. Όσον αφορά τις συνθήκες (και τις διακλαδώσεις - forks και τις συνδέσεις - joins), αυτές χωρίζουν τη ροή του διαγράμματος. Επομένως, ένα διάγραμμα δραστηριοτήτων αντιμετωπίζεται στην πραγματικότητα ως ένα σύνολο ακολουθιών, η κάθε μία από τις οποίες περιλαμβάνει τις δραστηριότητες που απαιτούνται για να διασχιστεί το διά-



Σχήμα 4.5: Παράδειγμα διαγράμματος δραστηριοτήτων για το έργο Restmarks

γραμμα από τον αρχικό κόμβο (start node) στον τελικό κόμβο (end node). Για παράδειγμα, για το διάγραμμα του Σχήματος 4.5 ορίζεται μια ακολουθία  $StartNode > Logged\ In? > Login\ to\ account > Provide\ bookmark\ URL > Create\ Bookmark > Add\ tag > User\ wants\ to\ add\ tag? > EndNode$ . Μετά την ανάλυση των δύο διαγραμμάτων και την εξαγωγή ενός συνόλου ακολουθιών για κάθε διάγραμμα, μπορούμε πλέον να συγκρίνουμε τα δύο σύνολα. Στην περίπτωση αυτή, και σε αντίθεση με την αντιστοίχιση διαγραμμάτων σεναρίων χρήσης, ορίζουμε ένα όριο  $t_{ACT}$  για τη μετρική σημασιολογικής ομοιότητας συμβολοσειρών (semantic string similarity). Επομένως, δύο συμβολοσειρές θεωρούνται όμοιες εάν η τιμή ομοιότητάς τους είναι μεγαλύτερη από αυτό το όριο. Ορίζουμε το  $t_{ACT}$  στο 0.5.

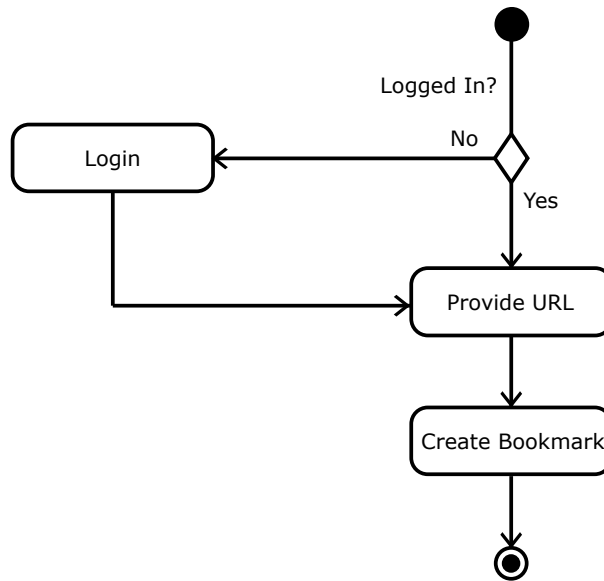
Για να προσδιορίσουμε την ομοιότητα μεταξύ δύο ακολουθιών χρησιμοποιούμε τη *Μέγιστη Κοινή Υπακολουθία (Longest Common Subsequence - LCS)* [177]. Η μέγιστη κοινή υπακολουθία μεταξύ δύο ακολουθιών  $S_1$  και  $S_2$  ορίζεται ως η μέγιστη υπακολουθία από τα κοινά (αλλά όχι απαραίτητα διαδοχικά) στοιχεία μεταξύ τους. Για παράδειγμα, η μέγιστη κοινή υπακολουθία για τις ακολουθίες  $[A, B, D, E, G]$  και  $[A, B, E, H]$  είναι η  $[A, B, E]$ . Τελικά, η τιμή ομοιότητας μεταξύ των δύο ακολουθιών ορίζεται ως:

$$sim(S_1, S_2) = 2 \cdot \frac{|LCS(S_1, S_2)|}{|S_1| + |S_2|} \quad (4.6)$$

Η τιμή ομοιότητας κανονικοποιείται στο διάστημα  $[0, 1]$ . Για δύο σύνολα ακολουθιών (ένα για κάθε ένα από τα δύο διαγράμματα), η ομοιότητά τους καθορίζεται παίρνοντας τον καλύτερο δυνατό συνδυασμό μεταξύ των ακολουθιών, δηλαδή τον συνδυασμό με τη μεγαλύτερη τιμή. Για παράδειγμα, για τα σύνολα  $\{[A, B, E], [A, B, D, E], [A, B, C, E]\}$  και  $\{[A, B, E]$ ,

$[A, C, E]$ , ο συνδυασμός που παράγει τη μεγαλύτερη τιμή ομοιότητας είναι ο  $\{([A, B, E], [A, B, E]), ([A, B, C, E], [A, C, E]), ([A, B, D, E], null)\}$ . Τελικά, η τιμή ομοιότητας μεταξύ των διαγραμμάτων είναι ο μέσος όρος των τιμών ομοιότητας των ακολουθιών τους.

Ως ένα παράδειγμα, θεωρούμε την αντιστοίχιση των διαγραμμάτων των Σχημάτων 4.5 και 4.6. Το σύστημά μας επιστρέφει την αντιστοίχιση μεταξύ των ακολουθιών των διαγραμμάτων, που φαίνεται στον Πίνακα 4.4, ενώ η τελική ομοιότητα, η οποία υπολογίζεται ως ο μέσος όρος των τιμών ομοιότητας των ακολουθιών, είναι  $(0.833+0.714+0+0)/4 = 0.387$ .



Σχήμα 4.6: Παράδειγμα διαγράμματος δραστηριοτήτων για αντιστοίχιση με το διάγραμμα του Σχήματος 4.5

Πίνακας 4.4: Αντιστοίχιση μεταξύ των Διαγραμμάτων των Σχημάτων 4.5 και 4.6.

Διάγραμμα 1	Διάγραμμα 2	Τιμή
StartNode > Logged In? > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > EndNode	StartNode > Logged In? > Provide URL > Create Bookmark > EndNode	0.833
StartNode > Logged In? > Login to account > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > EndNode	StartNode > Logged In? > Login > Provide URL > Create Bookmark > EndNode	0.714
StartNode > Logged In? > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > Provide tag text > Add tag to bookmark > EndNode	null	0.000
StartNode > Logged In? > Login to account > Provide bookmark URL > Create Bookmark > Add tag > User wants to add tag? > Provide tag text > Add tag to bookmark > EndNode	null	0.000

Η διαδικασία αντιστοίχισης μεταξύ των ακολουθιών υποδεικνύει ότι ο μηχανικός του δεύτερου διαγράμματος θα μπορούσε να προσθέσει μια νέα ροή που θα περιελάμβανε την επιλογή προσθήκης μιας ετικέτας (tag) όταν δημιουργείται ένας σελιδοδείκτης (bookmark).

## 4.5 Αξιολόγηση

### 4.5.1 Εξόρυξη Λειτουργικών Απαιτήσεων

Προκειμένου να δείξουμε την εγκυρότητα της προσέγγισής μας, πραγματοποιήσαμε εξόρυξη απαιτήσεων και παρέχουμε προτάσεις για ένα έργο λογισμικού. Το επιλεγμένο έργο είναι το έργο Restmarks, η υπηρεσία κοινωνικής δικτύωσης για σελιδοδείκτες που παρουσιάστηκε στο προηγούμενο κεφάλαιο. Οι απαιτήσεις της υπηρεσίας παρουσιάζονται στο Σχήμα 4.7. Τα βασικά σενάρια περιλαμβάνουν την αποθήκευση των σελιδοδεικτών (bookmarks) ενός χρήστη, την κοινή χρήση τους με την κοινότητα και την αναζήτηση για σελιδοδείκτες χρησιμοποιώντας ετικέτες (tags). Για να εφαρμόσουμε τη μεθοδολογία μας, δημιουργήσαμε τους κανόνες συσχέτισης χρησιμοποιώντας τα υπόλοιπα 29 έργα και απομονώσαμε τους κανόνες που ενεργοποιούνται από τις σχολιασμένες απαιτήσεις του Restmarks, συμπεριλαμβανομένων των αντίστοιχων τιμών υποστήριξης (support) και εμπιστοσύνης (confidence).

---

A user must be able to create a user account by providing a username and a password.  
 A user must be able to login to his/her account by providing his username and password.  
 A user that is logged in to his/her account must be able to update his/her password.  
 A logged in user must be able to add a new bookmark to his/her account.  
 A logged in user must be able to retrieve any bookmark from his/her account.  
 A logged in user must be able to delete any bookmark from his/her account.  
 A logged in user must be able to update any bookmark from his/her account.  
 A logged in user must be able to mark his/her bookmarks as public or private.  
 A logged in user must be able to add tags to his/her bookmarks.  
 Any user must be able to retrieve the public bookmarks of any RESTMARKS's community user.  
 Any user must be able to search by tag the public bookmarks of a specific RESTMARKS's user.  
 Any user must be able to search by tag the public bookmarks of all RESTMARKS users.  
 A logged in user must be able to search by tag his/her private bookmarks as well.

---

(a)

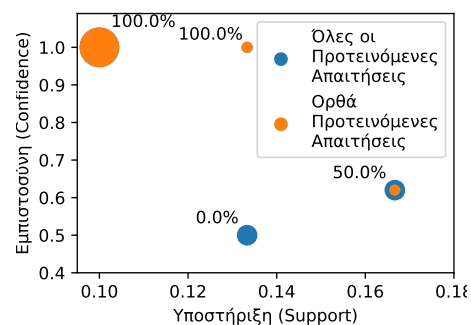
---

+ The user must be able to edit bookmark. ( $\sigma = 0.138, c = 1.0$ )  
 + The user must be able to view bookmark. ( $\sigma = 0.103, c = 1.0$ )  
 + The user must be able to view account. ( $\sigma = 0.103, c = 1.0$ )  
 + The user must be able to edit tag. ( $\sigma = 0.103, c = 1.0$ )  
 + The user must be able to edit account. ( $\sigma = 0.103, c = 1.0$ )  
 + The user must be able to logout account. ( $\sigma = 0.172, c = 0.62$ )  
 - The user must be able to contact account. ( $\sigma = 0.172, c = 0.62$ )  
 - The user must be able to contact bookmark. ( $\sigma = 0.138, c = 0.5$ )  
 - The user must be able to stop account. ( $\sigma = 0.138, c = 0.5$ )

---

+/-: Ορθά/Λανθασμένα Προτεινόμενες Απαιτήσεις  
 $\sigma$ : Υποστήριξη (Support),  $c$ : Εμπιστοσύνη (Confidence)

(b)



(c)

Σχήμα 4.7: Παράδειγμα όπου φαίνονται (a) οι λειτουργικές απαιτήσεις του Restmarks, (a) οι προτεινόμενες απαιτήσεις και (c) η οπτικοποίηση των προτεινόμενων απαιτήσεων

Οι προτεινόμενες απαιτήσεις είναι αρκετά λογικές. Για παράδειγμα, προτείνεται η επιλογή να μπορεί ο χρήστης να επεξεργαστεί μια ετικέτα ή να μπορεί να αποσυνδεθεί από το

λογαριασμό του. Οι απαιτήσεις αυτές είναι πιθανό να έχουν παραλειφθεί από τους μηχανικούς/ενδιαφερόμενα μέρη που καθόρισαν αρχικά τις απαιτήσεις του Restmarks. Επιπλέον, η ποιότητα κάθε πρότασης φαίνεται να έχει σημαντική συσχέτιση με τις αντίστοιχες τιμές στήριξης και εμπιστοσύνης. Αν απεικονίσουμε τον αριθμό των προτεινόμενων απαιτήσεων για κάθε διαφορετικό συνδυασμό τιμών υποστήριξης και εμπιστοσύνης (με κόκκινο χρώμα), μαζί με το ποσοστό των ορθά προτεινόμενων απαιτήσεων για αυτούς τους συνδυασμούς (με μπλε χρώμα), καταλήγουμε στο συμπέρασμα ότι οι περισσότερες προτάσεις έχουν υψηλές τιμές εμπιστοσύνης (confidence). Για παράδειγμα, μπορούμε να σημειώσουμε ότι οι προτεινόμενες απαιτήσεις για το έργο Restmarks που έχουν εμπιστοσύνη ίση με 1 είναι πάντα σωστές (άνω μέρος του Σχήματος 4.7c). Οι προτάσεις για απαιτήσεις με χαμηλότερη εμπιστοσύνη είναι σε αυτήν την περίπτωση λιγότερες (όπως φαίνεται και από τους μικρότερους κύκλους στο Σχήμα 4.7c), ωστόσο μπορούμε να παρατηρήσουμε ότι οι απαιτήσεις με τιμές εμπιστοσύνης (confidence) κοντά στο 0.6/0.7 είναι επίσης πολύ πιθανό να είναι σωστές, αρκεί να έχουν σχετικά υψηλές τιμές υποστήριξης (support).

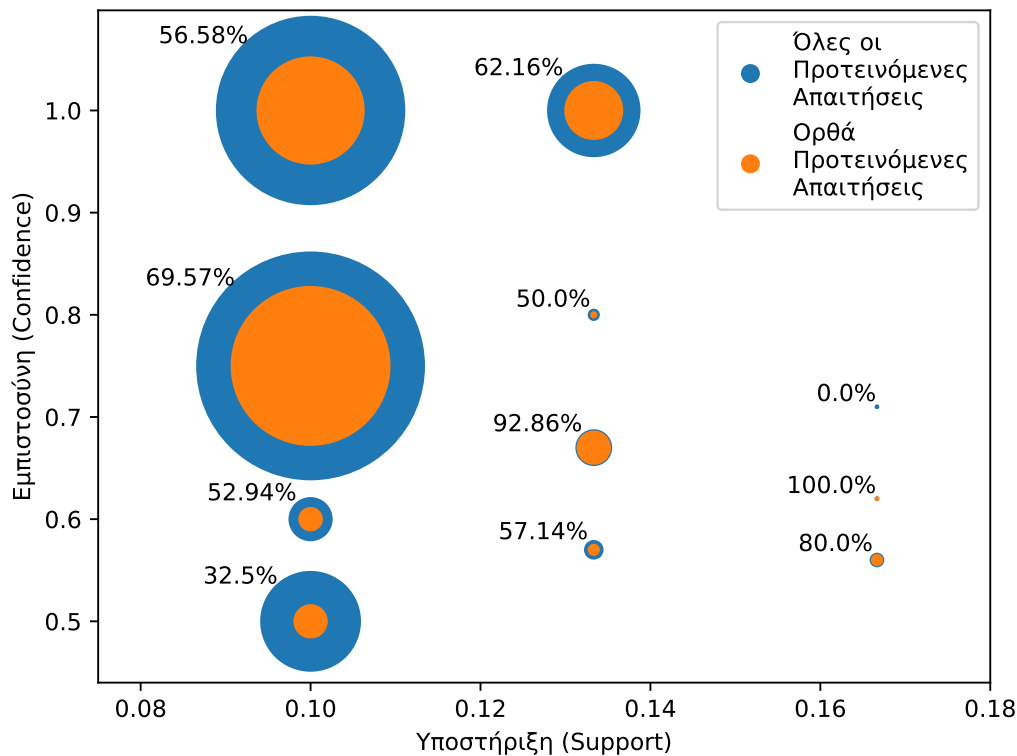
Επιπρόσθετα, εφαρμόσαμε μια λογική cross-validation για να αξιολογήσουμε περαιτέρω την προσέγγισή μας και να διερευνήσουμε την επίδραση των μετρικών της υποστήριξης (support) και της εμπιστοσύνης (confidence) στην ποιότητα των προτάσεων. Διαχωρίσαμε το σύνολο δεδομένων σε 6 τμήματα με 5 έργα το καθένα. Για κάθε τμήμα, αφαιρέσαμε τα 5 έργα του τμήματος από το σύνολο δεδομένων, εξήγαμε τους κανόνες σύνδεσης από τα υπόλοιπα 25 έργα και προτείναμε νέες απαιτήσεις για τα 5 έργα που αφαιρέθηκαν. Μετά από αυτή τη διαδικασία, εξετάσαμε τις προτεινόμενες απαιτήσεις για κάθε έργο και προσδιορίσαμε αν κάθε μία από αυτές θα μπορούσε να είναι έγκυρη. Τα συγκεντρωτικά αποτελέσματα της αξιολόγησης για όλα τα έργα παρουσιάζονται στον Πίνακα 4.5 και οπτικοποιούνται στο Σχήμα 4.8.

Πίνακας 4.5: Αποτελέσματα Αξιολόγησης για τις Προτεινόμενες Απαιτήσεις

Υποστήριξη (Support - $\sigma$ )	Εμπιστοσύνη (Confidence - $c$ )	# Ορθά Προτ. Απαιτήσεις	Προτεινόμενες Απαιτήσεις	% Ορθά Προτ. Απαιτήσεις
0.2	1.0	1	2	50.0%
0.133	1.0	23	37	62.16%
0.1	1.0	43	76	56.58%
0.133	0.8	2	4	50.0%
0.2	0.75	0	1	0.0%
0.1	0.75	64	92	69.57%
0.167	0.71	0	1	0.0%
0.133	0.67	13	14	92.86%
0.167	0.62	1	1	100.0%
0.1	0.6	9	17	52.94%
0.133	0.57	4	7	57.14%
0.167	0.56	4	5	80.0%
0.1	0.5	13	40	32.5%
Σύνολο		177	297	59.6%

Συνολικά, το σύστημά μας πρότεινε 297 απαιτήσεις, από τις οποίες οι 177 ήταν σωστές προτάσεις. Δεδομένου ότι σχεδόν 60% των προτάσεων μπορούν να οδηγήσουν σε χρήσιμες απαιτήσεις, θεωρούμε ότι τα αποτελέσματα είναι ικανοποιητικά. Αν για ένα έργο στον





Σχήμα 4.8: Οπτικοποίηση των προτεινόμενων απαιτήσεων καθώς και τους ποσοστού των ορθά προτεινόμενων απαιτήσεων για διαφορετικές τιμές υποστήριξης (support) και εμπιστοσύνης (confidence)

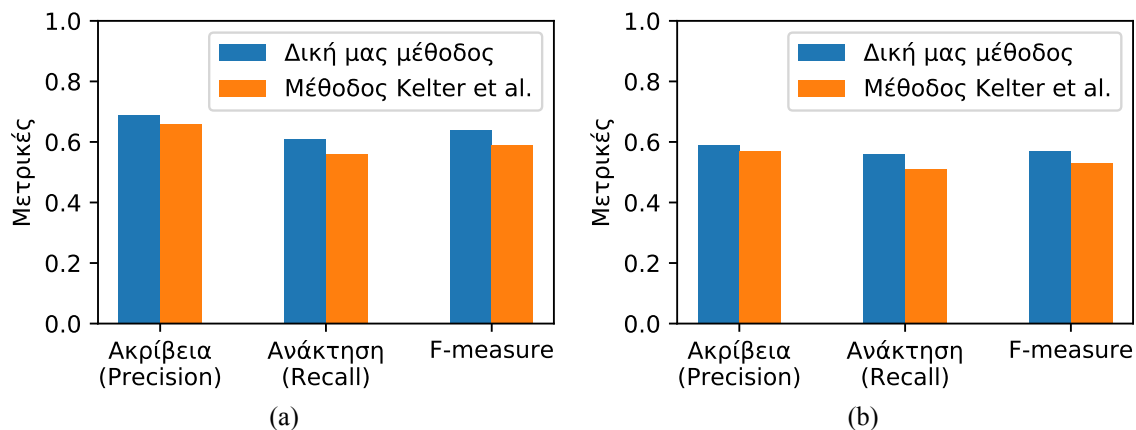
μηχανικό απαιτήσεων είχε παρουσιαστεί ένα σύνολο 10 απαιτήσεων, τότε θα είχε επιλέξει 6 για να προσθέσει στο έργο. Οι περισσότερες από τις προτεινόμενες απαιτήσεις εξάγονται από κανόνες με χαμηλή υποστήριξη (support), κάτι που είναι αναμενόμενο, καθώς το σύνολο δεδομένων μας δεν περιορίζεται σε κάποιον συγκεκριμένο τομέα (είναι δηλαδή domain-agnostic). Ωστόσο, οι κανόνες χαμηλής υποστήριξης δεν οδηγούν αναγκαστικά σε προτάσεις χαμηλής ποιότητας, εφόσον η εμπιστοσύνη (confidence) τους είναι αρκετά μεγάλη. Ενδεικτικά, 2 στις 3 προτάσεις που εξάγονται από κανόνες με τιμές εμπιστοσύνης ίσες με 0.5 μπορεί να μην είναι χρήσιμες. Ωστόσο, θέτοντας την τιμή της εμπιστοσύνης στο 0.75 εξασφαλίζεται ότι για κάθε 3 προτεινόμενες απαιτήσεις περισσότερες των 2 εξ αυτών θα προστεθούν στο έργο.

#### 4.5.2 Εξόρυξη Μοντέλων UML

Για να αξιολογήσουμε το μοντέλο εξόρυξης UML, χρησιμοποιούμε ένα σύνολο δεδομένων με 65 διαγράμματα σεναρίων χρήσης (use case diagrams) και 72 διαγράμματα δραστηριοτήτων (activity diagrams), τα οποία προέρχονται από έργα λογισμικού με διαφορετική σημασιολογία. Η μεθοδολογία μας περιλαμβάνει την εύρεση παρόμοιων διαγραμμάτων και στη συνέχεια την παροχή προτάσεων. Έτσι, αρχικά διαχωρίζουμε τα διαγράμματα σε 6 κατηγορίες, που περιλαμβάνουν τους τομείς της υγείας/κινητικότητας (health/mobility), της κυκλοφορίας/μεταφοράς (traffic/transportation), των κοινωνικών/p2p δικτύων (social/p2p

networks), των υπηρεσιών λογαριασμών/προϊόντων (account/product services), των επιχειρηματικών διαδικασιών (business process) και τέλος ορίζεται και η κατηγορία γενικών διαγραμμάτων (generic diagrams). Κατασκευάζουμε όλα τα πιθανά ζεύγη διαγραμμάτων σεναρίων χρήσης και δραστηριοτήτων, τα οποία είναι 2080 και 2556 ζεύγη αντίστοιχα, και σημειώνουμε κάθε ζεύγος ως σχετικό ή μη σχετικό με βάση το ανήκουν στην ίδια κατηγορία τα διαγράμματα του ζεύγους.

Συγκρίνουμε την προσέγγισή μας με αυτή του Kelter και των συνεργατών του (Kelter et al.) [157]. Δεδομένου ότι οι δύο προσεγγίσεις έχουν παρόμοια δομή, η εφαρμογή τους σε διαγράμματα σεναρίων χρήσης αποτελεί αξιολόγηση της σημασιολογικής μεθοδολογίας μας. Όσον αφορά τα διαγράμματα δραστηριοτήτων, η προσέγγιση του Kelter και των συνεργατών του χρησιμοποιεί ένα στατικό μοντέλο δεδομένων, οπότε η αξιολόγησή μας θα δείξει πώς η χρήση ενός μοντέλου δυναμικής ροής μπορεί να είναι πιο αποτελεσματική. Μετά την εφαρμογή των προσεγγίσεων στα σύνολα των διαγραμμάτων σεναρίων χρήσης και δραστηριοτήτων, οι τιμές των αντιστοιχίσεων κανονικοποιήθηκαν με βάση τον μέσο όρο τους (έτσι ώστε δύο διαγράμματα να ορίζονται ως ένα ζεύγος όταν η τιμή ομοιότητάς τους είναι υψηλότερη από 0.5). Στη συνέχεια, υπολογίσαμε την ακρίβεια (precision), την ανάκληση (recall) και τη μετρική F-measure για κάθε προσέγγιση και για κάθε κατηγορία διαγραμμάτων. Τα αποτελέσματα φαίνονται στο Σχήμα 4.9.



Σχήμα 4.9: Αποτελέσματα ταξινόμησης της προσέγγισής μας και της προσέγγισης του Kelter και των συνεργατών του για (a) διαγράμματα σεναρίων χρήσης και (b) διαγράμματα δραστηριοτήτων

Είναι σαφές ότι η προσέγγισή μας είναι πιο αποτελεσματική από αυτή του Kelter και των συνεργατών του όσον αφορά την εύρεση σχετικών ζευγών διαγραμμάτων τόσο για διαγράμματα σεναρίων χρήσης όσο και για τα διαγράμματα δραστηριοτήτων. Οι υψηλότερες τιμές ανάκλησης (recall) υποδεικνύουν ότι η μεθοδολογία μας μπορεί να ανακτήσει αποτελεσματικά περισσότερα συναφή ζεύγη διαγραμμάτων, ενώ οι υψηλές τιμές ακρίβειας δείχνουν ότι τα ανακτώμενα ζεύγη είναι πράγματι συναφή και ότι τα μη συναφή ζεύγη που θεωρήθηκαν σχετικά (false positives) είναι λιγότερα. Το F-measure επίσης υποδεικνύει ότι η προσέγγισή μας είναι πιο αποτελεσματική για την εξαγωγή σημασιολογικά παρόμοιων ζευγών διαγραμμάτων.

## 4.6 Συμπεράσματα

Αν και αρκετές ερευνητικές προσπάθειες έχουν επικεντρωθεί στον τομέα των εξόρυξης απαιτήσεων, οι περισσότερες από αυτές τις προσπάθειες προσανατολίζονται σε χαρακτηριστικά προϊόντος και/ή βασίζονται σε λεξιλόγια συγκεκριμένων τομέων. Σε αυτό το κεφάλαιο παρουσιάσαμε μια μεθοδολογία εξόρυξης που μπορεί να βοηθήσει τους μηχανικούς να καθορίσουν τις λειτουργικές απαιτήσεις και τα μοντέλα UML για το υπό ανάπτυξη σύστημα.

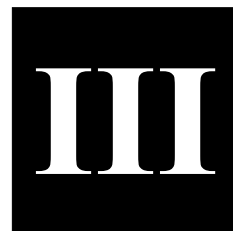
Όσον αφορά τις λειτουργικές απαιτήσεις, το σύστημά μας εφαρμόζει τεχνικές εξόρυξης κανόνων συσχέτισης για να εξάγει χρήσιμους κανόνες και χρησιμοποιεί αυτούς τους κανόνες για τη δημιουργία προτάσεων νέων απαιτήσεων. Όπως δείξαμε στο προηγούμενο υποκεφάλαιο, το σύστημά μας παρέχει ένα σύνολο προτάσεων από τις οποίες το 60% περίπου είναι ορθές. Έτσι, δεδομένου ότι ο μηχανικός απαιτήσεων (ή γενικά ένα ενδιαφερόμενο μέρος) έχει συντάξει ένα σύνολο απαιτήσεων, θα μπορούσε στη συνέχεια να ελέγξει με εύκολο τρόπο αν έχει παραλείψει κάποιες ενδεχομένως σημαντικές απαιτήσεις.

Η μεθοδολογία μας για την εξόρυξη μοντέλων UML έχει σημασιολογία που δεν εξαρτάται από το λεξιλόγιο κάποιου τομέα (είναι domain-agnostic) και χρησιμοποιεί πρακτικές αναπαραστάσεις για UML διαγράμματα σεναρίων χρήσης και δραστηριοτήτων. Ως εκ τούτου, καλύπτονται τόσο η στατική όσο και η δυναμική όψη των έργων λογισμικού, ενώ λαμβάνονται υπόψη τα ιδιαίτερα χαρακτηριστικά των διαγραμμάτων σεναρίων χρήσης και των διαγραμμάτων δραστηριοτήτων, αφού για τα πρώτα χρησιμοποιείται μια δομή συνόλων και για τα δεύτερα μια δομή που μοντελοποιεί τη ροή των ενεργειών. Τα αποτελέσματα της αξιολόγησής μας δείχνουν ότι η προσέγγισή μας είναι πιο αποτελεσματική από τις τρέχουσες προσεγγίσεις της βιβλιογραφίας.

Ως πιθανές μελλοντικές προεκτάσεις, αναφέρουμε αρχικά ότι ένα ενδιαφέρον πρόβλημα στον τομέα των απαιτήσεων είναι ο συνδυασμός των προτιμήσεων των ενδιαφερόμενων (stakeholder preference). Στο πλαίσιο των λειτουργικών απαιτήσεων, μια επέκταση του συστήματός μας μπορεί να εφαρμοστεί σε ένα σενάριο με πολλούς ενδιαφερόμενους, όπου οι διαφορετικές απαιτήσεις τους θα συνδυάζονται και έτσι θα εξάγεται ένα σύνολο κανόνων συσχέτισης που θα είναι πιο ταιριαστοί στο έργο. Μια άλλη ενδιαφέρουσα μελλοντική κατεύθυνση θα ήταν να βελτιωθεί η μεθοδολογία σημασιολογικής αντιστοίχισης με την ενσωμάτωση πληροφοριών από online πηγές πληροφοριών λογισμικού (π.χ. GitHub). Τέλος, η μεθοδολογία εξόρυξης μοντέλων UML θα μπορούσε να περιλαμβάνει την ανάλυση διαφορετικών τύπων διαγραμμάτων, όπως UML διαγράμματα κλάσεων (class diagrams) ή διαγράμματα ακολουθιών (sequence diagrams).



**Μέρος**



**ΕΞΟΥΧΗ ΠΗΓΑΙΟΥ ΚΩΔΙΚΑ**



# 5

## Ευρετηριοποίηση Κώδικα για Επαναχρησιμοποίηση

### 5.1 Επισκόπηση

Τελευταία, η ιδέα της ανάπτυξης λογισμικού με πρακτικές ανοιχτού κώδικα (open-source software development) έχει αλλάξει τον τρόπο με τον οποίο αναπτύσσεται και διανέμεται το λογισμικό. Αρκετοί ερευνητές και επαγγελματίες αναπτύσσουν πλέον λογισμικό ανοιχτού κώδικα και διαμοιράζονται τα έργα τους στο διαδίκτυο. Αυτή η τάση έχει αποκτήσει δυναμική κατά την τελευταία δεκαετία, οδηγώντας στην ανάπτυξη πολλών υπηρεσιών φιλοξενίας κώδικα, όπως το GitHub<sup>1</sup> και το Sourceforge<sup>2</sup>. Ένα από τα βασικότερα επιχειρήματα υπέρ της ανάπτυξης λογισμικού ανοιχτού κώδικα είναι ότι τα διαθέσιμα αποθετήρια λογισμικού μπορούν να αξιοποιηθούν για τη αναζήτηση χρήσιμου κώδικα και καλών πρακτικών.

Ένα άλλο ισχυρό επιχειρήμα είναι ότι οι προγραμματιστές μπορούν να επωφεληθούν σε μεγάλο βαθμό από την επαναχρησιμοποίηση του λογισμικού, προκειμένου να μειωθεί ο χρόνος που δαπανάται για την ανάπτυξη, καθώς και να βελτιωθεί η ποιότητα των έργων τους. Σήμερα, η διαδικασία της ανάπτυξης λογισμικού περιλαμβάνει την αναζήτηση επαναχρησιμοποιήσιμου κώδικα σε μηχανές αναζήτησης γενικής χρήσης (π.χ. Google ή Yahoo), σε αποθήκες λογισμικού ανοιχτού κώδικα (π.χ. GitHub) ή ακόμα και σε κοινότητες ερωταπαντήσεων (π.χ. Stack Overflow<sup>3</sup>). Αυτή η ανάγκη αναζήτησης και επαναχρησιμοποίησης τμημάτων λογισμικού έχει οδηγήσει στη σχεδίαση μηχανών αναζήτησης που ειδικεύονται στην αναζήτηση κώδικα. Οι *μηχανές αναζήτησης κώδικα* (Code Search Engines - CSEs) βελτιώνουν τη διαδικασία αναζήτησης σε αποθήκες πηγαίου κώδικα, επιτρέποντας την αναζήτηση χρησιμοποιώντας λέξεις-κλειδιά ή ακόμα και χρησιμοποιώντας ερωτήματα με βάση τη σύνταξη (syntax-aware queries, π.χ. αναζήτηση μεθόδου με συγκεκριμένες παραμέτρους). Μια άλλη κατηγορία πιο εξειδικευμένων συστημάτων είναι αυτή των *Συστημάτων Προτάσεων στην Τεχνολογία Λογισμικού* (Recommendation Systems in Software Engineering -

---

<sup>1</sup><https://github.com/>

<sup>2</sup><https://sourceforge.net/>

<sup>3</sup><http://stackoverflow.com/>

RSSes). Στο πλαίσιο της επαναχρησιμοποίησης κώδικα, τα RSSes είναι έξυπνα συστήματα που εξάγουν το ερώτημα από τον πηγαίο κώδικα του προγραμματιστή (query extraction) και επεξεργάζονται τα αποτελέσματα των CSEs για να προτείνουν χρήσιμα τμήματα κώδικα, συνοδευόμενα από πληροφορίες που βοηθούν τον προγραμματιστή να τα κατανοήσει και να τα επαναχρησιμοποιήσει στον κώδικά του. Σε αυτό το κεφάλαιο επικεντρωνόμαστε στις CSEs, ενώ στο επόμενο επικεντρωνόμαστε στα RSSes.

Αν και οι σύγχρονες CSEs έχουν σίγουρα το δικό τους μερίδιο στην αγορά, η σωστή σχεδίαση ενός συστήματος που καλύπτει τις ανάγκες των προγραμματιστών εξακολουθεί να είναι ένα ανοιχτό ερευνητικό ερώτημα. Συγκεκριμένα, οι αδυναμίες των σύγχρονων CSEs εντοπίζονται σε ένα ή περισσότερα από τα ακόλουθα ζητήματα: (α) δεν προσφέρουν ευέλικτες επιλογές για να επιτρέπουν την εκτέλεση προηγμένων ερωτημάτων (π.χ. ερωτήματα με βάση τη σύνταξη - syntax-aware queries, κανονικές εκφράσεις - regular expressions, ερωτήματα για snippets ή για ολόκληρα έργα), β) δεν ενημερώνονται αρκετά συχνά για να αντιμετωπίσουν τις ad hoc ανάγκες των προγραμματιστών, δεδομένου ότι δεν είναι συνδεδεμένες με υπηρεσίες φιλοξενίας κώδικα, και γ) δεν προσφέρουν πλήρως τεκμηριωμένα APIs, ώστε να επιτρέπουν εύκολα τη χρήση τους από άλλα εργαλεία. Το τελευταίο ζήτημα είναι αρκετά σημαντικό και για τα RSSes: ο στόχος των RSSes να προσφέρουν εξατομικευμένες προτάσεις στον προγραμματιστή σύμφωνα με το πρόβλημά του συνδέεται στενά με την ανάγκη για χρήση εξειδικευμένων CSEs με ανοικτά APIs, κάτι που έχει τονιστεί από διάφορους ερευνητές [95–97, 100].

Σε αυτό το κεφάλαιο παρουσιάζουμε την AGORA<sup>4</sup>, μια CSE καλύπτει τις προαναφερθείσες απαιτήσεις. Η AGORA αποθηκεύει όλα τα στοιχεία του πηγαίου κώδικα, ενώ επιπλέον πραγματοποιείται ευρετηριοποίηση (indexing) του κώδικα σε επίπεδο όρων (token level) και διατηρείται η ιεραρχία των έργων και των αρχείων τους. Έτσι, υποστηρίζει την αναζήτηση με βάση τη σύνταξη (syntax-aware search), δηλαδή την αναζήτηση για κλάσεις με συγκεκριμένες μεθόδους κ.α., την αναζήτηση χρησιμοποιώντας κανονικές εκφράσεις (regular expressions), την αναζήτηση μικρών τμημάτων κώδικα (snippets) ή και αντικειμένων που αποτελούνται από πολλαπλά τμήματα κώδικα (σε επίπεδο έργου). Τα αποθηκευμένα έργα ενημερώνονται συνεχώς λόγω της σύνδεσης της AGORA με το GitHub. Τέλος, η AGORA προσφέρει ένα RESTful API που επιτρέπει την απρόσκοπτη χρήση της από άλλα εργαλεία.

## 5.2 Βιβλιογραφία για τις Μηχανές Αναζήτησης Κώδικα

### 5.2.1 Κριτήρια Μηχανών Αναζήτησης Κώδικα

Σύμφωνα με τη βιβλιογραφία και με βάση την παραπάνω συζήτηση, οι CSEs θα πρέπει να πληρούν τα ακόλουθα κριτήρια:

---

<sup>4</sup>Ο όρος “agora” αναφέρεται στις αγορές που αποτελούσαν το κεντρικό σημείο των αρχαίων ελληνικών πόλεων-κρατών. Η σημασία του όρου σήμερα είναι κάπως διαφορετική, ωστόσο επιλέγοντας τον αρχαίο ορισμό, μπορούμε να θεωρήσουμε ότι η “αγορά” είναι μια συνέλευση, ένας τόπος όπου οι άνθρωποι βρίσκονταν όχι μόνο για να ανταλλάξουν εμπορεύματα αλλά και για να ανταλλάξουν ιδέες. Είναι ένας όρος που είναι κατάλληλος για τη μηχανή αναζήτησής μας, καθώς την οραματιζόμαστε ως ένα ηλεκτρονικό μέρος όπου οι προγραμματιστές μπορούν να ανταλλάσσουν τον πηγαίο κώδικα τους και ως εκ τούτου τις ιδέες τους.



**Κριτήριο 1: Αναγνώριση σύνταξης (syntax-awareness)**

Το σύστημα πρέπει να υποστηρίζει ερωτήματα με βάση συγκεκριμένα στοιχεία ενός αρχείου, π.χ. όνομα κλάσης, όνομα μεθόδου, τύπος επιστροφής μεθόδου κ.α.

**Κριτήριο 2: Κανονικές εκφράσεις (regular expressions)**

Οι κανονικές εκφράσεις πρέπει να υποστηρίζονται έτσι ώστε ο χρήστης να μπορεί να αναζητήσει ένα συγκεκριμένο μοτίβο.

**Κριτήριο 3: Αναζήτηση μικρών τμημάτων κώδικα (snippet search)**

Παρόλο που οι περισσότερες CSEs υποστηρίζουν την αναζήτηση σε μορφή κειμένου, αυτό είναι συνήθως αναποτελεσματικό για ερωτήσεις πολλαπλών γραμμών. Μια CSE θα πρέπει να επιτρέπει την αναζήτηση μικρών τμημάτων κώδικα, που είναι γνωστά ως snippets.

**Κριτήριο 4: Αναζήτηση έργων (project search)**

Το σύστημα πρέπει να επιτρέπει στο χρήστη την αναζήτηση έργων με κάποια συγκεκριμένη λειτουργικότητα. Με άλλα λόγια, το σύστημα πρέπει να αποθηκεύει τα δεδομένα με κάποιο είδος δομής (schema) έτσι ώστε να μπορούν να απαντηθούν σύνθετα ερωτήματα για την εύρεση έργων που περιέχουν ένα ή περισσότερα αρχεία.

**Κριτήριο 5: Σύνδεση με υπηρεσίες φιλοξενίας κώδικα (code hosting integration)**

Η εισαγωγή έργων στο αποθετήριο της CSE και η ενημέρωση του ευρετηρίου (index) θα πρέπει να είναι απλή. Ως εκ τούτου, οι CSEs πρέπει να συνδέονται με υπηρεσίες φιλοξενίας κώδικα (π.χ. GitHub).

**Κριτήριο 6: Υποστήριξη API (API support)**

Το σύστημα πρέπει να διαθέτει ανοικτό (public) API. Παρόλο που το κριτήριο αυτό μπορεί να φαίνεται περιττό, είναι κρίσιμο επειδή ένα καλά τεκμηριωμένο ανοικτό API επιτρέπει τη χρήση του συστήματος με πολλούς διαφορετικούς τρόπους, π.χ. δημιουργώντας ένα UI, κατασκευάζοντας κάποιο IDE plugin, κ.λπ.

Στις επόμενες ενότητες παρέχουμε μια ιστορική ανασκόπηση για την περιοχή των CSEs και συζητούμε τις ελλείψεις των γνωστών CSEs σε σχέση με τα παραπάνω κριτήρια. Με αυτόν τον τρόπο δικαιολογούμε τη λογική μας για τη δημιουργία μιας νέας CSE.

## 5.2.2 Ιστορική Ανασκόπηση Μηχανών Αναζήτησης Κώδικα

Δεδομένου του τεράστιου όγκου του κώδικα που υπάρχει σε online αποθετήρια (όπως αναλύθηκε ήδη στο Κεφάλαιο 2), το βασικό πρόβλημα που προκύπτει είναι η αναζήτηση και η επιτυχής ανάκτηση κατάλληλου κώδικα ανάλογα με το εκάστοτε ερώτημα. Κάποιες από τις πιο δημοφιλείς CSEs παρουσιάζονται στον Πίνακα 5.1.

Μια σημαντική ερώτηση που διερευνάται σε αυτό το κεφάλαιο είναι εάν αυτές οι CSEs είναι όσο αποτελεσματικές και χρήσιμες θα μπορούσαν να είναι για τον προγραμματιστή. Το Codase ήταν μια από τις πρώτες μηχανές αναζήτησης που υποστήριζε αναζήτηση με βάση τη σύνταξη (π.χ. αναζήτηση για όνομα κλάσης ή μεθόδου) και γενικώς αποτελούσε μια πολλά υποσχόμενη επιλογή· ωστόσο, η ανάπτυξή του διεκόπη κατά τη φάση του beta testing. Το Krugle Open Search, από την άλλη πλευρά, είναι μία από τις πρώτες CSEs που είναι ακόμα ενεργές. Το βασικό πλεονέκτημα του Krugle είναι τα υψηλής ποιότητας αποτελέσματά του· η υπηρεσία προσφέρει αρκετές δυνατότητες αναζήτησης (π.χ. υποστηρίζονται ερωτήματα

με βάση τη σύνταξη) σε ένα μικρό αλλά προσεκτικά επιλεγμένο αριθμό έργων λογισμικού, που περιλαμβάνει έργα όπως το Apache Ant, η jython κ.α. Παρόλο που το Krugle καλύπτει επαρκώς τα κριτήρια 1 και 3, δεν είναι συνδεδεμένο με κάποια υπηρεσία φιλοξενίας κώδικα και δεν παρέχει κάποιο ανοικτό API.

Πίνακας 5.1: Δημοφιλείς Μηχανές Αναζήτησης Κώδικα<sup>5</sup>

Μηχανή Αναζήτησης	Έτος	Ιστοσελίδα	# Έργα
Codase	2005–2014	<a href="http://www.codase.com/">http://www.codase.com/</a>	> 10K
Krugle	2006–σήμερα	<a href="http://opensearch.krugle.org/">http://opensearch.krugle.org/</a>	~ 500
Google CSE	2006–2013	<a href="http://www.google.com/codesearch">http://www.google.com/codesearch</a>	> 250K
MeroBase	2007–2014	<a href="http://www.merobase.com/">http://www.merobase.com/</a>	> 5K
Sourcerer	2007–2012	<a href="http://sourcerer.ics.uci.edu/">http://sourcerer.ics.uci.edu/</a>	> 70K
GitHub CSE	2008–σήμερα	<a href="https://github.com/search">https://github.com/search</a>	~ 14.2M
BlackDuck	2008–2016	<a href="https://code.openhub.net/">https://code.openhub.net/</a>	> 650K
Searchcode	2010–σήμερα	<a href="https://searchcode.com/">https://searchcode.com/</a>	> 300K

Το 2006, ένας από τους μεγάλους παίκτες, η Google, αποφάσισε να εισέλθει στην αρένα των μηχανών αναζήτησης κώδικα. Η CSE της Google ήταν γρήγορη, υποστήριζε κανονικές εκφράσεις και επέτρεπε την αναζήτηση κώδικα σε πάνω από 250000 έργα. Οι όποιες αδυναμίες της εντοπίζονται στα κριτήρια 3 και 4, καθώς η CSE δεν υποστήριζε την αναζήτηση μικρών τμημάτων κώδικα (snippets) ή την αναζήτηση αντικειμένων που αποτελούνται από πολλαπλά τμήματα κώδικα. Ωστόσο, το 2011 η Google ανακοίνωσε την επικείμενη διακοπή της λειτουργίας της CSE<sup>6</sup> για να την κλείσει τελικά το 2013. Παρά το κλείσιμό της, η υπηρεσία αύξησε το ενδιαφέρον για τις CSEs, οδηγώντας τους προγραμματιστές να αναζητήσουν εναλλακτικές λύσεις.

Μια τέτοια εναλλακτική λύση ήταν το MeroBase [90]. Το ευρετήριο (index) του MeroBase μοντελοποιούσε τα έργα ως αντικείμενα λογισμικού και υποστήριζε την αναζήτηση και ανάκτηση πηγαίου κώδικα και έργων σύμφωνα με διάφορα κριτήρια, όπως κριτήρια πηγαίου κώδικα (π.χ. ονόματα κλάσεων και μεθόδων), κριτήρια αδειών λογισμικού (software licenses), κ.α. Όπως σημειώνεται από τους δημιουργούς της MeroBase [90], η κύρια αδυναμία του είναι η επίπεδη δομή του (flat structure), η οποία είναι δύσκολο να αλλάξει λόγω του τρόπου ευρετηριοποίησης. Ως αποτέλεσμα αυτού, το MeroBase δεν έχει τις δυνατότητες προηγμένης αναζήτησης που περιγράφονται στα κριτήρια 2 και 3, ενώ το ευρετήριό του, αν και μεγάλο, δεν είναι ενημερώσιμο καθώς δε συνδέεται με κάποια υπηρεσία φιλοξενίας κώδικα. Μια παρόμοια πρόταση είναι η μηχανή Sourcerer [92], που χρησιμοποιεί ένα σχήμα (schema) με βάση τη σύνταξη του κώδικα και έτσι υποστηρίζει διαφορετικούς τύπους ερωτημάτων. Ωστόσο, το Sourcerer επίσης δε συνδέεται με online αποθετήρια, οπότε το ευρετήριο του πρέπει να ενημερώνεται χειροκίνητα.

Το GitHub, ο πιο δημοφιλής ιστότοπος φιλοξενίας πηγαίου κώδικα σήμερα, έχει τη δική του μηχανή αναζήτησης για τα αποθετήρια του που είναι ανοικτού κώδικα. Η GitHub CSE

<sup>5</sup>Ο Πίνακας 5.1 περιλαμβάνει μόνο CSEs, δηλαδή μηχανές αναζήτησης που επιτρέπουν την αναζήτηση πηγαίου κώδικα. Έτσι, μηχανές όπως το GrepCode (<http://grepcode.com/>), το NerdyData (<http://nerdydata.com/>) ή το SymbolHound (<http://symbolhound.com/>) δεν περιλαμβάνονται, καθώς αφορούν την αναζήτηση έργων (στην περίπτωση του GrepCode), κώδικα για ιστοσελίδες (στην περίπτωση του NerdyData) ή απαντήσεις από υπηρεσίες ερωταπαντήσεων (στην περίπτωση του SymbolHound).

<sup>6</sup>Πηγή: <http://googleblog.blogspot.gr/2011/10/fall-sweep.html>

είναι γρήγορη και αξιόπιστη, παρέχοντας αποτελέσματα όχι μόνο για τον πηγαίο κώδικα αλλά και για στατιστικά στοιχεία των αποθετηρίων (commits, forks, κ.λπ.). Το μεγαλύτερο πλεονέκτημά της όσον αφορά τον πηγαίο κώδικα είναι το τεράστιο ευρετήριο έργων που διαθέτει, ενώ το API της είναι επίσης πολύ καλά ορισμένο. Ωστόσο, οι επιλογές αναζήτησης είναι σχετικά ξεπερασμένες: δεν υποστηρίζονται τα ερωτήματα με κανονικές εκφράσεις ή γενικά τα ερωτήματα με βάση τη σύνταξη. Ως αποτέλεσμα, δεν καλύπτει κανένα από τα πρώτα 4 κριτήρια.

Το Black Duck Open Hub είναι επίσης μία από τις πιο δημοφιλείς CSEs με ένα αρκετά μεγάλο ευρετήριο. Η εταιρεία BlackDuck απέκτησε την CSE της Koders το 2008 και τη συγχώνευσε με το ευρετήριο του Ohloh (που αποκτήθηκε το 2010) για να παρέχει τελικά τη μηχανή κώδικα Ohloh Code το 2012. Παρόλο που η υπηρεσία είχε αρχικά ως στόχο την κάλυψη του κενού που άφησε το κλείσιμο της Google CSE<sup>7</sup>, αργότερα μετατράπηκε σε έναν πλήρη κατάλογο έργων πηγαίου κώδικα, παρέχοντας στατιστικά στοιχεία, πληροφορίες αδειών κ.α. Η μηχανή Ohloh Code μετονομάστηκε σε BlackDuck Open Hub Code Search το 2014. Η υπηρεσία υποστηρίζει τη δημιουργία ερωτημάτων με βάση τη σύνταξη και προσφέρει plugins για γνωστά IDEs (Eclipse, Microsoft Visual Studio). Ωστόσο, η αναζήτηση snippets και η αναζήτηση σε επίπεδο έργου δεν υποστηρίζονται (κριτήρια 3 και 4).

Τέλος, μια πολλά υποσχόμενη υπηρεσία στον τομέα των CSEs είναι το Searchcode. Κατά την έναρξη της λειτουργίας του (με το όνομα search[co.de]), η υπηρεσία επέτρεπε την αναζήτηση τεκμηρίωσης για δημοφιλείς βιβλιοθήκες. Σήμερα, συνδέεται αρκετές αποθήκες λογισμικού (GitHub, Sourceforge κ.λπ.) για την ανάκτηση και ευρετηριοποίηση πηγαίου κώδικα, κάτι που αποτελεί κι ένα από τα βασικότερα πλεονεκτήματά του. Ωστόσο, δεν υποστηρίζει ερωτήματα με βάση τη σύνταξη, ερωτήματα για snippets αποσπάσματα ή ερωτήματα σε επίπεδο έργου, επομένως τα κριτήρια 1, 3 και 4 δεν πληρούνται. Συμπερασματικά, καμία από τις CSEs που φαίνονται στον Πίνακα 5.1 (και καμία άλλη εξ όσων γνωρίζουμε) δεν καλύπτει επαρκώς όλα τα παραπάνω κριτήρια.

### 5.2.3 Χαρακτηριστικά της AGORA

Η αποτελεσματικότητα μιας CSE εξαρτάται από διάφορους παράγοντες, όπως η ποιότητα των έργων στο ευρετήριό της και ο τρόπος με τον οποίο αυτά τα έργα είναι ευρετηριοποιημένα. Συγκρίνουμε την AGORA με άλλες γνωστές CSEs ως προς τα κριτήρια που έχουμε ορίσει. Η σύγκριση φαίνεται στον Πίνακα 5.2.

Όσον αφορά τις λειτουργίες που σχετίζονται με την αναζήτηση, δηλαδή τα κριτήρια 1-4, η AGORA είναι η CSE που επιτρέπει το μεγαλύτερο εύρος τύπων ερωτημάτων. Υποστηρίζει ερωτήματα με βάση τη σύνταξη, κανονικές εκφράσεις, ακόμη και αναζήτηση snippets και έργων. Άλλες CSEs, όπως το Codase ή το BlackDuck, καλύπτουν τα κριτήρια της αναγνώρισης σύνταξης και κανονικών εκφράσεων, χωρίς ωστόσο να υποστηρίζουν την αναζήτηση snippets ή την αναζήτηση έργων. Το Krugle υποστηρίζει την αναζήτηση snippets, αλλά δεν επιτρέπει τη χρήση κανονικών εκφράσεων ή την αναζήτηση έργων. Το MeroBase και το Sourcerer είναι οι μόνες CSEs (εκτός της AGORA) που υποστηρίζουν μερικώς την αναζήτηση σε επίπεδο έργων, αλλά στοχεύουν κυρίως στην τεκμηρίωση των έργων.

<sup>7</sup>Πηγή: <http://techcrunch.com/2012/07/20/ohloh-wants-to-fill-the-gap-left-by-google-code-search/>

Πίνακας 5.2: Σύγκριση Χαρακτηριστικών Δημοφιλών Μηχανών Αναζήτησης Κώδικα με την AGORA

Μηχανή Αναζήτησης	Κριτήριο 1 Αναγνώριση σύνταξης	Κριτήριο 2 Κανονικές εκφράσεις	Κριτήριο 3 Αναζήτηση snippets	Κριτήριο 4 Αναζήτηση έργων	Κριτήριο 5 Σύνδεση με αποθετήρια	Κριτήριο 6 Υποστήριξη API
AGORA	✓	✓	✓	✓	✓	✓
Codase	✓	×	×	×	×	✓
Krugle	✓	×	✓	×	×	×
Google CSE	✓	✓	×	×	✓	✓
MeroBase	✓	×	×	✓	×	✓
Sourcerer	✓	✓	×	✓	×	✓
GitHub CSE	×	×	×	×	✓	✓
BlackDuck	✓	✓	×	×	×	×
Searchcode	×	✓	×	×	✓	✓

✓: Υποστηρίζεται ×: Δεν υποστηρίζεται ✓: Υποστηρίζεται μερικώς

Η σύνδεση με υπηρεσίες φιλοξενίας κώδικα είναι άλλο ένα χαρακτηριστικό που παραβλέπεται στις σύγχρονες CSEs. Η GitHub CSE και η καταργηθείσα Google CSE προφανώς το υποστηρίζουν, αφού οι υπηρεσίες φιλοξενίας βρίσκονται ουσιαστικά στους δικούς τους διακομιστές (GitHub και Google Code αντίστοιχα). Πέρα από αυτές τις CSEs, όμως, το Searchcode είναι η μόνη άλλη υπηρεσία της οποίας το ευρετήριο συνδέεται με ιστότοπους φιλοξενίας, όπως το Github, το Sourceforge, κ.α.

Το κριτήριο 6 είναι επίσης πολύ σημαντικό, δεδομένου ότι οι CSEs με τεκμηριωμένα API μπορούν εύκολα να επεκταθούν με τη δημιουργία διαδικτυακών UI ή plugins για κάποιο IDE. Οι περισσότερες CSEs πληρούν αυτό το κριτήριο, με το GitHub και το Searchcode να έχουν δύο από τα πιο ολοκληρωμένα APIs. Το API της AGORA, που ουσιαστικά χρησιμοποιεί το API που παρέχεται από το Elasticsearch, είναι καλά σχεδιασμένο και επιτρέπει την εκτέλεση διαφορετικών τύπων ερωτημάτων.

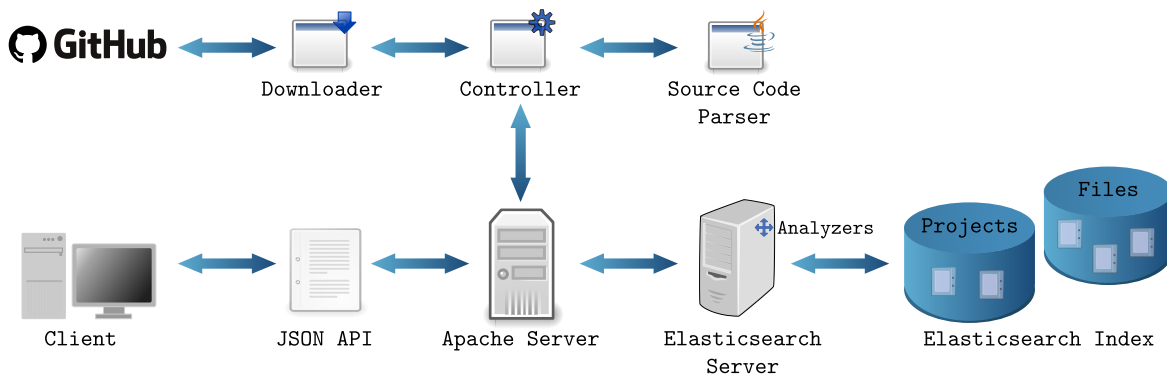
Δεν ισχυριζόμαστε ότι δημιουργήσαμε μια συνολικά ισχυρότερη CSE· αυτό θα απαιτούσε επίσης την αξιολόγηση άλλων πτυχών, όπως π.χ. το μέγεθος του ευρετηρίου της ή το πλήθος των γλωσσών που υποστηρίζονται. Η AGORA είναι σε πειραματικό στάδιο. Ωστόσο, η αρχιτεκτονική της όσον αφορά τις γλώσσες που υποστηρίζονται καθώς και τις υπηρεσίες φιλοξενίας κώδικα είναι πλήρως προσαρμόσιμη· η προσθήκη νέων γλωσσών και/ή η ενσωμάτωση νέων υπηρεσιών φιλοξενίας κώδικα (εκτός του GitHub) είναι εφικτή. Πρακτικά, η AGORA αποτελεί μια λύση που είναι κατάλληλη για την υποστήριξη της έρευνας στην περιοχή της επαναχρησιμοποίησης λογισμικού.

### 5.3 Η Μηχανή Αναζήτησης Κώδικα AGORA

Σε αυτό το υποκεφάλαιο παρουσιάζουμε λεπτομερώς τη μηχανή αναζήτησης κώδικα AGORA που προτείνουμε ως λύση για την επαναχρησιμοποίηση κώδικα.

### 5.3.1 Επισκόπηση

Η αρχιτεκτονική της AGORA φαίνεται στο Σχήμα 5.1. Το Elasticsearch [178] επιλέχθηκε ως το ευρετήριο αναζήτησης καθώς αξιοποιεί το Lucene, ενώ συγχρόνως επιτρέπει την αποθήκευση αντικειμένων με ιεραρχικού τύπου δομή. Η αποθήκευση στο Elasticsearch είναι αρκετά παρόμοια με τη λογική των NoSQL βάσεων και βασίζεται σε έγγραφα (documents). Τα αντικείμενα αποθηκεύονται ως έγγραφα (που αντιστοιχούν σε εγγραφές - records σε σχεσιακό σχήμα), τα έγγραφα αποθηκεύονται σε συλλογές (collections), που είναι το αντίστοιχο των σχεσιακών πινάκων (tables), και τέλος οι συλλογές αποθηκεύονται σε ευρετήρια (indexes). Ορίζουμε δύο συλλογές: *projects* και *files*. Η συλλογή περιέχει πληροφορίες για έργα λογισμικού, ενώ η συλλογή *files* περιέχει πληροφορίες σχετικά με τα αρχεία πηγαίου κώδικα.



Σχήμα 5.1: Αρχιτεκτονική της AGORA

Όπως απεικονίζεται στο Σχήμα, το ευρετήριο του Elasticsearch (*Elasticsearch Index*) είναι προσβάσιμο μέσω του διακομιστή *Elasticsearch Server*. Επιπλέον, ένας διακομιστής Apache (*Apache Server*) παρεμβάλλεται για να ελέγχει όλη την επικοινωνία με το διακομιστή. Ο διακομιστής Apache επιτρέπει μόνο εξουσιοδοτημένη είσοδο στο ευρετήριο, διασφαλίζοντας έτσι ότι μόνο ο διαχειριστής μπορεί να εκτελεί συγκεκριμένες ενέργειες (π.χ. διαγραφή ενός εγγράφου, δημιουργία αντιγράφων ασφαλείας του ευρετηρίου, κ.λπ.). Για τους χρήστες της AGORA, ο διακομιστής είναι προσβάσιμος μέσω του *JSON API* που παρέχεται από το Elasticsearch.

Ο *Controller* που φαίνεται στο Σχήμα 5.1 είναι ένα από τα πιο σημαντικά τμήματα της AGORA. Ο *Controller*, που είναι υλοποιημένος σε Python, παρέχει μια διεπαφή για την εισαγωγή έργων από το GitHub στο Elasticsearch. Συγκεκριμένα, ο *Controller* παίρνει ως είσοδο ένα GitHub URL για ένα έργο και το παρέχει στον *Downloader*. Το τμήμα του *Downloader* ελέγχει αν ο πηγαίος κώδικας του έργου έχει ήδη κατέβει και τον κατεβάζει ή τον ενημερώνει αντίστοιχα. Ο *Downloader* περιλαμβάνει δύο υποτμήματα, τον *GitHub API Downloader* και τον *Git Downloader*. Το υποτμήμα *GitHub API Downloader* κατεβάζει πληροφορίες για ένα έργο χρησιμοποιώντας το GitHub API, συμπεριλαμβανομένης της διεύθυνσης git (git address) του έργου και του δένδρου του πηγαίου κώδικα (source code tree). Το υποτμήμα *Git Downloader* είναι στην πραγματικότητα ένας wrapper για το εργαλείο Git, που επιτρέπει το κατέβασμα του κώδικα ενός έργου από το GitHub (λειτουργία git clone) ή την ενημέρωσή του κώδικα ενός έργου αν έχει ήδη κατέβει (λειτουργία git pull). Με βάση το δένδρο που παρέχεται από το API, ο *Downloader* παράγει μια λίστα αρχείων και

sha ids που στέλνεται στον Controller. Τα sha ids χρησιμοποιούνται για να προσδιοριστεί ποια αρχεία πρέπει να προστεθούν στο ευρετήριο, ποια πρέπει να ενημερωθούν και ποια από τα υπάρχοντα αρχεία πρέπει να διαγραφούν από το ευρετήριο.

Πριν την εισαγωγή νέων εγγράφων στο ευρετήριο, ο κώδικας του έργου αναλύεται χρησιμοποιώντας τον *Source Code Parser*. Ο Source Code Parser είναι ένας αναλυτής πηγαίου κώδικα που λαμβάνει ως είσοδο ένα αρχείο πηγαίου κώδικα και εξάγει την *υπογραφή (signature)* του. Η υπογραφή ενός αρχείου αναπαρίσταται ως ένα έγγραφο JSON που περιέχει όλα τα στοιχεία που ορίζονται ή χρησιμοποιούνται στο αρχείο, π.χ. ονόματα κλάσεων, ονόματα μεθόδων κ.λπ.

Τα έγγραφα JSON προστίθενται στο ευρετήριο ως έργα και αρχεία. Τα σχήματα (schemas) των πεδίων υπαγορεύονται από τα *mappings* του Elasticsearch. Η AGORA έχει δύο mappings, ένα για τα έγγραφα του ευρετηρίου έργων και ένα για τα έγγραφα του ευρετηρίου αρχείων. Όταν προστίθεται ένα έγγραφο στο ευρετήριο, τα πεδία του (εγγραφές JSON) αναλύονται χρησιμοποιώντας έναν *αναλυτή (analyzer)*. Αυτό το βήμα είναι σημαντικό επειδή επιτρέπει στο Elasticsearch να διεκπεραιώνει γρήγορα ερωτήματα πάνω στα πεδία που έχουν αναλυθεί.

Η AGORA μπορεί προφανώς να αποθηκεύσει στο ευρετήριο έργα γραμμένα σε διάφορες γλώσσες προγραμματισμού. Συγκεκριμένα, η προσθήκη αρχείων πηγαίου κώδικα ενός έργου γραμμένου σε μια συγκεκριμένη γλώσσα προγραμματισμού απαιτεί τρία βήματα: (α) την ανάπτυξη ενός αναλυτή πηγαίου κώδικα (source code parser) για την εξαγωγή πληροφοριών από τον πηγαίο κώδικα, (β) τη δημιουργία κατάλληλων αναλυτών (analyzers) που να ανταποκρίνονται στις προδιαγραφές της γλώσσας προγραμματισμού, και (γ) τη σχεδίαση μιας δομής που θα αποθηκεύει τα στοιχεία του πηγαίου κώδικα που περιλαμβάνονται σε κάθε αρχείο. Χωρίς βλάβη της γενικότητας, στις επόμενες παραγράφους δείχνουμε πώς εκτελούνται αυτά τα βήματα για τη γλώσσα προγραμματισμού Java.

Ο αναλυτής πηγαίου κώδικα για έργα Java κατασκευάστηκε χρησιμοποιώντας το Java Tree Compiler API [179] που παρέχεται από την Oracle. Λαμβάνει ως είσοδο ένα ή περισσότερα αρχεία και διασχίζει το *Αφηρημένο Συντακτικό Δένδρο (Abstract Syntax Tree - AST)* κάθε αρχείου για να εξάγει όλα τα στοιχεία Java που ορίζονται ή χρησιμοποιούνται στο αρχείο: πακέτα (packages), δηλώσεις βιβλιοθηκών (imports), κλάσεις (classes), μεθόδους (methods), μεταβλητές (variables), κ.λπ. Επιπλέον, συλλέγονται πληροφορίες σχετικά με αυτά τα στοιχεία, όπως το πεδίο (scope) των μεταβλητών, ο τύπος επιστροφής (return type) των μεθόδων κ.λπ. Όσον αφορά τους αναλυτές (analyzers) του Elasticsearch, αυτοί πρέπει επίσης να ανταποκρίνονται στα χαρακτηριστικά των αρχείων Java (π.χ. να υποστηρίζουν το διαχωρισμό όρων σε camel case). Οι αναλυτές που αναπτύξαμε για τη γλώσσα Java παρουσιάζονται στην ενότητα 5.3.2, ενώ η ενότητα 5.3.3 δείχνει τα mappings που χρησιμοποιούνται για την αποθήκευση έργων και αρχείων.

### 5.3.2 Αναλυτές

Η ανάλυση των πεδίων είναι ένα μία από τις πιο σημαντικές διεργασίες των συστημάτων ευρετηρίασης, καθώς επηρεάζει την ποιότητα των αποτελεσμάτων που επιστρέφονται και την αποδοτικότητα του συστήματος. Στο Elasticsearch, οι αναλυτές περιλαμβάνουν έναν *τμηματοποιητή (tokenizer)* είτε χωρίς κάποιο φίλτρο, είτε με ένα ή περισσότερα *φίλτρα (token filters)*.

Για κάθε πεδίο, ο τμηματοποιητής χωρίζει το περιεχόμενό του σε όρους (tokens), και τα φίλτρα δέχονται μια ακολουθία από όρους, τους οποίους μπορούν να τροποποιήσουν, να διαγράψουν ή ακόμα και να προσθέσουν. Για παράδειγμα, σε ένα σενάριο μιας μηχανής αναζήτησης κειμένου, ένας τμηματοποιητής θα χώριζε ένα έγγραφο σε λέξεις σύμφωνα με τα διαστήματα και τα σημεία στίξης και στη συνέχεια θα εφάρμοζε πιθανώς ένα φίλτρο που μετατρέπει όλες τις λέξεις σε πεζά γράμματα (lowercase filter) και ένα άλλο που θα αφαιρούσε τα stopwords (stopword filter). Στη συνέχεια, όλα τα πεδία θα αποθηκεύονταν στο ευρετήριο ως λέξεις, επιτρέποντας έτσι τη διεξαγωγή ερωτημάτων με ταχύτητα και ακρίβεια. Παρόμοια λογική εφαρμόζεται και για την ανάλυση πηγαίου κώδικα.

Το Elasticsearch έχει διάφορους προεγκατεστημένους αναλυτές (default analyzers) ενώ επιτρέπει και τη δημιουργία προσαρμοσμένων αναλυτών (custom analyzers), χρησιμοποιώντας τους τμηματοποιητές που παρέχονται και τα αντίστοιχα φίλτρα. Για τη δική μας περίπτωση, χρησιμοποιήσαμε τον αναλυτή *Standard* που παρέχεται από το Elasticsearch και δημιουργήσαμε επίσης τρεις προσαρμοσμένους αναλυτές: τον αναλυτή *Filepath*, ο οποίος είναι ένας αναλυτής για τις διαδρομές αρχείων, τον αναλυτή *CamelCase* που αναλύει όρους τύπου camel case, και τον αναλυτή *Javafile* για την ανάλυση αρχείων java.

### 5.3.2.1 Αναλυτής Standard

Ο αναλυτής Standard αρχικά χρησιμοποιεί τον τμηματοποιητή Standard (Standard tokenizer) του Elasticsearch για να διαχωρίσει το κείμενο σε όρους σύμφωνα με το Unicode Standard Annex #29 [180], δηλαδή διαχωρίζοντας στα κενά, στα σημεία στίξης, ειδικούς χαρακτήρες (special characters), κ.λπ. Κατόπιν, οι όροι μετατρέπονται σε πεζά χρησιμοποιώντας ένα φίλτρο Lower Case (Lower Case Token Filter) και οι κοινότυπες λέξεις αφαιρούνται χρησιμοποιώντας ένα φίλτρο Stop (Stop Token Filter)<sup>8</sup>. Ο αναλυτής Standard είναι ο προεπιλεγμένος αναλυτής του Elasticsearch, καθώς είναι αποτελεσματικός για τα περισσότερα πεδία.

### 5.3.2.2 Αναλυτής Filepath

Ο αναλυτής Filepath χρησιμοποιείται για την ανάλυση διαδρομών αρχείων ή ηλεκτρονικών διευθύνσεων. Αποτελείται από τον τμηματοποιητή Path Hierarchy και το φίλτρο Lower Case του Elasticsearch. Ο αναλυτής δέχεται είσοδο της μορφής “/path/to/file” και εξάγει τους όρους “/path”, “/path/to” και “/path/to/file”.

### 5.3.2.3 Αναλυτής CamelCase

Παρόλο που ο αναλυτής Standard είναι αποτελεσματικός για τα περισσότερα πεδία, τα πεδία που προέρχονται από αρχεία πηγαίου κώδικα συνήθως ανταποκρίνονται στα πρότυπα της γλώσσας. Δεδομένου ότι η γλώσσα που μας ενδιαφέρει είναι η Java, δημιουργήσαμε έναν αναλυτή που διαχωρίζει κείμενο τύπου camel case. Εφαρμόζουμε έναν τμηματοποιητή τύπου *pattern*, ο οποίος χρησιμοποιεί μια κανονική έκφραση για να διαχωρίσει το κείμενο σε όρους. Για κείμενο τύπου camel case, χρησιμοποιούμε τον τμηματοποιητή που παρέχεται στο [181]. Η σχετική κανονική έκφραση φαίνεται στο Σχήμα 5.2.

<sup>8</sup>Το Elasticsearch υποστηρίζει την αφαίρεση stopwords για αρκετές γλώσσες. Στην περίπτωσή μας, η προεπιλεγμένη επιλογή των αγγλικών είναι η πιο κατάλληλη.

---

```

#Match
([^\p{L}\d]+)           #non-letters and numbers,
| (?<=\D)(?=\d)       #or non-number then number,
| (?<=\d)(?=\D)       #or number then non-number,
| (?<=[\p{L} && [^\p{Lu}]]) #or lower case
#followed by
(?:\p{Lu})             #upper case,
| (?<=\p{Lu})         #or upper case
(?:\p{Lu}[\p{L}&&[^\p{Lu}]]) #then upper and lower case

```

---

Σχήμα 5.2: Κανονική έκφραση για τον αναλυτή CamelCase

Μετά το διαχωρισμό του κειμένου σε όρους, ο αναλυτής CamelCase μετατρέπει τους όρους σε πεζά χρησιμοποιώντας το φίλτρο Lower Case του Elasticsearch. Σε αυτήν την περίπτωση, δεν αφαιρούνται stopwords.

#### 5.3.2.4 Αναλυτής Javafile

Ο αναλυτής Javafile χρησιμοποιείται για την ανάλυση αρχείων πηγαίου κώδικα java. Ο αναλυτής αποτελείται από έναν τμηματοποιητή Standard και ένα φίλτρο Stop του Elasticsearch. Το κείμενο διαχωρίζεται σε όρους και οι πιο κοινότεροι όροι της Java αφαιρούνται. Τα Java stopwords που αφαιρούνται φαίνονται στο Σχήμα 5.3.

---

```

void, if, else, do, for, switch, case, break, while, default, true, false, return,
this, null, instanceof, try, catch, throws, throw, finally, super, assert, const,
package, import, implements, extends, synchronized, continue, interface,
class, public, private, protected, final, static, abstract, native, transient

```

---

Σχήμα 5.3: Λίστα από Java stopwords

Η λίστα από stopwords του Σχήματος 5.3 περιέχει όλους τους όρους που είναι καλύτερο να αφαιρεθούν κατά την αναζήτηση για μικρά τμήματα κώδικα (snippets). Σημειώστε, ωστόσο, ότι διατηρούνται οι τύποι αντικειμένων (π.χ. “int”, “float”, “double”) καθώς είναι χρήσιμοι για την εύρεση παρόμοιων snippets. Ο τύπος “void”, όμως, αφαιρείται καθώς είναι τόσο συχνός που πρακτικά δεν προσθέτει κάποια αξία στην τελική σύγκριση.

#### 5.3.3 Ευρετηριοποίηση Εγγράφων

Όπως αναφέρθηκε στην ενότητα 5.3.1, το Elasticsearch αποθηκεύει τα δεδομένα ως έγγραφα (documents). Ο τρόπος που αποθηκεύεται κάθε έγγραφο (δηλαδή το schema του) και ο τρόπος που αναλύεται κάθε πεδίο (field) από τους αναλυτές της ενότητας 5.3.2 καθορίζονται από το *mapping* του. Για την AGORA και για τη γλώσσα Java, ορίζουμε δύο mappings, ένα για έγγραφα τύπου project και ένα για έγγραφα τύπου file. Επιπλέον, συνδέουμε τα δύο



αυτά mappings μεταξύ τους σε μια σχέση γονέα-τέκνου (*parent-child*), όπου κάθε αρχείο αντιστοιχεί σε ένα έγγραφο γονέα που είναι έργο. Τα mappings αναλύονται στις επόμενες υποενότητες.

### 5.3.3.1 Ευρετηριοποίηση Έργων

Τα έργα προσδιορίζονται μοναδικά στην AGORA με το όνομά τους και το όνομα του δημιουργού τους (GitHub account), ενωμένα με μια κάθετο (/). Έτσι, το πεδίο *\_id* του mapping παίρνει τιμές της μορφής “user/repo”. Τα υπόλοιπα πεδία των εγγράφων που αντιστοιχούν σε έργα λογισμικού φαίνονται στον Πίνακα 5.3, μαζί με ένα παράδειγμα για το egit plugin του Eclipse.

Πίνακας 5.3: Mapping των Έργων στο Ευρετήριο της AGORA

Όνομα	Τύπος	Αναλυτής	Παράδειγμα
fullname	string	-	eclipse/egit-github
default_branch	string	-	master
url	string	-	https://api.github.com/repos/eclipse/egit-github
git_url	string	-	git://github.com/eclipse/egit-github.git
user	string	Standard	eclipse
name	string	Standard	egit-github

Τα περισσότερα από τα πεδία του έργου είναι χρήσιμο να αποθηκεύονται, ωστόσο η ανάλυση όλων των πεδίων δεν προσφέρει κάποια αξία. Συνεπώς, τα URLs (“url” και “git\_url” του GitHub API) και το πεδίο “default\_branch” δεν αναλύονται. Το πεδίο “fullname”, ωστόσο, που έχει την ίδια πληροφορία με το “\_id”, αναλύεται με τον αναλυτή Standard ώστε να μπορεί να συμμετέχει σε ερωτήματα. Επιπρόσθετα, σε ερωτήματα συμμετέχουν και τα πεδία “user” και “name”, που αντιστοιχούν στον κάτοχο και στο όνομα του έργου, και τα οποία επίσης αναλύονται με τον αναλυτή Standard.

### 5.3.3.2 Ευρετηριοποίηση Αρχείων

Όπως ήδη αναφέρθηκε, έχουμε συνδέσει τα mappings των έργων και των αρχείων με τη σχέση γονέα-τέκνου που παρέχεται από το Elasticsearch. Κάθε αρχείο έχει ένα πεδίο “\_parent”, που είναι ένας σύνδεσμος στο αντίστοιχο έγγραφο του έργου. Το πεδίο αυτό χρησιμοποιείται για τη σύνταξη ερωτημάτων τύπου *has\_child* (βλέπε υποενότητα 5.4.2.3) ή για την ανάκτηση όλων των αρχείων ενός έργου. Το “\_id” κάθε αρχείου πρέπει επίσης να είναι μοναδικό· σχηματίζεται με την πλήρη διαδρομή του αρχείου συμπεριλαμβάνοντας επίσης το έργο, δηλαδή έχει τη μορφή “user/project/path/to/file”. Ο Πίνακας 5.4 περιέχει τα υπόλοιπα πεδία, καθώς και ένα παράδειγμα για το αρχείο DateFormatter.java του egit plugin του Eclipse του Πίνακα 5.3.

Όπως και στα έργα, το “\_id” επιπλέον αντιγράφεται στο πεδίο “fullpathname” που αναλύεται με τον αναλυτή Filepath ώστε να συμμετέχει σε ερωτήματα. Το πεδίο “path”, που αναφέρεται στη διαδρομή του αρχείου χωρίς το πρόθεμα του έργου, επίσης αναλύεται με

Πίνακας 5.4: Mapping των Αρχείων στο Ευρετήριο της AGORA

Όνομα	Τύπος	Αναλυτής	Παράδειγμα
fullpathname	string	Filepath	eclipse/egit-github/org.../DateFormatter.java
path	string	Filepath	org.../DateFormatter.java
name	string	Standard	DateFormatter.java
project	string	Standard	eclipse/egit-github
mode	string	-	100644
sha	string	-	e51f1efca2155fc7af3...
type	string	-	blob
extension	string	-	java
url	string	-	https://api.github.com/repos/eclipse/.../blobs/...
content	string	Standard	...
analyzedcontent	string	Javafile	...
code			
package	string	CamelCase	org.eclipse...client
imports	string	CamelCase	[java.util.TimeZone, java.util.Date, ...]
class	see Table 5.5		see Table 5.5
otherclasses	see Table 5.5		[]

τον αναλυτή Filepath. Το όνομα κάθε αρχείου (“name”) και το όνομα του έργου στο οποίο ανήκει (“project”) αναλύονται με τον αναλυτή Standard<sup>9</sup>.

Όπως και προηγουμένως, κάποια πεδία δεν είναι απαραίτητο να συμμετέχουν σε ερωτήματα, αλλά είναι ίσως χρήσιμο να αποθηκεύονται. Τα πεδία “mode” και “type”, που αναφέρονται στον git mode του αρχείου<sup>10</sup> και στο git type του αρχείου<sup>11</sup>, καθώς επίσης και η κατάληξη (“extension”) και η ηλεκτρονική διεύθυνση (“url”) του κάθε αρχείου δεν αναλύονται. Το πεδίο “sha” επίσης δεν αναλύεται. Ωστόσο, πρέπει να αναφερθεί ότι το “sha” είναι ένα αρκετά σημαντικό πεδίο, καθώς αποθηκεύει ένα id που αναφέρεται στο περιεχόμενο του αρχείου. Το πεδίο αυτό χρησιμοποιείται για να ελεγχθεί αν ένα αρχείο χρειάζεται να ενημερωθεί όταν ενημερώνεται κάποιο έργο.

Επιπλέον, καθώς τα αρχεία περιέχουν κώδικα, αρκετά σημαντικά πεδία πρέπει να οριστούν για τη δημιουργία ερωτημάτων. Αρχικά, το περιεχόμενο κάθε αρχείου αναλύεται χρησιμοποιώντας δύο αναλυτές, τον αναλυτή Standard και τον αναλυτή Javafile, και τα αποτελέσματα αποθηκεύονται στα πεδία “content” και “analyzedcontent” αντίστοιχα. Αυτή η ανάλυση επιτρέπει την εφαρμογή αυστηρής σύγκρισης κειμένου (strict text matching) στο πεδίο “content” και πιο χαλαρής σύγκρισης κειμένου (loose text matching) στο πεδίο “analyzedcontent”, καθώς τα stopwords στη δεύτερη περίπτωση αφαιρούνται.

Στη συνέχεια, οι τιμές των υποπεδίων του πεδίου “code” εξάγονται χρησιμοποιώντας τον Source Code Parser (βλέπε Σχήμα 5.1). Δύο τέτοια πεδία που υπάρχουν σε αρχεία java είναι τα πεδία “package” και “imports” (τα οποία αποθηκεύονται ως λίστες, καθώς μπορούν να έχουν περισσότερες από μία τιμές). Τα δύο πεδία αναλύονται χρησιμοποιώντας τον αναλυτή

<sup>9</sup>Σημειώστε πως ο αναλυτής Standard αρχικά διαχωρίζει το όνομα σε δύο τμήματα, ένα με την κατάληξη (extension) και ένα χωρίς, καθώς εκτελεί διαχωρισμούς με βάση τη στίξη.

<sup>10</sup>Η τιμή του git mode του αρχείου είναι μία από τις 040000, 100644, 100664, 100755, 120000 και 160000, που αντιστοιχούν στα git modes directory, regular non-executable file, regular non-executable group-writable file, regular executable file, symbolic link και gitlink αντίστοιχα.

<sup>11</sup>To git type του αρχείου είναι ένα από τα blob, tree, commit και tag.

CamelCase, καθώς τόσο το πακέτο ενός αρχείου java όσο και οι δηλώσεις βιβλιοθηκών του ακολουθούν τις συμβάσεις της γλώσσας Java (Java coding conventions). Καθώς κάθε αρχείο java έχει μια δημόσια (public) κλάση και πιθανώς άλλες μη δημόσιες κλάσεις, το mapping επίσης περιέχει τα πεδία “class” και “otherclasses”.

Σημειώστε ότι το πεδίο “otherclasses” είναι μια λίστα τύπου “nested”. Η λίστα τύπου “nested” είναι ένας ιδιαίτερος τύπος πεδίου που παρέχεται από το Elasticsearch που επιτρέπει τη δημιουργία ερωτημάτων με συνεκτικό τρόπο. Συγκεκριμένα, έχοντας για παράδειγμα μια κλάση με δύο υποπεδία “name” και “type” (όπου το “type” μπορεί να είναι είτε class είτε interface), αν θέλουμε να πραγματοποιήσουμε αναζήτηση για μια κλάση με ένα συγκεκριμένο όνομα και ταυτόχρονα ένα συγκεκριμένο τύπο, τότε η λίστα πρέπει να είναι τύπου “nested”. Αλλιώς η αναζήτηση μπορεί να επιστρέψει ένα έγγραφο που έχει δύο διαφορετικές κλάσεις, μία που να έχει μόνο το ζητούμενο όνομα και μία που να έχει μόνο τον ζητούμενο τύπο (περισσότερα για τις αναζητήσεις τέτοιου τύπου στην υποενότητα 5.4.2.1). Τα υποπεδία μιας κλάσης φαίνονται στον Πίνακα 5.5, μαζί με ένα παράδειγμα για το αρχείο DateFormatter.java του Πίνακα 5.4.

Πίνακας 5.5: Εσωτερικό Mapping Κλάσης για τα Αρχεία της AGORA

Όνομα	Τύπος	Αναλυτής	Παράδειγμα
extends	string	CamelCase	-
implements	array	CamelCase	[JsonSerializer<Date>, ...]
methods	nested		
modifiers	array	-	[public]
name	string	CamelCase	serialize
parameters	nested		
name	string	CamelCase	date
type	string	CamelCase	Date
returntype	string	CamelCase	JsonElement
throws	array	CamelCase	[]
modifiers	array	-	[public]
name	string	CamelCase	DateFormatter
type	string	CamelCase	class
variables	nested		
modifiers	array	-	[private, final]
name	string	CamelCase	formats
type	string	CamelCase	DateFormat[]
innerclasses	(1-level)	recursive of the class mapping	

Όπως φαίνεται στον Πίνακα 5.5, τα πεδία “extends” και “implements” αναλύονται με τον αναλυτή CamelCase. Σημειώστε ότι το πεδίο “implements” είναι του τύπου “array”, καθώς στη Java μια κλάση μπορεί να υλοποιεί περισσότερα του ενός interfaces. Μια κλάση Java μπορεί επίσης να έχει πολλές μεθόδους, με κάθε μία από αυτές να έχει το δικό της όνομα, τις δικές της παραμέτρους, κ.λπ. Έτσι, το πεδίο “methods” είναι του τύπου “nested” ούτως ώστε να υποστηρίζονται αυτόνομα ερωτήματα για μεθόδους, ενώ ταυτόχρονα να επιτρέπονται και ερωτήματα με πολλαπλές μεθόδους (multi-method queries). Το όνομα (“name”) και ο τύπος επιστροφής (“returntype”) μιας μεθόδου αναλύονται με τον αναλυτή CamelCase, ενώ τα exceptions (“throws”) αποθηκεύονται σε πίνακες που επίσης αναλύονται με τον αναλυτή CamelCase. Το πεδίο “modifiers” είναι επίσης τύπου “array”, ωστόσο δεν αναλύεται καθώς

οι τιμές του είναι συγκεκριμένες (μία εκ των `public`, `private`, `final`, κ.λπ.). Στο υποπεδίο “parameters” αποθηκεύονται οι παράμετροι της μεθόδου, όπου συμπεριλαμβάνονται πεδία για το όνομα (“name”) και τον τύπο (“type”) κάθε παραμέτρου, τα οποία αναλύονται με τον αναλυτή CamelCase<sup>12</sup>.

Το πεδίο “modifiers” της κλάσης επίσης δεν αναλύεται, ενώ το όνομα της κλάσης αναλύεται. Ο τύπος της κλάσης (“type”) επίσης δεν αναλύεται καθώς οι τιμές του είναι συγκεκριμένες (μία εκ των `class`, `interface`, `type`). Στο πεδίο “variables” αποθηκεύονται οι μεταβλητές των κλάσεων σε μορφή “nested”, ενώ τα υποπεδία που αντιστοιχούν στο όνομα (“name”) και τον τύπο (“type”) κάθε μεταβλητής αναλύονται με τον αναλυτή CamelCase. Τέλος, οι εσωτερικές κλάσεις (inner classes) έχουν το ίδιο mapping με αυτό του Πίνακα 5.5, ωστόσο χωρίς να εμπεριέχεται το πεδίο “innerclasses”, επιτρέποντας έτσι μόνον ένα επίπεδο από εσωτερικές κλάσεις. Η αποθήκευση περισσότερων επιπέδων δε θα ήταν αποτελεσματική.

### 5.3.4 Κατάταξη Αποτελεσμάτων

Χρησιμοποιούμε το σχήμα βαθμολόγησης του Elasticsearch [182] για να κατατάξουμε τα αποτελέσματα ενός ερωτήματος σύμφωνα με τη σχετικότητά τους. Κατά τη δημιουργία ενός ερωτήματος, μπορεί κανείς να επιλέξει οποιοδήποτε από τα πεδία που ορίζονται στα mappings της AGORA (βλέπε ενότητα 5.3.3) ή και συνδυασμούς μεταξύ τους. Όπως ήδη αναφέρθηκε, όλα τα πεδία που αναλύθηκαν (χρησιμοποιώντας έναν από τους τέσσερις αναλυτές) εισάγονται στο ευρετήριο ως διαχωρισμένοι όροι (tokens), χωρίς να λαμβάνονται υπόψη τα σημεία στίξης και τα stopwords.

Αρχικά, το κείμενο του ερωτήματος αναλύεται χρησιμοποιώντας τον αναλυτή που αντιστοιχεί στο πεδίο που αυτό απευθύνεται. Στη συνέχεια, η ομοιότητα μεταξύ των όρων του ερωτήματος και των όρων του πεδίου του εγγράφου υπολογίζεται χρησιμοποιώντας το μοντέλο διανυσματικού χώρου (vector space model) [72]. Στο μοντέλο διανυσματικού χώρου, κάθε όρος είναι μια διάσταση (dimension), έτσι κάθε έγγραφο (ή το ερώτημα) μπορεί να αναπαρασταθεί ως ένα διάνυσμα (vector). Για μια συλλογή από έγγραφα  $D$ , η τιμή σχετικότητας (relevance weight) ενός όρου  $t$  σε ένα έγγραφο  $d$  υπολογίζεται χρησιμοποιώντας το *tf-idf* (term frequency-inverse document frequency) του όρου ως προς το έγγραφο:

$$w_{t,d} = \sqrt{f(t,d)} \cdot \left( 1 + \log \frac{|D|}{|d \in D : t \in d| + 1} \right) \cdot \frac{1}{\sqrt{|d|}} \quad (5.1)$$

Το πρώτο μέρος του παραπάνω γινομένου είναι η συχνότητα του όρου στο έγγραφο (term frequency - *tf*), που υπολογίζεται ως η τετραγωνική ρίζα του πλήθους των φορών που ο όρος εμφανίζεται στο έγγραφο  $f(t,d)$ . Το δεύτερο μέρος του γινομένου είναι η αντίστροφη συχνότητα εγγράφων (inverse document frequency - *idf*), που υπολογίζεται ως ο λογάριθμος του πηλίκου του συνολικού πλήθους των εγγράφων  $|D|$  με το συνολικό πλήθος των εγγράφων που περιέχουν τον όρο  $|d \in D : t \in d|$ . Τέλος, το τρίτο μέρος του γινομένου εφαρμόζει μια κανονικοποίηση στο μήκος του πεδίου (field-length normalization) που ελέγχει το πλήθος των όρων στο έγγραφο  $|d|$ , και ουσιαστικά ενισχύει τα μικρότερα έγγραφα. Στην περίπτωση ενός πεδίου που δεν έχει αναλυθεί με κάποιο αναλυτή, το περιεχόμενο του πεδίου πρέπει να προσδιοριστεί επακριβώς και τότε το σύστημα θα επιστρέψει μια δυαδική

<sup>12</sup>Ο αναλυτής CamelCase είναι ιδιαίτερα αποτελεσματικός για πεδία που έχουν τύπους της Java: οι τύποι συνηθίζεται να είναι σε camel case, ενώ τα primitives δεν επηρεάζονται από τον αναλυτή, π.χ. η λέξη “float” έχει ως αποτέλεσμα τον όρο “float”.

απόφαση, είτε ότι υπάρχει είτε ότι δεν υπάρχει απόλυτη ομοιότητα (δηλαδή το αποτέλεσμα της εξίσωσης (5.1) θα είναι είτε 1 είτε 0. Τα πεδία τύπου “array” αντιμετωπίζονται ως ήδη διαχωρισμένοι όροι.

Επομένως, τα διανύσματα εγγράφων αναπαρίστανται χρησιμοποιώντας τις τιμές βαρών (term weights) που ορίζονται από την παραπάνω εξίσωση. Η τελική τιμή ομοιότητας μεταξύ ενός διανύσματος ερωτήματος  $q$  και ενός διανύσματος εγγράφου  $d$  υπολογίζεται χρησιμοποιώντας την ομοιότητα συνημιτόνου (cosine similarity):

$$score(q, d) = \frac{q \cdot d}{|q| \cdot |d|} = \frac{\sum_1^n w_{t_i,q} \cdot w_{t_i,d}}{\sum_1^n w_{t_i,q}^2 \cdot \sum_1^n w_{t_i,d}^2} \quad (5.2)$$

όπου  $t_i$  είναι ο  $i$ -ος όρος του ερωτήματος και  $n$  είναι το συνολικό πλήθος των όρων του ερωτήματος.

Τέλος, όταν ένα ερώτημα εκτείνεται σε περισσότερα από ένα πεδία, τότε εφαρμόζεται μια λογική σχέση τύπου “and”. Αν υποθέσουμε ότι το αποτέλεσμα της εξίσωσης (5.2) υπολογίζεται για κάθε πεδίο, δηλαδή ότι η τιμή  $score(q, f)$  αναφέρεται στην τιμή για το ερώτημα  $q$  και το πεδίο  $f$ , τότε η συνολική βαθμολογία για κάθε έγγραφο  $d$  υπολογίζεται από την παρακάτω συνάρτηση:

$$score(q, d) = norm(q) \cdot coord(q, d) \cdot \sum_{t \in q} score(q, f) \quad (5.3)$$

όπου  $norm(q)$  είναι ο παράγοντας κανονικοποίησης του ερωτήματος (query normalization factor), που ορίζεται ως το αντίστροφο της τετραγωνικής ρίζας του αθροίσματος των τετραγώνων των  $idf$  κάθε όρου στο ερώτημα, δηλαδή  $1/\sqrt{\sum_1^n w_{t_i,q}}$ , και  $coord(q, d)$  είναι ο παράγοντας συντονισμού (coordination factor), που ορίζεται ως το πηλίκο του πλήθους των όρων του ερωτήματος που βρίσκονται στο έγγραφο με το συνολικό πλήθος των όρων του ερωτήματος, δηλαδή  $|t \in q : t \in d|/|q|$ .

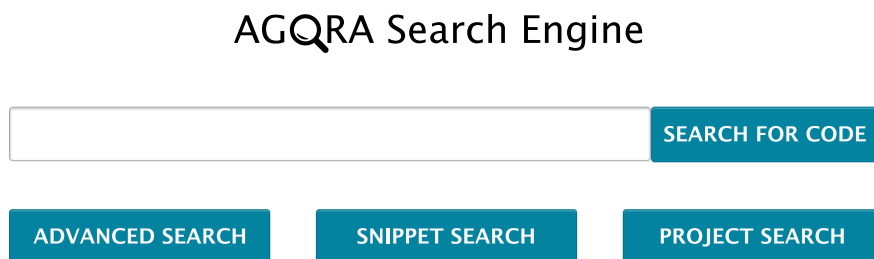
## 5.4 Αναζήτηση Κώδικα

### 5.4.1 Δυνατότητες Αναζήτησης της AGORA

Η AGORA έχει ένα ανοικτό RESTful API καθώς και μια διαδικτυακή εφαρμογή (web application), που φαίνεται στο Σχήμα 5.4. Παρόλο που το API της AGORA επιτρέπει την εκτέλεση όλων των πιθανών ερωτημάτων που υποστηρίζονται από το Elasticsearch, στην πράξη μπορούμε να διακρίνουμε 4 τύπους αναζητήσεων, οι οποίοι είναι και αυτοί που υποστηρίζονται από τη διαδικτυακή εφαρμογή: την απλή αναζήτηση για κώδικα (*Simple Search*), την προηγμένη αναζήτηση (*Advanced Search*) που αφορά ερωτήματα με βάση τη σύνταξη, την αναζήτηση για μικρά τμήματα κώδικα (*Snippet Search*) και την αναζήτηση για έργα λογισμικού (*Project Search*).

Η απλή αναζήτηση (*Simple Search*) πραγματοποιείται με το πεδίο “\_all”, που επιτρέπει την αναζήτηση μέσα σε όλα τα πεδία ενός αρχείου (π.χ. όνομα αρχείου, περιεχόμενο κ.λπ.). Αυτός ο τύπος αναζήτησης υποστηρίζει επίσης τη χρήση κανονικών εκφράσεων.

Η λειτουργία προηγμένης αναζήτησης (*Advanced Search*) επιτρέπει την αναζήτηση μιας κλάσης με συγκεκριμένα χαρακτηριστικά, όπως δηλώσεις βιβλιοθηκών (imports), ονόματα και τύπους μεταβλητών, ονόματα και τύπους επιστροφής μεθόδων, ονόματα και τύπους των παραμέτρων των μεθόδων κ.λπ.



© Copyright AGORA 2017

[API](#) | [Help](#) | [About](#) | [Disclaimer](#)

Σχήμα 5.4: Αρχική Σελίδα της AGORA

Με αυτόν τον τύπο αναζήτησης, οι προγραμματιστές μπορούν εύκολα να βρουν επαναχρησιμοποιήσιμα τμήματα κώδικα με συγκεκριμένες εισόδους/εξόδους για να τα ενσωματώσουν στον πηγαίο κώδικά τους ή ακόμα και να ανακαλύψουν τμήματα κώδικα που θα τους βοηθήσουν να καταλάβουν τον τρόπο χρήσης μιας βιβλιοθήκη ή ενός framework.

Η λειτουργία αναζήτησης μικρών τμημάτων κώδικα (*Snippet Search*) δέχεται ως είσοδο ένα snippet και επιστρέφει παρόμοιο κώδικα. Παρόλο που τα ερωτήματα με βάση τη σύνταξη (*syntax-aware queries*) επιστρέφουν ικανοποιητικά αποτελέσματα σε επίπεδο κλάσεων ή μεθόδων, στην περίπτωση που θέλει κανείς να βρει παραδείγματα χρήσης ενός API (*API usage examples*), η αναζήτηση για snippets είναι αρκετά χρήσιμη, καθώς τα περισσότερα σενάρια χρήσης API περιλαμβάνουν κλήσεις μεθόδων σε λίγες γραμμές κώδικα.

Τέλος, η λειτουργία *Project Search* επιτρέπει την αναζήτηση έργων λογισμικού που περιλαμβάνουν συγκεκριμένα τμήματα κώδικα ή ακολουθούν μια συγκεκριμένη δομή. Έτσι, μπορεί κανείς να ψάξει για έργα που ακολουθούν κάποια πρότυπα σχεδίασης [183] ή κάποια αρχιτεκτονική (π.χ. MVC).

Κατά τη στιγμή της συγγραφής, το ευρετήριο της AGORA περιλαμβάνει τα 3000 δημοφιλέστερα έργα του GitHub, όπου η δημοφιλία ενός έργου καθορίζεται από τον αριθμό των GitHub stars. Μετά την αφαίρεση των αποθετηρίων που έχουν γίνει fork και των μη έγκυρων αποθετηρίων (π.χ. χωρίς κώδικα java), απομένουν 2709 έργα. Σχετικές μελέτες έχουν δείξει ότι τα έργα με υψηλή κατάταξη (δηλαδή με μεγάλο αριθμό stars/forks) είναι επίσης έργα υψηλής ποιότητας (quality) [149], έχουν επαρκή τεκμηρίωση (documentation) και αρχεία readme [184, 185], και περιλαμβάνουν συχνά εκδόσεις συντήρησης (maintenance releases) [186]. Αξιολογώντας επιπλέον την επαναχρησιμοποιησιμότητα των τμημάτων λογισμικού με μετρικές στατικής ανάλυσης [148, 150], μπορούμε να ισχυριστούμε ότι τα έργα αυτά περιλαμβάνουν επαναχρησιμοποιήσιμο κώδικα· ενδεικτικά αναφέρουμε ότι περίπου το 90% του πηγαίου κώδικα των 100 δημοφιλέστερων αποθετηρίων του GitHub αποτελεί κώδικα που είναι επαναχρησιμοποιήσιμος και υψηλής ποιότητας [150]. Λαμβάνοντας επίσης υπόψη ότι τα έργα είναι πολύ διαφορετικά, και περιλαμβάνουν γνωστά έργα μεγάλης κλίμακας (όπως π.χ. το Spring framework ή τη γλώσσα προγραμματισμού Clojure) καθώς και έργα μικρότερου μεγέθους (π.χ. εφαρμογές Android), συμπεραίνουμε ότι το ευρετήριό μας είναι σε θέση να παρέχει μια ποικιλία από επαναχρησιμοποιήσιμες λύσεις.

Στην ενότητα 5.4.2 παρέχουμε παραδείγματα ερωτημάτων που εκτελούνται στο ευρετήριο της AGORA, υποδεικνύοντας τα αντίστοιχα σενάρια αναζήτησης και επεξηγώντας τη σύνταξη που απαιτείται για αναζήτηση στο API της AGORA. Κατόπιν, στην ενότητα 5.4.3

παρέχουμε ένα παράδειγμα σεναρίου χρήσης της διαδικτυακής εφαρμογής της AGORA, για την κατασκευή ενός έργου με τρία συνδεδεμένα τμήματα κώδικα.

## 5.4.2 Σενάρια Αναζήτησης της AGORA

### 5.4.2.1 Αναζήτηση για επαναχρησιμοποιήσιμα τμήματα

Ένα από τα πιο συνηθισμένα σενάρια που μελετάται από την τρέχουσα βιβλιογραφία [90, 96–98] είναι η αναζήτηση ενός επαναχρησιμοποιήσιμου τμήματος λογισμικού (reusable software component). Τα τμήματα λογισμικού μπορούν να θεωρηθούν ως μεμονωμένες οντότητες με συγκεκριμένες εισόδους/εξόδους που είναι κατάλληλες για επαναχρησιμοποίηση. Το πιο κλασικό παράδειγμα τμήματος στη γλώσσα Java είναι η κλάση (class). Έτσι, ένα απλό σενάριο αναζήτησης περιλαμβάνει την εύρεση μιας κλάσης με συγκεκριμένη λειτουργικότητα. Χρησιμοποιώντας το API της AGORA, η εύρεση μιας κλάσης με συγκεκριμένες μεθόδους και/ή μεταβλητές είναι απλή. Για παράδειγμα, υποθέστε ότι ένας προγραμματιστής θέλει να χρησιμοποιήσει μια δομή δεδομένων παρόμοια με μια στοίβα (stack) με τη γνωστή λειτουργικότητα εισαγωγής στοιχείου (push) και αφαίρεσης στοιχείου (pop). Τότε το ερώτημα διαμορφώνεται όπως φαίνεται στο Σχήμα 5.5.

---

```

{
  "query": {"bool": {"should": [
    {"match": {"code.class.name": "stack"}},
    {"nested": {
      "path": "code.class.methods",
      "query": {"bool": {"should": [
        {"match": {"code.class.methods.name": "push"}},
        {"term": {"code.class.methods.returntype": "void"}}
      ]}}
    ]}},
    {"nested": {
      "path": "code.class.methods",
      "query": {"bool": {"should": [
        {"match": {"code.class.methods.name": "pop"}},
        {"term": {"code.class.methods.returntype": "int"}}
      ]}}
    ]}},
  ]}},
}

```

---

Σχήμα 5.5: Παράδειγμα ερωτήματος για μια κλάση “stack” με δύο μεθόδους, μία μέθοδο “push” με τύπο επιστροφής “void” και μία μέθοδο “pop” με τύπο επιστροφής “int”

Εκτελείται ένα λογικό ερώτημα τύπου *bool* που επιτρέπει την αναζήτηση χρησιμοποιώντας συνθήκες οι οποίες πρέπει να ισχύουν ταυτόχρονα. Στην περίπτωση αυτή, το ερώτημα περιλαμβάνει τρεις συνθήκες: μία για το όνομα της κλάσης που πρέπει να είναι “stack”, μια

συνθήκη τύπου `nested` για μια μέθοδο με όνομα `“push”` και τύπο επιστροφής `“void”`, και τέλος άλλη μια συνθήκη τύπου `nested` για μια μέθοδο με όνομα `“pop”` και τύπο επιστροφής `“int”`. Η απάντηση για το ερώτημα αυτό παρουσιάζεται στο Σχήμα 5.6.

---

```
{
  "_shards": {"failed": 0, "successful": 1, "total": 1},
  "hits": {
    "hits": [
      {"_id": "lintool/Cloud9/.../StackOfInts.java", ...,
        "_index": "AGORA", "_score": 15.214152, "_source": ...},
      {"_id": "CyanogenMod/android_frame.../.../Stack.java", ...,
        "_index": "AGORA", "_score": 14.03837, "_source": ...},
      {"_id": "android/platform_frameworks_base/.../Stack.java", ...,
        "_index": "AGORA", "_score": 14.03837, "_source": ...},
      ...
    ],
    "max_score": 15.214152, "total": 412275
  },
  "timed_out": false, "took": 109
}
```

---

Σχήμα 5.6: Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.5

Για την εκτέλεση του ερωτήματος χρειάστηκαν 190 χιλιοστά του δευτερολέπτου, ενώ επιστράφηκαν 412275 έγγραφα (`“hits”`). Για κάθε έγγραφο, η βαθμολογία (`“_score”`) υπολογίζεται σύμφωνα με τη μεθοδολογία της ενότητας 5.3.4. Σε αυτήν την περίπτωση, ο προγραμματιστής μπορεί να εξετάσει τα αποτελέσματα και να επιλέξει εκείνο που ταιριάζει καλύτερα στον κώδικά του. Σημειώστε ότι τα έγγραφα στο Elasticsearch έχουν επίσης ένα ειδικό πεδίο με το όνομα `“_source”`, που εμπεριέχει όλα τα πεδία του εγγράφου. Έτσι, μπορεί κανείς να εξετάσει τα αποτελέσματα και να βρει τον πηγαίο κώδικα κάθε αρχείου στο πεδίο `“source.content”`.

#### 5.4.2.2 Αναζήτηση για συγγραφή σχετικού κώδικα

Άλλο ένα ενδιαφέρον σενάριο που συχνά αντιμετωπίζεται χρησιμοποιώντας RSSes [101] είναι το σενάριο της εύρεσης κώδικα που είναι σχετικός με τη χρήση κάποιας βιβλιοθήκης ή κάποιου `framework`. Όταν ο προγραμματιστής επιθυμεί να χρησιμοποιήσει μια συγκεκριμένη βιβλιοθήκη ή να επεκτείνει τη λειτουργικότητα κάπου `framework`, τότε μπορεί να βρει χρήσιμα παραδείγματα χρησιμοποιώντας ένα ερώτημα όπως αυτό του Σχήματος 5.7.

Σε αυτή την περίπτωση, ο προγραμματιστής θέλει να μάθει πώς να δημιουργήσει μια σελίδα (`page`) για έναν Eclipse wizard, προκειμένου να εξάγει (`export`) κάποιο στοιχείο από το Eclipse plugin του. Σύμφωνα το Eclipse Plugin API, οι σελίδες των Eclipse wizards επεκτείνουν (`extend`) την κλάση `“WizardPage”`. Επομένως, το ερώτημα περιέχει κλάσεις που περιέχουν τη λέξη `“Export”` και που επεκτείνουν την κλάση `“WizardPage”`, ενώ ταυτόχρονα



---

```

{
  "query": {"bool": {"should": [
    {"match": {"files.code.class.name": "Export"}},
    {"match": {"files.code.class.extends": "WizardPage"}},
    {"match": {"files.code.imports": "eclipse"}}
  ]}}
}

```

---

Σχήμα 5.7: Παράδειγμα ερωτήματος για μια κλάση της οποίας το όνομα περιέχει τον όρο “Export”, η οποία επεκτείνει (extends) μια κλάση που ονομάζεται “WizardPage”, ενώ το αρχείο java έχει τουλάχιστον μία δήλωση βιβλιοθήκης (import) που περιέχει τον όρο “eclipse”

περιέχουν δηλώσεις βιβλιοθηκών (imports) με τον όρο “eclipse”. Τα αποτελέσματα του ερωτήματος φαίνονται στο Σχήμα 5.8.

---

```

{
  "_shards": {"failed": 0, "successful": 1, "total": 1},
  "hits": {
    "hits": [
      {"_id": "erlide/erlide/.../EdocExportWizardPage.java", ...,
        "_index": "AGORA", "_score": 11.272949, "_source": ...},
      {"_id": "rssowl/RSSOwl/.../ExportElementsPage.java", ...,
        "_index": "AGORA", "_score": 11.06465, "_source": ...},
      {"_id": "rssowl/RSSOwl/.../ExportOptionsPage.java", ...,
        "_index": "AGORA", "_score": 11.032191, "_source": ...},
      ...
    ],
    "max_score": 11.272949, "total": 16681
  },
  "timed_out": false, "took": 16
}

```

---

Σχήμα 5.8: Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.7

Το ερώτημα επέστρεψε 16681 έγγραφα σε 16 χιλιοστά του δευτερολέπτου. Αναφέρουμε ότι όλα αυτά τα αποτελέσματα δεν καλύπτουν απαραίτητα όλα τα κριτήρια του ερωτήματος, π.χ. ένα αποτέλεσμα μπορεί να επεκτείνει την κλάση “WizardPage”, αλλά το όνομά του να μην περιέχει τον όρο “Export”. Ωστόσο, τα αποτελέσματα που πληρούν τα περισσότερα κριτήρια βρίσκονται στις υψηλότερες θέσεις στην κατάταξη.

### 5.4.2.3 Αναζήτηση προτύπων σε έργα

Το τρίτο σενάριο αναζήτησης είναι ένα ιδιαίτερα καινοτόμο σενάριο στον τομέα των CSEs. Συχνά οι προγραμματιστές θα ήθελαν να βρουν παρόμοια έργα με τα δικά τους προκειμένου να καταλάβουν πώς άλλες ομάδες προγραμματιστών (πιθανώς πιο έμπειρες ομάδες ή ομάδες που εμπλέκονται σε έργα μεγάλης κλίμακας) συνηθίζουν να δομούν τον κώδικα τους. Ως μια γενίκευση αυτού, θα μπορούσε κανείς να πει ότι η δομή που αναφέρουμε μπορεί να είναι ένα πρότυπο σχεδίασης. Στο πλαίσιο μιας αναζήτησης, ωστόσο, θα μπορούσε επίσης να είναι απλά ένας τρόπος να συνδεθούν κάποια τμήματα κώδικα. Ένα ερώτημα για αυτό το σενάριο φαίνεται στο Σχήμα 5.9.

Το ερώτημα σχηματίζεται χρησιμοποιώντας τη σύνταξη “has\_child”, έτσι ώστε τα έργα να έχουν (τουλάχιστον) μία κλάση που υλοποιεί μια διεπαφή (interface) με όνομα παρόμοιο με “Model”, μια άλλη κλάση που υλοποιεί μια διεπαφή με όνομα παρόμοιο με “View”, και, τέλος, μια τρίτη κλάση που υλοποιεί μια διεπαφή με όνομα παρόμοιο με “Controller”. Εν ολίγοις, αναζητούνται έργα που ακολουθούν την αρχιτεκτονική Model-View-Controller (MVC). Επιπλέον, στο ερώτημα ζητείται τα έργα να έχουν κλάσεις που επεκτείνουν (extend) μια κλάση “JFrame”, ουσιαστικά δηλαδή να χρησιμοποιούν την γνωστή βιβλιοθήκη GUI Swing. Αυτό το ερώτημα είναι πράγματι αρκετά λογικό, καθώς τα GUI frameworks είναι γνωστό ότι ακολουθούν τέτοιες αρχιτεκτονικές. Έτσι, ένα παράδειγμα MVC αρχιτεκτονικής για τη βιβλιοθήκη Swing θα μπορούσε να είναι χρήσιμο για έναν προγραμματιστή.

---

```

{
  "query": {"bool": {"should": [
    {"has_child": {"type": "files",
      "query": {"match": {"code.class.implements": "Model"}}
    }},
    {"has_child": {"type": "files",
      "query": {"match": {"code.class.implements": "View"}}
    }},
    {"has_child": {"type": "files",
      "query": {"match": {"code.class.implements": "Controller"}}
    }},
    {"has_child": {"type": "files",
      "query": {"match": {"code.class.extends": "JFrame"}}
    }
  ]}
}}
}

```

---

Σχήμα 5.9: Παράδειγμα ερωτήματος για αρχεία που επεκτείνουν κλάσεις με ονόματα παρόμοια με “Model”, “View” και “Controller”, καθώς επίσης και για αρχεία που επεκτείνουν μια κλάση “JFrame”

Η απάντηση για αυτό το ερώτημα φαίνεται στο Σχήμα 5.10. Το ερώτημα επέστρεψε 679 έγγραφα έργων σε 16 χιλιοστά του δευτερολέπτου. Τα αρχεία αυτών των έργων μπορούν να ανακτηθούν χρησιμοποιώντας το πεδίο “\_parent” του mapping των αρχείων.

---

```

{
  "_shards": {"failed": 0, "successful": 1, "total": 1},
  "hits": {
    "hits": [
      {"_id": "Prototik/HoloEverywhere", ...,
        "_index": "AGORA", "_score": 2.0, "_source": ...},
      {"_id": "xamarin/XobotOS", ...,
        "_index": "AGORA", "_score": 2.0, "_source": ...},
      {"_id": "android/platform_frameworks_base", ...,
        "_index": "AGORA", "_score": 2.0, "_source": ...},
      ...
    ],
    "max_score": 2.0, "total": 679
  },
  "timed_out": false, "took": 16
}

```

---

Σχήμα 5.10: Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.9

#### 5.4.2.4 Αναζήτηση για snippets

Το τέταρτο σενάριο αναζήτησης αφορά την εύρεση μικρών τμημάτων κώδικα (snippets). Παρόλο που η αναζήτηση με βάση τη σύνταξη επιστρέφει ικανοποιητικά αποτελέσματα σε επίπεδο κλάσεων ή μεθόδων, όσον αφορά τον ίδιο τον κώδικα, η αναζήτηση για ένα snippet μπορεί να αποδειχθεί αρκετά χρήσιμη. Ένα παράδειγμα ερωτήματος για αυτό το σενάριο φαίνεται στο Σχήμα 5.11.

---

```

{
  "query": {
    "match": {
      "analyzedcontent": "File myfile = new File(\"myfile.xml\");\n
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();\n
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();\n
        Document doc = dBuilder.parse(myfile);"
    }
  }
}

```

---

Σχήμα 5.11: Παράδειγμα ερωτήματος για ένα snippet που αφορά το διάβασμα αρχείων XML

Το ερώτημα εκτελείται στο πεδίο “analyzedcontent”. Το snippet δίνεται ως όρισμα του ερωτήματος με τις αλλαγές γραμμών να συμβολίζονται με το χαρακτήρα “\n”. Σε αυτή την περίπτωση, το snippet αφορά το διάβασμα αρχείων XML. Οπότε ο προγραμματιστής θα

ήθελε πιθανώς να γνωρίζει αν ο κώδικάς τους είναι κατάλληλος ή το πώς γενικά υλοποιείται το διάβασμα αρχείων XML. Η απάντηση φαίνεται στο Σχήμα 5.12.

---

```

{
  "_shards": {"failed": 0, "successful": 1, "total": 1},
  "hits": {
    "hits": [
      {"_id": "facebook/buck/.../WorkspaceGeneratorTest.java", ...,
        "_index": "AGORA", "_score": 1.8735983, "_source": ...},
      {"_id": "rhuss/jolokia/.../SpringConfigTest.java", ...,
        "_index": "AGORA", "_score": 1.363394, "_source": ...},
      {"_id": "openhav/openhav/.../DeviceCategoryLoader.java", ...,
        "_index": "AGORA", "_score": 1.2306316, "_source": ...},
      ...
    ],
    "max_score": 1.8735983, "total": 522327
  },
  "timed_out": false, "took": 78
}

```

---

Σχήμα 5.12: Παράδειγμα απάντησης για το ερώτημα του Σχήματος 5.11

Το ερώτημα εκτελέστηκε σε 78 χιλιοστά του δευτερολέπτου και επέστρεψε αρκετά έγγραφα (522327). Προφανώς δεν αφορούν όλα τα έγγραφα που επιστράφηκαν το διάβασμα αρχείων XML· αρκετά αποτελέσματα μπορεί π.χ. να περιέχουν μόνο την πρώτη γραμμή του ερωτήματος. Σε κάθε περίπτωση, τα αποτελέσματα που είναι πιο παρόμοια με το ερώτημα είναι αυτά με τη μεγαλύτερη βαθμολογία στην κορυφή της λίστας. Τέλος, σημειώνουμε ότι και αυτό το ερώτημα δεν χρειάστηκε σημαντικό χρόνο εκτέλεσης, παρόλο που το πεδίο “analyzedcontent” περιέχει σχετικά μεγαλύτερα ποσά πληροφορίας από τα άλλα πεδία. Αυτό υποδεικνύει ότι το ευρετήριο μας είναι αρκετά αποδοτικό.

### 5.4.3 Παράδειγμα Σεναρίου Χρήσης

Παρέχουμε ένα παράδειγμα χρήσης της AGORA για την κατασκευή τριών τμημάτων κώδικα που συνδέονται μεταξύ τους: (α) ένα τμήμα για το διάβασμα κειμένων από αρχεία, (β) ένα τμήμα για τη σύγκριση των κειμένων που εξήχθησαν χρησιμοποιώντας έναν αλγόριθμο απόστασης επεξεργασίας (edit distance), και (γ) ένα τμήμα για την αποθήκευση των αποτελεσμάτων σε μια δομή δεδομένων.

Στο Σχήμα 5.13 απεικονίζεται το ερώτημα για ένα τμήμα που διαβάζει το περιεχόμενο ενός αρχείου. Το ερώτημα αφορά μια μέθοδο “readFile” που λαμβάνει ως παράμετρο ένα όνομα αρχείου (“filename”) και επιστρέφει μια συμβολοσειρά (“String”). Ο προγραμματιστής μπορεί προαιρετικά να επιλέξει ένα μηχανισμό για την ανάγνωση των αρχείων, θέτοντας το πεδίο import. Στο Σχήμα 5.13, η επιλογή του προγραμματιστή είναι ο “BufferedReader”, ωστόσο θα μπορούσε το πεδίο να έχει την τιμή “Scanner”, ή ακόμα και την τιμή

“apache” για την χρήση των συναρτήσεων διαχείρισης αρχείων του Apache (ή ακόμα και την τιμή “eclipse” αν θα θέλαμε να διαβάσουμε τα αρχεία ως πόρους του γνωστού IDE).

The image shows a class editor interface with the following fields:

- Class:** Name, Interface, Extends, Implements, Imports (BufferedReader), Package, Access (default), final, abstract.
- Variables:** Variable 1 with Name, Type, Access (any), final, abstract, static.
- Methods:** Method 1 with Name (readFile), Return type (String), Access (any), final, abstract, static, Parameters (Name: filename, Type: String), Exceptions.

Below the editor, a code snippet is shown with two examples of the `readFile` method:

```

glacieruploader-printers/src/test/java/.../VaultInventoryPrinterTest.java
Project: MoriTanosuke/glacieruploader
import java.io.BufferedReader;
private String readFile(String filename) throws IOException;
Download file See at GitHub

Sample/src/com/example/bindableadapter/test/MyExpandableActivity.java
Project: amigold/FunDapter
import java.io.BufferedReader;
public String readFile(String filename);
Download file See at GitHub
    
```

Σχήμα 5.13: Ερώτημα για ένα τμήμα διαβάσματος αρχείων

Έχοντας διαβάσει μια συλλογή από συμβολοσειρές, χρησιμοποιούμε το ερώτημα του Σχήματος 5.14 για να βρούμε ένα τμήμα που λαμβάνει ως είσοδο δύο συμβολοσειρές και υπολογίζει την απόσταση επεξεργασίας (edit distance) τους. Σε αυτήν την περίπτωση, ο προγραμματιστής αναζητεί μια κλάση “EditDistance” που θα επιστρέφει την απόσταση επεξεργασίας μεταξύ δύο συμβολοσειρών (των οποίων τα ονόματα δε δίνονται καθώς δεν είναι απαραίτητα) ως έναν ακέραιο αριθμό. Παρόμοια αποτελέσματα ανακτώντας για μια μέθοδο με το όνομα “EditDistance” (ή ακόμα και αναζητώντας στην αρχική σελίδα της AGORA, που παρουσιάστηκε στο Σχήμα 5.4).

The screenshot shows the configuration for a class named "EditDistance". The "Name" field is "EditDistance". The "Access" dropdown is set to "default". The "Methods" section shows a method named "distance" with a return type of "int" and two parameters of type "String".

The inset window shows two code snippets:

```

onebusaway-webapp/src/main/java/.../StringEditDistance.java
Project: OneBusAway/onebusaway-application-modules
public class StringEditDistance{
    public static int getEditDistance(String s, String t);
}
Download file See at GitHub

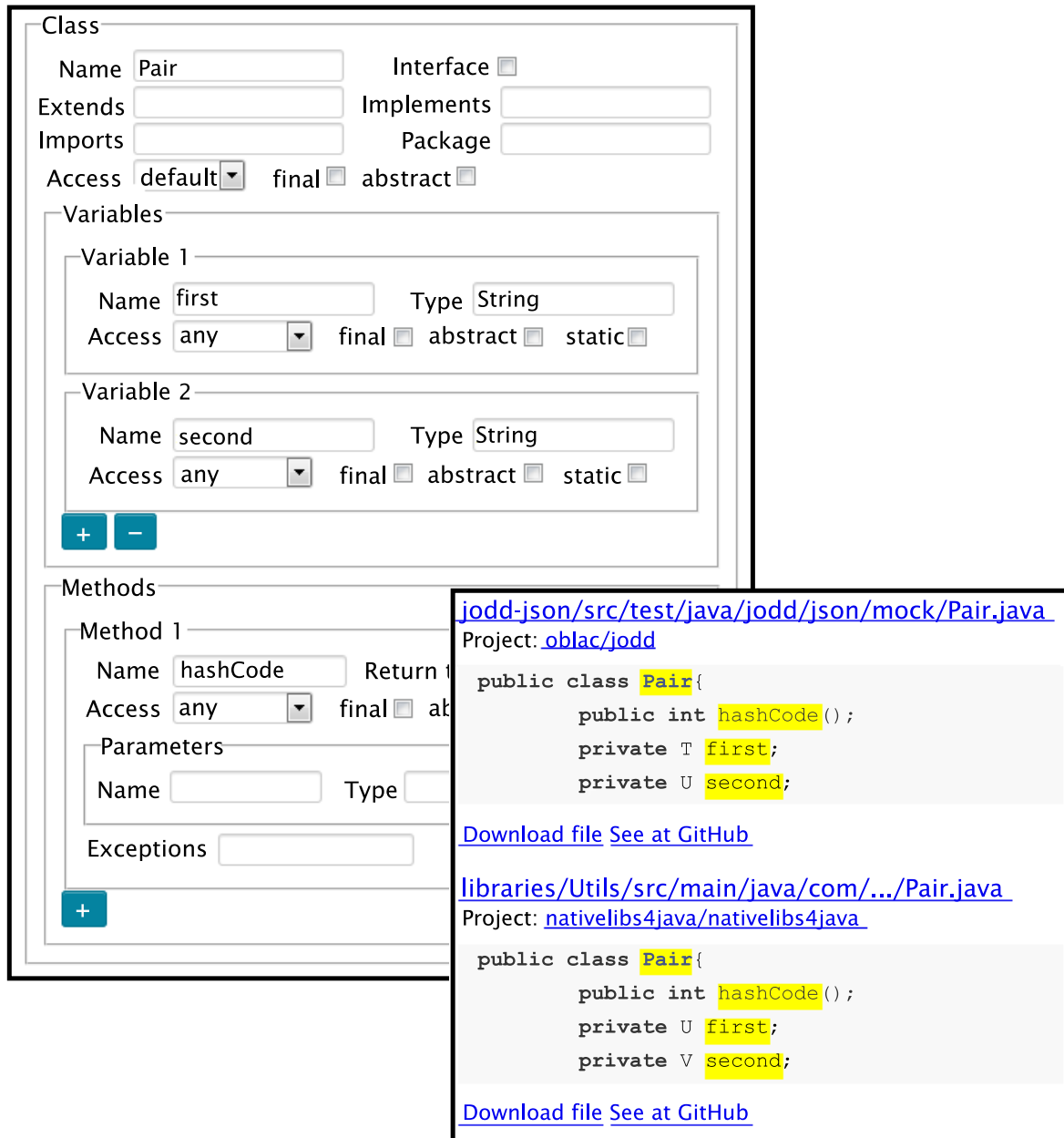
App/src/com/dozuki/ifixit/util/EditDistance.java
Project: iFixit/iFixitAndroid
public class EditDistance{
    public static int editDistance(String s, String t);
}
Download file See at GitHub

```

Σχήμα 5.14: Ερώτημα για έναν αλγόριθμο απόστασης επεξεργασίας

Για την αποθήκευση των αποστάσεων επεξεργασίας που υπολογίστηκαν χρειάζεται ένα ζεύγος (pair) που θα περιέχει τις δύο συμβολοσειρές (ή πιθανώς τα ids των δύο συμβολοσειρών) και ένα αντικείμενο τύπου Java HashMap που θα έχει ως κλειδί (key) το αυτό το ζεύγος και ως τιμή (value) την απόσταση επεξεργασίας των δύο συμβολοσειρών. Το ερώτημα για μια κλάση "Pair" με δύο μεταβλητές "first" και "second" παρουσιάζεται στο Σχήμα 5.15. Επιπλέον, εφόσον τα ζεύγη θα πρέπει να αποτελούν κλειδιά ενός HashMap, η κλάση που αναζητείται θα πρέπει επιπλέον να υλοποιεί τη μέθοδο "hashCode".

Σε αυτή την ενότητα περιγράφηκε ο τρόπος με τον οποίο ο προγραμματιστής μπορεί να χρησιμοποιήσει την AGORA για να βρει τμήματα κώδικα που μπορούν να επαναχρησιμοποιηθούν και έτσι να μειώσει την προσπάθεια που απαιτείται για τη ανάπτυξη του έργου του. Παρόμοια σενάρια χρήσης μπορούν να κατασκευαστούν και για τους άλλους τύπους ερωτημάτων της AGORA (π.χ. αυτούς που περιγράφηκαν στην προηγούμενη ενότητα).



Σχήμα 5.15: Ερώτημα για μια κλάση που αναπαριστά ένα ζεύγος

## 5.5 Αξιολόγηση

Αν και η σύγκριση της AGORA με άλλες CSEs έχει ενδιαφέρον (βλέπε ενότητα 5.2.3), χρειαζόμαστε μια ποσοτική μετρική για να αξιολογήσουμε την αποτελεσματικότητα της CSE που κατασκευάσαμε. Η μετρική που θα επιλέξουμε θα πρέπει να αντικατοπτρίζει τη σχετικότητα των αποτελεσμάτων που επιστρέφονται από την AGORA. Επομένως, πρέπει να εξετάσουμε αν τα αποτελέσματα για ένα ερώτημα καλύπτουν με ακρίβεια την επιθυμητή λειτουργικότητα. Συνεπώς, σε αυτό το υποκεφάλαιο, αξιολογούμε την AGORA έναντι δύο γνωστών CSEs: του GitHub Search και του BlackDuck Open Hub. Επιλέξαμε αυτές τις δύο CSEs, καθώς είναι και οι δύο ενεργές και έχουν μεγάλο αριθμό έργων (βλέπε Πίνακα 5.1).

Αξιολογούμε τις τρεις CSEs σε ένα σενάριο επαναχρησιμοποίησης τμημάτων (component reuse scenario). Στο σενάριο αυτό, ο προγραμματιστής επιθυμεί να βρει ένα συγκεκριμένο τμήμα οπότε υποβάλλει ένα ερώτημα σε κάποια CSE και λαμβάνει μια λίστα με αποτελέσματα. Στη συνέχεια, εξετάζει αυτά τα αποτελέσματα και καθορίζει εάν αυτά είναι σχετικά και αν ταιριάζουν με τον κώδικα του. Στην περίπτωση μας, όμως, αν εξετάζαμε χειροκίνητα τα αποτελέσματα θα είχαμε ζητήματα με την εγκυρότητα της μεθοδολογίας αξιολόγησης (threats to validity). Αντί αυτού, προσδιορίζουμε αν ένα αποτέλεσμα είναι σχετικό και αν είναι χρήσιμο χρησιμοποιώντας περιπτώσεις ελέγχου (test cases, που είναι ένας τρόπος αξιολόγησης που χρησιμοποιείται επίσης στο επόμενο κεφάλαιο). Ο μηχανισμός αξιολόγησης και το σύνολο δεδομένων που χρησιμοποιήσαμε παρουσιάζονται στην ενότητα 5.5.1, ενώ τα αποτελέσματα για τις τρεις CSEs παρουσιάζονται στην ενότητα 5.5.2.

### 5.5.1 Μηχανισμός Αξιολόγησης και Σύνολο Δεδομένων

Το σύνολο δεδομένων μας, που φαίνεται στον Πίνακα 5.6, αποτελείται από 13 τμήματα διαφορετικής πολυπλοκότητας. Σημειώστε επίσης ότι περιέχει δύο εκδόσεις του τμήματος “Stack” έτσι ώστε να αξιολογηθεί η αποτελεσματικότητα των CSEs για διαφορετικά ονόματα μεθόδων.

Πίνακας 5.6: Σύνολο Δεδομένων Αξιολόγησης για Μηχανές Αναζήτησης Κώδικα

Κλάση	Μέθοδοι
Account	deposit, withdraw, getBalance
Article	setId, getId, setName, getName, setPrice, getPrice
Calculator	add, subtract, divide, multiply
ComplexNumber	ComplexNumber, add, getRealPart, getImaginaryPart
CreditCardValidator	isValid
Customer	setAddress, getAddress
Gcd	gcd
Md5	md5Sum
Mortgage	setRate, setPrincipal, setYears, getMonthlyPayment
Movie	Movie, getTitle
Spreadsheet	put, get
Stack	push, pop
Stack2	pushObject, popObject

Για κάθε ένα από τα τμήματα του Πίνακα 5.6, δημιουργήσαμε ερωτήματα για τις CSEs και περιπτώσεις ελέγχου για τον προσδιορισμό της λειτουργικότητας (functionality) των αποτελεσμάτων. Όσον αφορά την δημιουργία ερωτημάτων, το ερώτημα για την AGORA είναι παρόμοιο με αυτά που δείξαμε στο προηγούμενο υποκεφάλαιο. Για το GitHub, εφόσον το API του δεν υποστηρίζει αναζήτηση με βάση τη σύνταξη, σχεδιάσαμε ένα ερώτημα που περιέχει όλες τις μεθόδους του τμήματος τη μία μετά την άλλη. Όταν αυτό το ερώτημα δεν επιστρέφει κάποιο αποτέλεσμα, δημιουργούμε επιπλέον ερωτήματα χωρίς τους τύπους επιστροφής των μεθόδων, χωρίς τις παραμέτρους των μεθόδων και, τέλος, περιλαμβάνοντας τις μεθόδους και τα ονόματα των κλάσεων ως διαχωρισμένα tokens. Ένα παράδειγμα αλληλουχίας ερωτημάτων φαίνεται στο Σχήμα 5.16.



---

```

Query 1: void setAddress(String) String getAddress()
Query 2: setAddress(String) String getAddress()
Query 3: setAddress getAddress
Query 4: set address get address
Query 5: customer

```

---

Σχήμα 5.16: Παράδειγμα αλληλουχίας ερωτημάτων για το GitHub

Το πρώτο ερώτημα ήταν επαρκές στις περιπτώσεις όπου τα ονόματα των μεθόδων ήταν αναμενόμενα, ενώ τα επόμενα ερωτήματα ήταν χρήσιμα όταν υπήρχαν διαφορετικά ονόματα μεθόδων (όπως αυτά του τροποποιημένου “Stack” του Πίνακα 5.6). Όσον αφορά το BlackDuck, ήταν εύκολο να κατασκευαστούν τα ερωτήματα στην ιστοσελίδα της υπηρεσίας, ωστόσο δεν υποστηρίζει κανένα API. Ως εκ τούτου, εκτελέσαμε τα ερωτήματα χειροκίνητα και κατεβάσαμε τα πρώτα αποτελέσματα για κάθε ερώτημα.

Μετά την κατασκευή των ερωτημάτων για τις CSEs δημιουργήσαμε επίσης περιπτώσεις ελέγχου για κάθε τμήμα. Όταν ένα συγκεκριμένο αποτέλεσμα ανακτάται από μια CSE, του αποδίδεται ένα πακέτο Java και ο κώδικας αποθηκεύεται σε ένα αρχείο κλάσης Java. Στη συνέχεια, η περίπτωση ελέγχου τροποποιείται ώστε να αντικατοπτρίζει τα ονόματα των κλάσεων, των μεταβλητών και των μεθόδων του αρχείου. Στο Σχήμα 5.17, παρουσιάζεται μια τροποποιημένη περίπτωση ελέγχου για μια κλάση “Customer1” με μεθόδους “setTheAddress()” και “getTheAddress()”, που επιστρέφεται ως αποτέλεσμα του ερωτήματος “Customer”.

---

```

package package1;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import java.lang.reflect.*;

public class CustomerTest{
    @Test
    public void testAddress() throws Exception{
        int m_index = 0;
        Class<?> clazz = Class.forName("package1.Customer1");
        Object c = clazz.newInstance();
        Method m = clazz.getDeclaredMethod("setTheAddress", String.class);
        m.invoke(c,"test");
        m_index = 1;
        m = clazz.getDeclaredMethod("getTheAddress");
        assertEquals("Wrong string!", "test",m.invoke(c));
    }
}

```

---

Σχήμα 5.17: Παράδειγμα τροποποιημένης περίπτωσης ελέγχου για ένα αποτέλεσμα

Η μεταβλητή “m\_index” προσδιορίζει τη σειρά των κλάσεων μέσα στο αρχείο. Επιπλέον, η σειρά των μεθόδων της κλάσης στο αρχείο πρέπει να συμφωνεί με τη σειρά των μεθόδων στην περίπτωση ελέγχου. Η κατάλληλη σειρά σε κάθε περίπτωση βρίσκεται εξετάζοντας όλους τους πιθανούς συνδυασμούς διατάξεων.

Ως εκ τούτου, το πλαίσιο αξιολόγησής μας καθορίζει εάν κάθε αποτέλεσμα που επιστρέφεται από μια CSE είναι μεταγλωττίσιμο (compilable) και αν περνάει τον έλεγχο (passing the test). Σημειώστε ότι η μεταγλωττισιμότητα αξιολογείται σε σχέση με την περίπτωση ελέγχου, εξασφαλίζοντας έτσι ότι το αποτέλεσμα είναι λειτουργικά ισοδύναμο με αυτό που επιθυμεί ο προγραμματιστής. Οπότε, θα ήταν εφικτό ο προγραμματιστής να επιλέξει τα αποτελέσματα αυτά ακόμα κι αν δεν περνάνε τον έλεγχο και ουσιαστικά να κάνει κάποιες τροποποιήσεις.

### 5.5.2 Αποτελέσματα Αξιολόγησης

Σε αυτή την ενότητα, παρουσιάζουμε τα αποτελέσματα της αξιολόγησής μας για τα ερωτήματα τμημάτων που περιγράφηκαν στην ενότητα 5.5.1. Στον Πίνακα 5.7 φαίνεται το πλήθος των αποτελεσμάτων για κάθε ερώτημα, πόσα από αυτά είναι μεταγλωττίσιμα και πόσα περνάνε τους ελέγχους, για κάθε CSE. Διατηρούμε τα πρώτα 30 αποτελέσματα για κάθε CSE, καθώς κρατώντας περισσότερα αποτελέσματα δε βελτιώθηκε η μεταγλωττισιμότητα και το πλήθος των επιτυχών ελέγχων για κάποια από τις CSEs, ενώ 30 αποτελέσματα είναι περισσότερα από όσα συνήθως θα εξέταζε κάποιος προγραμματιστής σε μια περίπτωση σαν αυτή.

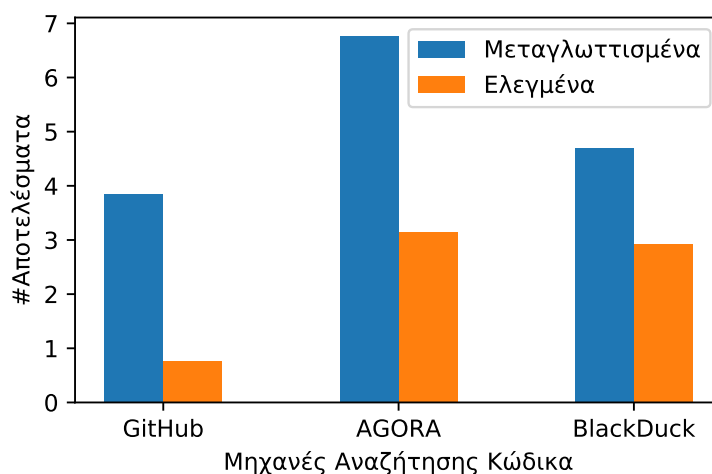
Πίνακας 5.7: Αποτελέσματα που μεταγλωττίστηκαν και αποτελέσματα που πέρασαν τους ελέγχους για τις τρεις CSEs, για κάθε ερώτημα και κατά μέσο όρο, όπου η μορφή για κάθε τιμή είναι Πλήθος Ελεγμένων Αποτελεσμάτων / Πλήθος Μεταγλωττισμένων Αποτελεσμάτων / Συνολικό Πλήθος Αποτελεσμάτων

Ερώτημα	Μηχανές Αναζήτησης Κώδικα		
	GitHub	AGORA	BlackDuck
Account	0 / 10 / 30	1 / 4 / 30	2 / 9 / 30
Article	1 / 6 / 30	0 / 5 / 30	2 / 3 / 15
Calculator	0 / 5 / 30	0 / 5 / 30	1 / 3 / 22
ComplexNumber	2 / 2 / 30	2 / 12 / 30	1 / 1 / 3
CreditCardValidator	0 / 1 / 30	1 / 4 / 30	2 / 3 / 30
Customer	2 / 3 / 30	13 / 15 / 30	7 / 7 / 30
Gcd	1 / 3 / 30	5 / 9 / 30	12 / 16 / 30
Md5	3 / 7 / 30	2 / 10 / 30	0 / 0 / 0
Mortgage	0 / 6 / 30	0 / 4 / 30	0 / 0 / 0
Movie	1 / 3 / 30	2 / 2 / 30	3 / 7 / 30
Spreadsheet	0 / 1 / 30	1 / 2 / 30	0 / 0 / 6
Stack	0 / 3 / 30	7 / 8 / 30	8 / 12 / 30
Stack <sup>(2)</sup>	0 / 0 / 30	7 / 8 / 30	0 / 0 / 0
Μέσος Αριθμός Αποτελεσμάτων	30.0	30.0	17.4
Μέσος Αριθμός Μεταγλωττισμένων Αποτελεσμάτων	3.846	6.769	4.692
Μέσος Αριθμός Ελεγμένων Αποτελεσμάτων	0.769	3.154	2.923

Η AGORA και το GitHub επέστρεψαν τουλάχιστον 30 αποτελέσματα για κάθε ερώτημα. Το BlackDuck, από την άλλη πλευρά, δεν επέστρεψε κανένα αποτέλεσμα για 3 ερωτήματα, ενώ υπάρχουν ακόμα 2 ερωτήματα για τα οποία τα αποτελέσματα που επιστράφηκαν είναι λιγότερα από 10. Αν εξετάσουμε τα ερωτήματα που επέστρεψαν λίγα αποτελέσματα, συμπεραίνουμε ότι αυτό οφείλεται στα διαφορετικά ονόματα μεθόδων. Για παράδειγμα, παρόλο που το BlackDuck επέστρεψε πολλά αποτελέσματα για μια στοίβα “Stack” με μεθόδους “push” και “pop”, δεν επέστρεψε κανένα αποτέλεσμα για τη “Stack” με “pushObject” και “popObject”. Αυτό δείχνει κι ένα από τα βασικά πλεονεκτήματα της AGORA, που οφείλεται ουσιαστικά στον αναλυτή CamelCase που παρουσιάστηκε στην υποενότητα 5.3.2.3. Το GitHub επίσης θα επέστρεφε λιγοστά αποτελέσματα, αν δεν ορίζαμε την αλληλουχία ερωτημάτων όπως στην ενότητα 5.5.1.

Όσον αφορά τη μεταγλωττισιμότητα των αποτελεσμάτων, και πρακτικά τη συμβατότητά τους με τις περιπτώσεις ελέγχου, τα αποτελέσματα της AGORA είναι σαφώς καλύτερα από αυτά των δύο άλλων CSEs. Συγκεκριμένα, η AGORA είναι η μόνη μηχανή που επιστρέφει μεταγλωττίσιμα αποτελέσματα και για τα 13 ερωτήματα, ενώ κατά μέσο όρο επιστρέφει περισσότερα από 6.5 μεταγλωττίσιμα αποτελέσματα ανά ερώτημα. Αντίστοιχα, το GitHub και το BlackDuck επιστρέφουν περίπου 4 και 4.5 αποτελέσματα ανά ερώτημα, υποδεικνύοντας ότι η CSE μας παρέχει περισσότερα χρήσιμα αποτελέσματα στις περισσότερες περιπτώσεις.

Ο αριθμός των αποτελεσμάτων που περνάνε τις περιπτώσεις ελέγχου για κάθε CSE αποτελούν επίσης μια ένδειξη της αποτελεσματικότητας της AGORA. Με βάση το μέσο πλήθος των αποτελεσμάτων που ελέγχθηκαν επιτυχώς (Πίνακας 5.7), είναι σαφές ότι η AGORA είναι πιο αποτελεσματική από το BlackDuck, ενώ το GitHub είναι το λιγότερο αποτελεσματικό από τα τρία CSEs. Παρόλο που η διαφορά μεταξύ της AGORA και του BlackDuck δε φαίνεται τόσο μεγάλη, είναι σημαντικό να σημειωθεί ότι η AGORA περιέχει σαφώς λιγότερα έργα λογισμικού από τα άλλα δύο συστήματα, συνεπώς είναι πιθανό για συγκεκριμένα ερωτήματα να μην υπάρχουν αντίστοιχα ελέγξιμα τμήματα κώδικα μέσα στο ευρετήριο της. Τέλος, το μέσο πλήθος για τα αποτελέσματα που μεταγλωττίστηκαν και για αυτά που πέρασαν τους ελέγχους οπτικοποιείται στο Σχήμα 5.18, όπου είναι σαφές ότι η AGORA είναι η πιο αποτελεσματική από τις τρεις CSEs.



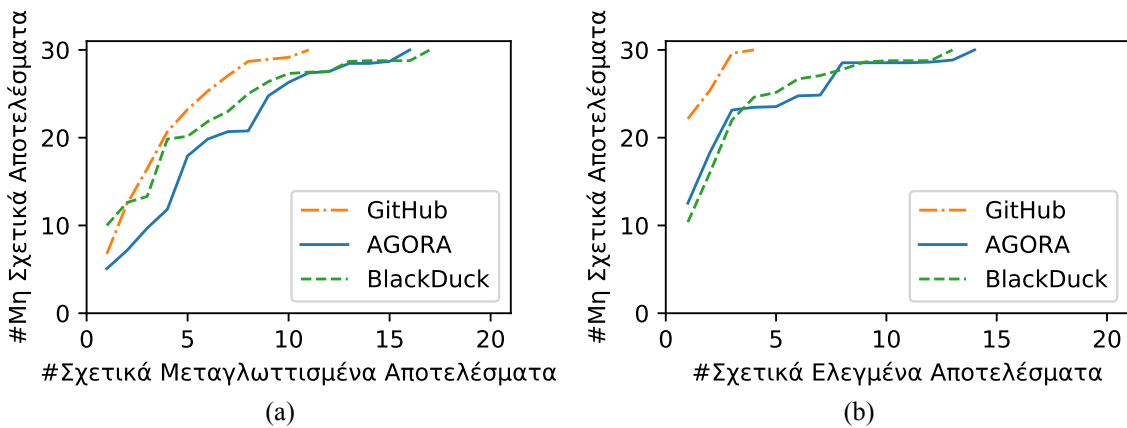
Σχήμα 5.18: Διάγραμμα που απεικονίζει το μέσο πλήθος αποτελεσμάτων που μεταγλωττίστηκαν και αυτών που πέρασαν τους ελέγχους για τις τρεις Μηχανές Αναζήτησης Κώδικα

Αφού εξετάσαμε αν τα αποτελέσματα για κάθε CSE είναι χρήσιμα για το σενάριο επαναχρησιμοποίησης τμημάτων, είναι επίσης σημαντικό να προσδιοριστεί αν τα αποτελέσματα αυτά κατατάσσονται αποτελεσματικά από κάθε CSE. Για το λόγο αυτό, χρησιμοποιούμε το μήκος αναζήτησης (search length) ως μετρική αξιολόγησης της κατάταξης κάθε συστήματος. Το μήκος αναζήτησης ορίζεται ως ο αριθμός των μη σχετικών αποτελεσμάτων που ο χρήστης πρέπει να εξετάσει ώστε να βρει το πρώτο σχετικό αποτέλεσμα. Ορίζεται επίσης αντίστοιχα το μήκος αναζήτησης για το δεύτερο σχετικό αποτέλεσμα, το τρίτο, κ.ο.κ. Στην περίπτωση μας, ορίζουμε δύο παραλλαγές αυτού του μέτρου, μία για τα αποτελέσματα που μεταγλωττίζονται και μία για τα αποτελέσματα που περνάνε τους ελέγχους. Στον Πίνακα 5.8 παρουσιάζονται τα μήκη αναζήτησης για τις δύο αυτές περιπτώσεις όσον αφορά την εύρεση του πρώτου σχετικού (μεταγλωττισμένου ή ελεγμένου) αποτελέσματος, για όλα τα ερωτήματα και για τις τρεις CSEs.

Πίνακας 5.8: Μήκη Αναζήτησης για τα Μεταγλωττισμένα και Ελεγμένα Αποτελέσματα, για κάθε Αποτέλεσμα και ως Μέσοι Όροι, όπου η μορφή για κάθε τιμή είναι Μήκος Αναζήτησης Ελεγμένων Αποτελεσμάτων / Μήκος Αναζήτησης Μεταγλωττισμένων Αποτελεσμάτων

Ερώτημα	Μηχανές Αναζήτησης Κώδικα		
	GitHub	AGORA	BlackDuck
Account	0 / 30	2 / 2	0 / 0
Article	0 / 21	0 / 30	0 / 2
Calculator	0 / 30	8 / 30	0 / 0
ComplexNumber	0 / 0	0 / 0	0 / 0
CreditCardValidator	3 / 30	8 / 15	0 / 2
Customer	0 / 0	0 / 0	1 / 1
Gcd	25 / 25	1 / 8	5 / 5
Md5	1 / 9	0 / 0	30 / 30
Mortgage	0 / 30	4 / 30	30 / 30
Movie	23 / 23	21 / 21	0 / 1
Spreadsheet	4 / 30	20 / 23	30 / 30
Stack	2 / 30	1 / 2	4 / 4
Stack2	30 / 30	1 / 2	30 / 30
Μέσο Μήκος Αναζήτησης Μεταγλωττισμένων	6.769	5.077	10.000
Μέσο Μήκος Αναζήτησης Ελεγμένων	22.154	12.538	10.385

Για την περίπτωση της εύρεσης μεταγλωττίσιμων αποτελεσμάτων, η AGORA υπερτερεί σαφώς των άλλων δύο CSEs, καθώς απαιτεί κατά μέσο όρο την εξέταση περίπου 5 μη σχετικών αποτελεσμάτων ώστε να βρεθεί το πρώτο σχετικό. Αντίθετα, η αναζήτηση στο GitHub ή στο BlackDuck θα απαιτούσε την εξέταση περίπου 6.8 και 10 μη σχετικών αποτελεσμάτων, αντίστοιχα. Το μέσο μήκος αναζήτησης για την εύρεση περισσότερων του ενός σχετικών αποτελεσμάτων φαίνεται στο Σχήμα 5.19a. Από αυτό το Σχήμα, είναι επίσης σαφές ότι χρησιμοποιώντας την AGORA χρειάζεται κανείς να εξετάσει λιγότερα μη σχετικά αποτελέσματα για να βρει κάποια σχετικά. Αν υποθέσουμε, π.χ., ότι ο προγραμματιστής θέλει να βρει τα πρώτα 3 σχετικά αποτελέσματα, τότε κατά μέσο όρο θα χρειαζόταν να εξετάσει 9.7 αποτελέσματα από την AGORA, 13.3 αποτελέσματα από το BlackDuck και 16.5 αποτελέσματα από το GitHub.



Σχήμα 5.19: Διαγράμματα που απεικονίζουν (a) το μήκος αναζήτησης για την εύρεση μεταγλωττίσιμων αποτελεσμάτων, και (b) το μήκος αναζήτησης για την εύρεση αποτελεσμάτων που περνάνε τους ελέγχους, για τις τρεις Μηχανές Αναζήτησης Κώδικα.

Στον Πίνακα 5.8 φαίνεται επίσης το μήκος αναζήτησης για τα αποτελέσματα που περνάνε τους ελέγχους για τις τρεις CSEs. Στην περίπτωση αυτή, το BlackDuck δείχνει να απαιτεί την εξέταση λιγότερων μη σχετικών αποτελεσμάτων κατά μέσο όρο σε σχέση με την AGORA, σχεδόν 10.5 και 12.5 αντίστοιχα, ενώ το GitHub απαιτεί την εξέταση περισσότερων από 22 αποτελεσμάτων που δεν πέρασαν δηλαδή τους ελέγχους. Ωστόσο, είναι σημαντικό να σημειωθεί ότι το πρώτο σχετικό αποτέλεσμα της AGORA περιλαμβάνεται στα πρώτα 30 αποτελέσματα για 10 από τα 13 ερωτήματα, ενώ για το BlackDuck για 9 από τα 13 ερωτήματα. Παρατηρούμε επίσης ότι το μήκος αναζήτησης για τις CSEs επηρεάζεται σημαντικά από ακραίες τιμές. Για παράδειγμα, το BlackDuck είναι αρκετά αποτελεσματικό για τα ερωτήματα “Account”, “Calculator” και “ComplexNumber”, κάτι που είναι αναμενόμενο καθώς συγκρίνουμε CSEs που περιέχουν διαφορετικά έργα λογισμικού και επομένως διαφορετικά τμήματα κώδικα. Συνεπώς, η AGORA είναι αρκετά αποτελεσματική, αν σκεφτεί κανείς ότι περιέχει λιγότερα από 3000 έργα λογισμικού, ενώ το GitHub και το BlackDuck περιέχουν περίπου 14 εκατομμύρια και 650 χιλιάδες έργα αντίστοιχα.

Το μέσο μήκος αναζήτησης για την εύρεση περισσότερων του ενός επιτυχώς ελεγμένων αποτελεσμάτων φαίνεται στο Σχήμα 5.19b. Όπως φαίνεται σε αυτό το Σχήμα, τόσο η AGORA όσο και το BlackDuck είναι σαφώς πιο αποτελεσματικά από το GitHub. Το BlackDuck απαιτεί την εξέταση ελαφρώς λιγότερων αποτελεσμάτων για την εύρεση των 3 πρώτων αποτελεσμάτων που περνάνε τους ελέγχους, ενώ η AGORA υπερτερεί όταν απαιτούνται περισσότερα αποτελέσματα.

## 5.6 Συμπεράσματα

Σε αυτό το κεφάλαιο παρουσιάσαμε την AGORA, μια CSE με προηγμένες δυνατότητες αναζήτησης, ένα ολοκληρωμένο RESTful API και απρόσκοπτη σύνδεση με το GitHub. Η δύναμη της AGORA έγκειται στην αρχιτεκτονική της και στον ισχυρό μηχανισμό ευρετηριοποίησης (indexing) της. Δείξαμε πώς μπορεί κανείς να χρησιμοποιήσει την AGORA για διάφορα ερωτήματα και πώς μπορεί να είναι χρήσιμη για τον προγραμματιστή σε ένα σενάριο επαναχρησιμοποίησης τμημάτων κώδικα.

Όσον αφορά τη μελλοντική έρευνα, θα μπορούσαμε να διεξάγουμε μια μελέτη σε ρεαλιστικό περιβάλλον για να προσδιορίσουμε αν τα χαρακτηριστικά της AGORA είναι αποδεκτά από την κοινότητα προγραμματιστών. Το ευρετήριο μπορεί επίσης να επεκταθεί περιλαμβάνοντας μετρικές ποιότητας (π.χ. μετρικές στατικής ανάλυσης) για κάθε αρχείο. Τέλος, μπορούμε να χρησιμοποιήσουμε στρατηγικές επέκτασης ερωτημάτων (query expansion) και/ή να ενσωματώσουμε σημασιολογία (semantics) για να βελτιωθεί περαιτέρω η σχετικότητα των αποτελεσμάτων.

# 6

## Εξόρυξη Κώδικα για Επαναχρησιμοποίηση Τμημάτων

### 6.1 Επισκόπηση

Όπως αναφέρθηκε ήδη στο προηγούμενο κεφάλαιο, οι πρακτικές ανάπτυξης λογισμικού ανοικτού κώδικα (open-source software development) επωφελούνται από την επαναχρησιμοποίηση τμημάτων πηγαίου κώδικα από διάφορες διαδικτυακές πηγές. Με αυτόν τον τρόπο μειώνεται σημαντικά ο χρόνος και η προσπάθεια που δαπανάται για την ανάπτυξη λογισμικού, ενώ βελτιώνεται επίσης και η ποιότητα των έργων. Σε αυτό το πλαίσιο, οι *Μηχανές Αναζήτησης Κώδικα (Code Search Engines - CSEs)* αποδείχθηκαν πολύτιμες για τον εντοπισμό χρήσιμων τμημάτων κώδικα (βλέπε προηγούμενο κεφάλαιο). Ωστόσο, έχουν και σημαντικούς περιορισμούς, καθώς δεν λαμβάνουν υπόψη το πλαίσιο (context) ενός ερωτήματος, δηλαδή τον πηγαίο κώδικα και/ή τη λειτουργικότητα που επιθυμεί να καλύψει ο προγραμματιστής. Για να αντιμετωπιστεί αυτή η αδυναμία, έχουν αναπτυχθεί διάφορα εξειδικευμένα συστήματα στην ευρύτερη περιοχή των *Συστημάτων Προτάσεων στην Τεχνολογία Λογισμικού (Recommendation Systems in Software Engineering - RSSEs)*. Στο πλαίσιο της επαναχρησιμοποίησης κώδικα, τα RSSEs είναι έξυπνα συστήματα που εξάγουν το ερώτημα από τον κώδικα του προγραμματιστή και επεξεργάζονται τα αποτελέσματα των CSEs για να προτείνουν τμήματα λογισμικού, συνοδευόμενα από πληροφορίες που βοηθούν τον προγραμματιστή να κατανοήσει και να επαναχρησιμοποιήσει τον κώδικα. Τον τελευταίο καιρό, ακολουθώντας τα πρότυπα της *Ανάπτυξης Λογισμικού Οδηγούμενης από Ελέγχους (Test-Driven Development - TDD)*, πολλά RSSEs χρησιμοποιούν περιπτώσεις ελέγχου για να προσδιορίσουν εάν τα τμήματα κώδικα που προτείνονται καλύπτουν την επιθυμητή λειτουργικότητα.

Παρόλο που έχουν αναπτυχθεί αρκετά από αυτά τα συστήματα *επαναχρησιμοποίησης οδηγούμενης από ελέγχους (Test-Driven Reuse - TDR)*, η αποτελεσματικότητά τους σε ρεαλιστικά σενάρια αναζήτησης τμημάτων κώδικα είναι αμφισβητούμενη. Όπως αναφέρει ο Walker [187], τα περισσότερα από αυτά τα συστήματα επικεντρώνονται στο πρόβλημα της εύρεσης σχετικών αντικειμένων λογισμικού από τη σκοπιά της ανάκτησης πληροφοριών

(information retrieval), ενώ αγνοούν τη σύνταξη (syntax) και τη σημασιολογία (semantics) του πηγαίου κώδικα. Επιπλέον, σύμφωνα με τον Nuroolahzade και τους συνεργάτες του [99], τα τρέχοντα RSSEs βασίζονται στην αντιστοίχιση υπογραφών μεταξύ των τμημάτων (signatures μεθόδων με τις παραμέτρους ή τα πεδία τους), χρησιμοποιώντας όμως μεθόδους αντιστοίχισης που δεν είναι αρκετά ευέλικτες (π.χ. παρόμοιες μέθοδοι με διαφορετικά ονόματα μπορεί να μην αντιστοιχιστούν) και/ή ελλείπονται σημασιολογίας, επομένως περιορίζεται το πλήθος και η ποιότητα των αποτελεσμάτων. Ακόμη και όταν επιστρέφονται κάποια αποτελέσματα, συχνά δεν πραγματοποιείται μεταεπεξεργασία (postprocessing), κι έτσι απαιτείται περαιτέρω προσπάθεια από τον προγραμματιστή για να τα προσαρμόσει στον πηγαίο κώδικά του. Άλλες σημαντικές χρονοβόρες προκλήσεις περιλαμβάνουν την κατανόηση της λειτουργικότητας των ανακτημένων τμημάτων και την επισήμανση πιθανών εξαρτήσεων τους. Επιπλέον, πολλά RSSEs εξαρτώνται από στατικές, μη ανανεώσιμες και μερικές φορές απαρχαιωμένες (obsolete) αποθήκες λογισμικού, οπότε η ποιότητα των αποτελεσμάτων τους είναι αμφισβητήσιμη. Τέλος, ως ένα ακόμα μειονέκτημα μπορούμε να αναφέρουμε την πολυπλοκότητα ορισμένων λύσεων, οι οποίες δεν αποδίδουν αποτελέσματα σε εύλογο χρονικό διάστημα.

Στο κεφάλαιο αυτό, παρουσιάζουμε το *Mantissa*<sup>1</sup>, ένα RSSE που επιτρέπει την αναζήτηση τμημάτων λογισμικού σε ηλεκτρονικά αποθετήρια. Το *Mantissa* εξάγει το ερώτημα από τον πηγαίο κώδικα του προγραμματιστή και χρησιμοποιεί την AGORA και το GitHub για την αναζήτηση χρήσιμου κώδικα. Τα αποτελέσματα ταξινομούνται χρησιμοποιώντας το Μοντέλο Διανυσματικού Χώρου (*Vector Space Model - VSM*), ενώ για την καλύτερη κατάταξη των αποτελεσμάτων χρησιμοποιούνται τεχνικές *Ανάκτησης Πληροφοριών (Retrieval Information - IR)* και ευριστικοί κανόνες (heuristics). Επιπλέον, εκτελείται ένα σύνολο μετασχηματισμών πηγαίου κώδικα έτσι ώστε κάθε αποτέλεσμα να είναι έτοιμο προς χρήση από τον προγραμματιστή. Τέλος, για κάθε αποτέλεσμα, εκτός από τη βαθμολογία της σχετικότητάς του, παρέχονται και χρήσιμες πληροφορίες που υποδεικνύουν τον αρχικό πηγαίο κώδικα, τη ροή ελέγχου (control flow) για τον κώδικα, καθώς και τις εξωτερικές του εξαρτήσεις (external dependencies).

## 6.2 Βιβλιογραφία για τα Συστήματα Προτάσεων στην Τεχνολογία Λογισμικού

### 6.2.1 Επισκόπηση

Η ιδέα του McIlroy [9] για επαναχρησιμοποίηση τμημάτων στον τομέα της τεχνολογίας λογισμικού, που χρονολογείται από το 1968, είναι πιο επίκαιρη από ποτέ. Σε μια περίοδο κατά την οποία υπάρχουν τεράστια ποσά δεδομένων στο διαδίκτυο, ενώ ταυτόχρονα ο χρόνος διάθεσης στην αγορά (time-to-market) έχει ζωτικό ρόλο στη βιομηχανία, η επαναχρησιμοποίηση λογισμικού μπορεί να οδηγήσει σε εξοικονόμηση χρόνου, καθώς και στην ανάπτυξη αξιόπιστου λογισμικού υψηλής ποιότητας.

Τα RSSEs έχουν εμφανιστεί τα τελευταία χρόνια με σκοπό τη διευκόλυνση της διαδικασίας επαναχρησιμοποίησης κώδικα [14, 188–190]. Υπάρχουν αρκετά συστήματα προ-

<sup>1</sup>Ο όρος “Mantissa” αναφέρεται στις ιέρειες της αρχαίας Ελλάδας, οι οποίες ερμήνευαν τα σημάδια που έστελναν οι θεοί και έδιναν χρησμούς. Με παρόμοιο τρόπο, το σύστημά μας ερμηνεύει το ερώτημα του χρήστη και παρέχει τα χρήσιμα αποτελέσματα.



τάσεων που προσανατολίζονται στην επαναχρησιμοποίηση κώδικα και τα βήματα της λειτουργίας τους είναι παρόμοια. Συγκεκριμένα, πολλά σύγχρονα RSSEs [95, 96, 100, 101, 104, 191, 192] ακολουθούν μια διαδικασία που αποτελείται από 5 στάδια:

- Στάδιο 1 - Συλλογή Δεδομένων (Data Collection): Τοπικά αποθετήρια (local repositories), CSEs ή ακόμη και online αποθετήρια ανοικτού κώδικα χρησιμοποιούνται για την ανάκτηση πηγαίου κώδικα.
- Στάδιο 2 - Προεπεξεργασία Δεδομένων (Data Preprocessing): Μετά την εξαγωγή πληροφοριών από τον πηγαίο κώδικα, τα δεδομένα μετασχηματίζονται σε κάποια αναπαράσταση ώστε να μπορούν αργότερα να εξορυχθούν.
- Στάδιο 3 - Ανάλυση και Εξόρυξη Δεδομένων (Data Analysis and Mining): Περιλαμβάνει όλες τις τεχνικές που χρησιμοποιούνται για την εξαγωγή χρήσιμων πληροφοριών από τα τμήματα κώδικα. Κάποια παραδείγματα τεχνικών που χρησιμοποιούνται σε αυτό το στάδιο περιλαμβάνουν την ανίχνευση διπλότυπων (duplicate detection), την εξόρυξη συχνών ακολουθιών (frequent sequences mining), κ.α.
- Στάδιο 4 - Μεταεπεξεργασία Δεδομένων (Data Postprocessing): Κατασκευάζονται οι προτάσεις του συστήματος, συνήθως με τη μορφή κατάταξης πιθανών λύσεων.
- Στάδιο 5 - Παρουσίαση Αποτελεσμάτων (Results Presentation): Τα αποτελέσματα παρουσιάζονται μαζί με κάποια βαθμολογία για τη σχετικότητά τους και πιθανώς κάποιες τιμές μετρικών σχετικά με τις λειτουργικές (functional) ή μη λειτουργικές (non-functional) πτυχές κάθε αποτελέσματος.

Στις παρακάτω ενότητες, παραθέτουμε κάποιες από τις πιο γνωστές προσεγγίσεις συστημάτων RSSE.

## 6.2.2 Συστήματα Επαναχρησιμοποίησης Κώδικα

Ένα από τα πρώτα RSSEs επαναχρησιμοποίησης κώδικα είναι το CodeFinder, ένα σύστημα που αναπτύχθηκε από τον Heninger [193]. Το CodeFinder δεν κάνει χρήση κάποιας εξειδικευμένης γλώσσας ερωτημάτων· βασίζεται σε λέξεις-κλειδιά (keywords), ετικέτες κατηγοριών (category labels) και χαρακτηριστικά (attributes) για τη διεξαγωγή αναζητήσεων. Η μεθοδολογία του βασίζεται στα *συσχετιστικά δίκτυα (associative networks)* και έχει κοινά στοιχεία με τον αλγόριθμο *PageRank* [194] που δημοσιεύθηκε λίγα χρόνια αργότερα. Το CodeWeb [195] είναι ένα άλλο σύστημα που αναπτύχθηκε για εφαρμογές του KDE, το οποίο λαμβάνει υπόψη τη δομή ενός ερωτήματος για να προτείνει αποτελέσματα κώδικα, που συνοδεύονται από τεκμηρίωση. Αν και είναι ένα από τα πρώτα RSSEs που χρησιμοποιούν *κανόνες συσχέτισης (association rules)*, είναι στην πραγματικότητα ένα σύστημα περιήγησης και όχι αναζήτησης.

Το 2002, ένα πολλά υποσχόμενο σύστημα που ονομάζεται CodeBroker δημοσιεύτηκε από τους Ye και Fischer [196]. Το CodeBroker αναπτύχθηκε ως plugin για τον επεξεργαστή κειμένου Emacs. Το εργαλείο εξάγει το ερώτημα χρησιμοποιώντας σχόλια τεκμηρίωσης (documentation comments) ή υπογραφές (signatures) και είναι το πρώτο που χρησιμοποιεί

ένα τοπικό αποθετήριο για σχόλια τύπου Javadoc. Το CodeBroker βασίζεται επίσης σε μεγάλο βαθμό στον προγραμματιστή, επιτρέποντας την ανάκτηση με αναδιατύπωση ερωτήματος (query reformulation), ενώ παράλληλα παρουσιάζει αποτελέσματα που συνοδεύονται από τη σχετική τεκμηρίωση. Ωστόσο, η χρήση μόνο μιας τοπικής βάσης δεδομένων και η εξάρτησή της από την ύπαρξη σχολίων τύπου Javadoc αποτελούν σημαντικούς περιορισμούς για το CodeBroker.

Από το 2004 και μετά, όταν δημιουργήθηκε το Eclipse Foundation<sup>2</sup>, πολλά RSSEs αναπτύχθηκαν ως plugins για το Eclipse IDE. Δύο από τα πρώτα συστήματα τέτοιου τύπου είναι το Strathcona [93] και ο Prospector [94], που δημοσιεύθηκαν και τα δύο το 2005. Το Strathcona εξάγει ερωτήματα από τμήματα κώδικα και χρησιμοποιεί τεχνικές αντιστοίχισης με βάση τη δομή (structural matching), με σκοπό την προώθηση των πιο συχνών αποτελεσμάτων στον χρήστη, τα οποία επιπλέον συνοδεύονται από διαγράμματα UML. Ωστόσο, επίσης χρησιμοποιεί ένα τοπικό αποθετήριο, με αποτέλεσμα τα αποτελέσματα να είναι περιορισμένα.

Ο Prospector εισήγαγε διάφορα νέα χαρακτηριστικά, συμπεριλαμβανομένης της δυνατότητας παραγωγής κώδικα (code generation). Το κύριο σενάριο αυτού του εργαλείου είναι η εύρεσης μιας διαδρομής μεταξύ ενός αντικειμένου πηγής και ενός αντικειμένου προορισμού στον πηγαίο κώδικα. Αυτές οι διαδρομές ονομάζονται *jungloids* και η ροή προγράμματος που προκύπτει ονομάζεται γράφος *jungloid* (jungloid graph). Ο Prospector απαιτεί επίσης τη συντήρηση μιας τοπικής βάσης δεδομένων, ενώ το κύριο μειονέκτημα του εργαλείου είναι ότι οι πληροφορίες για τα APIs που χρησιμοποιεί μπορεί να είναι ελλιπείς, επομένως η εφαρμογή του μπορεί να μην είναι εφικτή.

Από το 2006, που εμφανίστηκαν οι πρώτες CSEs, τα περισσότερα RSSEs άρχισαν να βασίζονται σε αυτές για την εύρεση πηγαίου κώδικα, εξασφαλίζοντας έτσι ότι τα αποτελέσματά τους είναι ενημερωμένα. Ένα από τα πρώτα συστήματα που χρησιμοποιούν μια CSE είναι το MAPO, ένα σύστημα που αναπτύχθηκε από τους Xie και Pei [100], με μια δομή που μοιάζει με τα περισσότερα σύγχρονα RSSE. Μετά το κατέβασμα πηγαίου κώδικα από αποθετήρια, το MAPO αναλύει τα snippets και εξάγει ακολουθίες. Σε αυτές οι ακολουθίες στη συνέχεια εφαρμόζονται τεχνικές εξόρυξης ακολουθιών για να δημιουργηθεί μια κατάταξη σύμφωνα με το ερώτημα του προγραμματιστή. Το XSnippet, το οποίο δημοσιεύτηκε το 2006 από τους Sahavechaphan και Claypool [101], είναι άλλο ένα δημοφιλές RSSE που είναι παρόμοιο με το MAPO. Ωστόσο, χρησιμοποιεί δομές όπως ακυκλικούς γράφους (acyclic graphs), δένδρα τύπου B+ (B+ trees), ενώ επίσης εφαρμόζει και μια επέκταση του αλγορίθμου *Breadth-First Search* (BFS) για εξόρυξη γράφων.

Το PARSEWeb είναι άλλο ένα πολύ δημοφιλές σύστημα που αναπτύχθηκε από τους Thummalapenta και Xie [95]. Ομοίως με το MAPO, το PARSEWeb προτείνει παραδείγματα χρήσεων για APIs χρησιμοποιώντας snippets από τη Google CSE<sup>3</sup>. Το PARSEWeb χρησιμοποιεί *Αφηρημένα Συντακτικά Δένδρα* (Abstract Syntax Trees - AST) και *Κατευθυνόμενους Ακυκλικούς Γράφους* (Directed Acyclic Graphs - DAGs) για την αναπαράσταση κώδικα, και εφαρμόζει τεχνικές ομαδοποίησης για τις παρόμοιες ακολουθίες. Επίσης, κάνει χρήση ευριστικών κανόνων (heuristics) για την εξαγωγή ακολουθιών, ενώ παράλληλα επιτρέπει τη διάσπαση ερωτημάτων (query splitting) σε περίπτωση που δεν έχουν βρεθεί αρκετά αποτελέσματα.

<sup>2</sup><https://eclipse.org/org/foundation/>

<sup>3</sup>Όπως ήδη αναφέρθηκε στο προηγούμενο κεφάλαιο, η λειτουργία της CSE της Google διεκόπη το 2013.

Τον τελευταίο καιρό, πολλά από τα συστήματα που αναπτύχθηκαν δέχονται ερωτήματα σε φυσική γλώσσα. Χαρακτηριστικά παραδείγματα αποτελούν το Portfolio [197], το Exemplar [198] ή το Snipmatch [103] που είναι το σύστημα προτάσεων κώδικα του Eclipse IDE. Το Bing Code Search [104] εισάγει επιπλέον ένα σύστημα κατάταξης πολλαπλών παραμέτρων (multi-parameter ranking), το οποίο χρησιμοποιεί πηγαίο κώδικα και σημασιολογία που παρέχονται από τα αποτελέσματα της μηχανής αναζήτησης Bing. Τέλος, ορισμένα RSSEs αξιοποιούν δεδομένα από συστήματα ερωταπαντήσεων για την πρόταση τμημάτων κώδικα, όπως π.χ. το Example Overflow [102] που προτείνει snippets από το Stack Overflow, ενώ τελευταία υπάρχει και προτίμηση σε δυναμικά και διαδραστικά συστήματα, όπως το CodeHint [191].

### 6.2.3 Συστήματα Επαναχρησιμοποίησης Οδηγούμενης από Ελέγχους

Όπως ήδη αναφέρθηκε, σε μια προσπάθεια να διασφαλιστεί ότι καλύπτεται η επιθυμητή λειτουργικότητα, αρκετά RSSEs αξιολογούν τα τμήματα κώδικα χρησιμοποιώντας περιπτώσεις ελέγχου (test cases). Δεδομένου ότι αυτά τα συστήματα είναι αρκετά παρόμοια με το δικό μας, τα εξετάζουμε εκτενώς σε αυτή την ενότητα.

Η TDD έχει γίνει αρκετά δημοφιλής την τελευταία δεκαετία. Η έννοια της TDD βασίζεται στη συγγραφή των περιπτώσεων ελέγχου πριν τη συγγραφή του κώδικα [64]. Έτσι, ο στόχος είναι να παραχθεί ορθός κώδικας που να καλύπτει την απαιτούμενη λειτουργικότητα. Η TDD συνδυάζει τρεις βασικές δραστηριότητες: τη *συγγραφή κώδικα (coding)*, τον *έλεγχο (testing)* και τη βελτίωση της δομής του κώδικα (*refactoring*). Θα πρέπει τόσο ο πηγαίος κώδικας όσο και οι έλεγχοι να λειτουργούν σωστά: ο έλεγχος να αποτυγχάνει πριν γράφει το κατάλληλο τμήμα κώδικα και να περνάει μετά την προσθήκη του.

Στο πλαίσιο της επαναχρησιμοποίησης, αρκετοί ερευνητές επιχειρήσαν να αξιοποιήσουν τα πλεονεκτήματα της TDD. Αν και η έννοια των συστημάτων TDR είναι γνωστή ήδη από το 2004 [199], η έρευνα στα σχετικά συστήματα έχει διεξαχθεί κυρίως μετά το 2007 [99].

Ένα από τα πρώτα συστήματα σε αυτή την κατηγορία είναι ο Code Conjurer, ένα RSSE που αναπτύχθηκε από τον Hummel και τους συνεργάτες του [96, 200, 201] ως ένα plugin του Eclipse. Ο Code Conjurer χρησιμοποιεί τη CSE MeroBase [90], η οποία δημιουργήθηκε από την ίδια ερευνητική ομάδα για να επιτρέψει την δημιουργία ερωτημάτων με βάση τη σύνταξη για την εύρεση επαναχρησιμοποιήσιμων τμημάτων. Βασικά πλεονεκτήματα του Code Conjurer αποτελούν η επιλογή ομαδοποίησης παρόμοιων αποτελεσμάτων καθώς και το ότι τα αποτελέσματα συνοδεύονται από μετρικές πολυπλοκότητας. Το εργαλείο υποστηρίζει επίσης μια λειτουργία που επιτρέπει τα ερωτήματα να μπορούν να δημιουργηθούν δυναμικά κάθε φορά που ο προγραμματιστής προσθέτει, τροποποιεί ή διαγράφει ένα κομμάτι κώδικα. Παρόλο που ο Code Conjurer είναι ένα αποτελεσματικό RSSE, οι τεχνικές αντιστοίχισης που χρησιμοποιεί δεν είναι ευέλικτες σε περιπτώσεις με μικρές διαφορές μεταξύ του ανακτημένου κώδικα και των ελέγχων, ενώ η προηγμένη έκδοση του εργαλείου (έκδοση adaptation) είναι πολύ υπολογιστικά πολύπλοκη. Επιπλέον, η λειτουργία της CSE MeroBase που χρησιμοποιεί έχει διακοπεί, οπότε το εργαλείο δεν είναι λειτουργικό.

Μια άλλη ενδιαφέρουσα προσέγγιση είναι αυτή του CodeGenie που προτείνεται από τον Lemos και τους συνεργάτες του [192, 202, 203]. Ομοίως με τον Code Conjurer, το CodeGenie χρησιμοποιεί το δικό της CSE, το Sourcerer [91, 92], για να ανακτήσει τμήματα πηγαίου κώδικα. Ένα σημαντικό πλεονέκτημα του CodeGenie είναι η δυνατότητα εξαγω-

γής του ερωτήματος από τον κώδικα της περίπτωση ελέγχου του προγραμματιστή, οπότε ο τελευταίος χρειάζεται να γράψει μόνο μια περίπτωση ελέγχου τύπο JUnit. Ωστόσο, όπως σημειώνεται στο [204], η εξαγωγή του ερωτήματος από την περίπτωση ελέγχου θα μπορούσε να οδηγήσει σε αποτυχία καθώς δεν είναι απλή διαδικασία. Επιπλέον, το εργαλείο δεν ανιχνεύει διπλότυπα (duplicates) στα αποτελέσματα, ενώ επίσης δεν είναι αρκετά ευέλικτο σε περιπτώσεις όπου υπάρχουν διαφορές μεταξύ του κώδικα του ερωτήματος και του κώδικα των τμημάτων.

Ο Reiss ακολουθεί μια ελαφρώς διαφορετική προσέγγιση στο S6 [98, 205]. Το S6 είναι μια διαδικτυακή εφαρμογή (web application) που ενσωματώνει πολλές CSEs και αποθήκες λογισμικού, όπως π.χ. το Krugle<sup>4</sup> και το GitHub. Αν και ο προγραμματιστής χρειάζεται να κατασκευάσει πιο σύνθετα ερωτήματα (π.χ. απαιτούνται κάποιες σημασιολογικές λέξεις-κλειδιά), τα αποτελέσματα γίνονται αυτοματοποιημένα refactoring, καθιστώντας το S6 ως πιθανώς το μόνο RSSE που ενσωματώνει χαρακτηριστικά μετασχηματισμού κώδικα (code transformation). Ωστόσο, η υπηρεσία επιστρέφει διπλότυπα (duplicates), ενώ τα περισσότερα αποτελέσματα είναι ελλιπή snippets που μπορεί να δυσκολέψουν τον προγραμματιστή.

Ένα άλλο ενδιαφέρον σύστημα είναι το FAST, ένα RSSE με τη μορφή ενός Eclipse plugin που αναπτύχθηκε από την Krug [204]. Το FAST χρησιμοποιεί τη CSE MeroBase και δίνει έμφαση στην αυτόματη εξαγωγή ερωτημάτων (automated query extraction)· η υπογραφή (signature) εξάγεται από μια περίπτωση ελέγχου τύπου JUnit και στη συνέχεια χρησιμοποιείται για το σχηματισμό ενός ερώτημα για τη CSE MeroBase. Τα αποτελέσματα που λαμβάνονται στη συνέχεια αξιολογούνται χρησιμοποιώντας αυτοματοποιημένες τεχνικές ελέγχου. Ένα μειονέκτημα αυτού του συστήματος είναι ότι ο προγραμματιστής πρέπει να εξετάσει το ερώτημα που εξάγεται από την περίπτωση ελέγχου πριν εκτελέσει την αναζήτηση. Επιπλέον, η υπηρεσία δεν χρησιμοποιεί αναπτυσσόμενες αποθήκες λογισμικού, αλλά τη CSE MeroBase που πλέον δεν είναι ενεργή.

## 6.2.4 Χαρακτηριστικά Συστημάτων Επαναχρησιμοποίησης Κώδικα

Παρόλο που έχουν αναπτυχθεί αρκετά RSSEs τα τελευταία χρόνια, τα περισσότερα από αυτά έχουν σημαντικούς περιορισμούς. Για παράδειγμα, δεν είναι όλα συνδεδεμένα με ενημερωμένα αποθετήρια λογισμικού, ενώ οι τεχνικές αντιστοίχισης τους περιορίζονται στην ακριβή αντιστοίχιση μεταξύ ονομάτων μεθόδων και παραμέτρων. Υποστηρίζουμε ότι ένα RSSE πρέπει να έχει τα ακόλουθα χαρακτηριστικά:

### Χαρακτηριστικό 1: Σύνταξη ερωτήματος (query composition)

Ένα από τα βασικά μειονεκτήματα των CSEs είναι ότι απαιτούν τη δημιουργία ερωτημάτων σε συγκεκριμένη μορφή. Ένα RSSE θα πρέπει να σχηματίζει το απαιτούμενο ερώτημα χρησιμοποιώντας μια γλώσσα την οποία κατανοούν οι προγραμματιστές ή ακόμα καλύτερα να το εξάγει αυτόματα από τον κώδικα του προγραμματιστή.

### Χαρακτηριστικό 2: Εξόρυξη με βάση τη σύνταξη (syntax-aware mining)

Η σχεδίαση και ενσωμάτωση ενός μοντέλου εξόρυξης με βάση τη σύνταξη που επικεντρώνεται στην αναζήτηση κώδικα, αντί για ένα μοντέλο γενικού σκοπού (π.χ. που βασίζεται σε λέξεις-κλειδιά), εγγυάται καλύτερα αποτελέσματα.

<sup>4</sup><http://opensearch.krugle.org/>

**Χαρακτηριστικό 3: Αυτόματος μετασχηματισμός κώδικα (code transformation)**

Η τροποποίηση του κώδικα που ανακτάται θα μπορούσε να οδηγήσει σε αποτελέσματα που είναι μεταγλωττίσιμα. Υπάρχουν περιπτώσεις όπου οι απαιτούμενοι μετασχηματισμοί είναι απλοί, π.χ. όταν χρειάζεται να αλλάξει κάποιο όνομα μεθόδου.

**Χαρακτηριστικό 4: Αξιολόγηση αποτελεσμάτων (results assessment)**

Είναι σημαντικό το σύστημα να παρέχει πληροφορίες σχετικά με τα αποτελέσματα που ανακτούνται, τόσο σε επίπεδο λειτουργικότητας όσο και σε επίπεδο ποιότητας. Έτσι, τα αποτελέσματα θα πρέπει να ταξινομηθούν ανάλογα με το αν είναι σχετικά με το ερώτημα του προγραμματιστή.

**Χαρακτηριστικό 5: Αναζήτηση σε δυναμικά αποθετήρια (dynamic repositories)**

Η αναζήτηση σε αναπτυσσόμενα αποθετήρια αντί για χρήση τοπικών αποθετηρίων ή ακόμα και στατικών CSEs οδηγεί σε πιο ενημερωμένα αποτελέσματα. Έτσι, τα RSSEs πρέπει να συνδέονται με ενημερωμένα ευρετήρια.

**6.2.5 Χαρακτηριστικά του Mantissa**

Μετά την παρουσίαση της υπάρχουσας βιβλιογραφία στα συστήματα TDR, συγκρίνουμε αυτά τα συστήματα με το Mantissa με βάση τα χαρακτηριστικά που ορίστηκαν στην ενότητα 6.2.4. Η σύγκριση, η οποία παρουσιάζεται στον Πίνακα 6.1, δείχνει ότι το Mantissa καλύπτει όλα τα απαιτούμενα χαρακτηριστικά/κριτήρια, ενώ τα άλλα συστήματα περιορίζονται συνήθως στην πλήρη κάλυψη δύο ή τριών από αυτά τα χαρακτηριστικά.

Πίνακας 6.1: Σύγκριση Χαρακτηριστικών Δημοφιλών Συστημάτων Επαναχρησιμοποίησης με το Mantissa

Σύστημα	Χαρακτ. 1 Σύνταξη Ερωτήματος	Χαρακτ. 2 Αναγνώριση σύνταξης	Χαρακτ. 3 Μετασχηματισ- μός κώδικα	Χαρακτ. 4 Αξιολόγηση αποτελεσμάτων	Χαρακτ. 5 Δυναμικά αποθετήρια
Mantissa	✓	✓	✓	✓	✓
Code Conjurer	✓	✓	✗	✗	×
CodeGenie	✓	✓	×	✗	×
S6	×	✓	✓	✗	✓
FAST	✗	✓	×	✗	×

✓: Υποστηρίζεται    ×: Δεν υποστηρίζεται    ✗: Υποστηρίζεται μερικώς

Τα ερωτήματα με βάση τη σύνταξη (χαρακτηριστικό 2) υποστηρίζονται από όλα τα συστήματα. Ωστόσο, ορισμένα RSSEs, όπως το S6 και το FAST, απαιτούν από τον προγραμματιστή να κατασκευάσει το ερώτημα σε μια συγκεκριμένη μορφή με την οποία μπορεί να μην είναι εξοικειωμένος. Αντίθετα, το Mantissa υποστηρίζει την αυτόματη δημιουργία ερωτημάτων (χαρακτηριστικό 1), καθώς εξάγει το ερώτημα από τον πηγαίο κώδικα του προγραμματιστή, χωρίς να απαιτείται εξοικείωση με κάποια γλώσσα. Τα συστήματα Code Conjurer και CodeGenie είναι επίσης αρκετά αποτελεσματικά για την εξαγωγή του ερωτήματος του προγραμματιστή, ωστόσο δεν υποστηρίζουν πλήρως τους αυτοματοποιημένους μετασχηματισμούς κώδικα (χαρακτηριστικό 3), έτσι μπορεί να παραλείπουν χρήσιμα αποτελέσματα. Το Mantissa, από την άλλη πλευρά, εκτελεί μετασχηματισμούς κώδικα προκειμένου να ταιριάξει όσο το δυνατό περισσότερα αποτελέσματα με το ερώτημα.

Η αξιολόγηση των αποτελεσμάτων είναι ένα άλλο κρίσιμο χαρακτηριστικό για τα RSSEs επαναχρησιμοποίησης κώδικα (χαρακτηριστικό 4). Όσον αφορά τα συστήματα TDR, αυτό το χαρακτηριστικό συνήθως μεταφράζεται στην αξιολόγηση του κατά πόσο τα αποτελέσματα περνάνε επιτυχώς από τις περιπτώσεις ελέγχου που παρέχονται ή τουλάχιστον αν μεταγλωττίζονται μαζί με τις περιπτώσεις ελέγχου. Τα περισσότερα συστήματα TDR, συμπεριλαμβανομένου του Mantissa, καλύπτουν αποτελεσματικά αυτό το χαρακτηριστικό. Ωστόσο, το Mantissa επεξεργάζεται περαιτέρω τα ανακτημένα τμήματα για να παρέχει χρήσιμες πληροφορίες που υποδεικνύουν όχι μόνο την προέλευσή τους (GitHub repo), αλλά και τη ροή ελέγχου (control flow) τους καθώς και σχετικές εξωτερικές εξαρτήσεις (external dependencies). Ως εκ τούτου, οι προγραμματιστές μπορούν εύκολα να κατανοήσουν την εσωτερική λειτουργία ενός τμήματος και να το ενσωματώσουν στον πηγαίο κώδικα τους. Ένα άλλο πολύ σημαντικό χαρακτηριστικό του Mantissa είναι η σύνδεσή της με αποθετήρια λογισμικού που ενημερώνονται δυναμικά (χαρακτηριστικό 5). Αυτό εξασφαλίζει ότι τα αποτελέσματα είναι πάντα ενημερωμένα. Όπως φαίνεται στον Πίνακα 6.1, αυτό το χαρακτηριστικό συνήθως παραβλέπεται στα τρέχοντα συστήματα TDR (εκτός από το S6). Συμπερασματικά, υποστηρίζουμε ότι κανένα από τα συστήματα που αναλύσαμε και κανένα άλλο εξ όσων γνωρίζουμε δεν καλύπτει αποτελεσματικά τα χαρακτηριστικά που ορίζονται στην ενότητα 6.2.4. Στο επόμενο υποκεφάλαιο παρουσιάζουμε το σύστημά μας και τη λειτουργικότητά του.

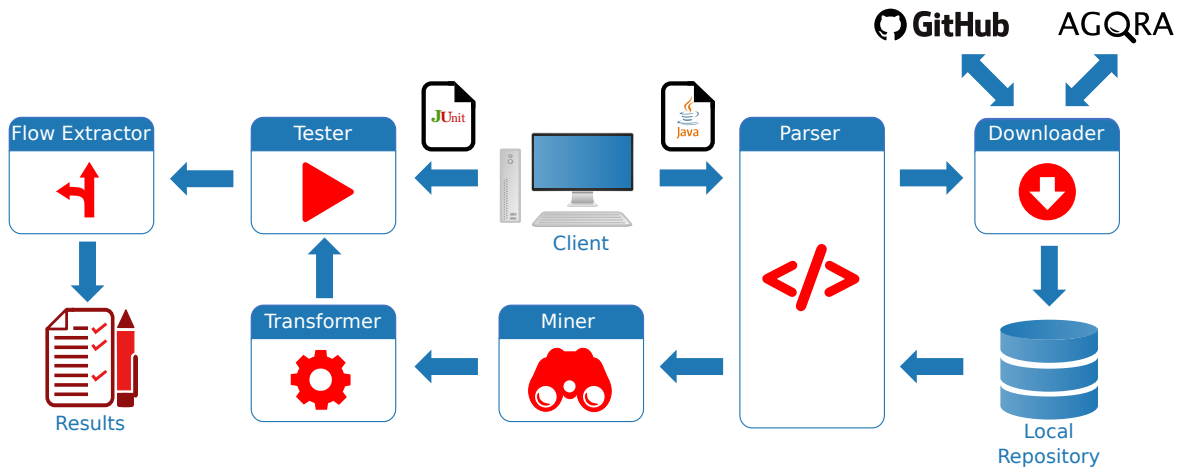
## 6.3 Το Σύστημα Επαναχρησιμοποίησης Κώδικα Mantissa

Σε αυτό το υποκεφάλαιο παρουσιάζουμε το Mantissa [206], το TDR σύστημα που αναπτύξαμε, το οποίο υποστηρίζει τα χαρακτηριστικά που αναλύθηκαν στην ενότητα 6.2.4.

### 6.3.1 Επισκόπηση

Η αρχιτεκτονική του Mantissa φαίνεται στο Σχήμα 6.1. Το Mantissa περιλαμβάνει 6 τμήματα: τον *Downloader*, τον *Parser*, τον *Miner*, τον *Transformer*, τον *Tester* και τον *Flow Extractor*. Ο προγραμματιστής παρέχει το ερώτημα στην είσοδο ως *υπογραφή (signature)* πηγαίου κώδικα. Μια υπογραφή είναι αρκετά παρόμοια με μια διεπαφή (interface) για ένα τμήμα· περιλαμβάνει όλες τις μεθόδους. Επιπλέον, ο προγραμματιστής πρέπει να παρέχει την αντίστοιχη περίπτωση ελέγχου για το τμήμα. Αρχικά ο *Parser* λαμβάνει την υπογραφή και την αναλύει (parsing) για να εξάγει το AST. Ο *Downloader* χρησιμοποιεί το AST για να κατασκευάσει τα ερωτήματα για τις συνδεδεμένες CSEs. Μετά από αυτό, κι εφόσον εκτελεστεί κάποιο ερώτημα, ο *Downloader* κατεβάζει τα αποτελέσματα και τα αποθηκεύει σε ένα τοπικό αποθετήριο (*Local Repository*).

Τα αποτελέσματα αναλύονται στη συνέχεια από τον *Parser*, ο οποίος εξάγει το AST για κάθε αποτέλεσμα. Ο *Miner* κατατάσσει τα αποτελέσματα χρησιμοποιώντας ένα μοντέλο βαθμολόγησης με βάση τη σύνταξη. Στη συνέχεια, τα αρχεία μεταφέρονται στον *Transformer*, που εκτελεί μετασχηματισμούς κώδικα προκειμένου να τα αντιστοιχίσει συντακτικά με το ερώτημα. Ο *Tester* μεταγλωττίζει τα αρχεία μαζί με τις περιπτώσεις ελέγχου και στη συνέχεια τα εκτελεί. Η λίστα κατάταξης των αποτελεσμάτων περιλαμβάνει τη βαθμολογία για κάθε τμήμα κώδικα, μαζί με πληροφορίες σχετικά με το εάν το κάθε αποτέλεσμα είναι μεταγλωττίσιμο και αν έχει περάσει τον έλεγχο. Τέλος, η ροή ελέγχου και οι εξαρτήσεις



Σχήμα 6.1: Αρχιτεκτονική του Mantissa

για κάθε αποτέλεσμα εξάγονται από τον Flow Extractor, ενώ για κάθε εξάρτηση/δήλωση βιβλιοθήκης παρέχονται σύνδεσμοι στην υπηρεσία grepcode<sup>5</sup>.

Σημειώστε ότι η αρχιτεκτονική του Mantissa είναι ανεξάρτητη από τη γλώσσα προγραμματισμού, καθώς μπορεί να προτείνει τμήματα κώδικα λογισμικού γραμμένα σε διαφορετικές γλώσσες. Συγκεκριμένα, η υποστήριξη μιας γλώσσας προγραμματισμού απαιτεί την ανάπτυξη ενός Parser για την εξαγωγή των ASTs των τμημάτων κώδικα και ενός Tester έτσι ώστε να ελέγχονται τα αποτελέσματα. Στις επόμενες παραγράφους, σχεδιάζουμε μια υλοποίηση του Mantissa για τη γλώσσα προγραμματισμού Java.

Αρχικά, η υπογραφή εισόδου του Mantissa δίνεται σε μορφή παρόμοια με μια διεπαφή Java και οι περιπτώσεις ελέγχου γράφονται σε JUnit. Ένα παράδειγμα υπογραφής για μια δομή δεδομένων στοιβάς (“Stack”) με δύο μεθόδους “push” και “pop” εμφανίζεται στο Σχήμα 6.2. Σημειώστε επίσης ότι ένας wildcard χαρακτήρας (\_) μπορεί να χρησιμοποιη-

---

```
public class Stack {
    public void push(Object o){}
    public Object pop(){}
```

---

Σχήμα 6.2: Παράδειγμα υπογραφής για μια κλάση “Stack” με μεθόδους “push” και “pop”

θεί στην περίπτωση που ο χρήστης θα ήθελε να αγνοήσει το όνομα κλάσης/μεθόδου, ενώ μπορούν επίσης να χρησιμοποιηθούν γενικοί τύποι Java (Java generics).

### 6.3.2 Parser

Το τμήμα του αναλυτή κώδικα (Parser) χρησιμοποιείται για την εξαγωγή πληροφοριών από τον πηγαίο κώδικα του προγραμματιστή καθώς και για την εξαγωγή των AST των τμημάτων κώδικα που έχουν ανακτηθεί. Ο Parser χρησιμοποιεί το srcML<sup>6</sup>, ένα εργαλείο που

<sup>5</sup><http://grepcode.com/>

<sup>6</sup><http://www.srcml.org/>

εξάγει το AST σε μορφή XML, και στο οποίο οι ετικέτες (tags) επισημαίνουν στοιχεία σύνταξης για τη γλώσσα. Το εργαλείο επιτρέπει επίσης τη διενέργεια refactoring μετατρέποντας τα δένδρα XML πίσω σε πηγαίο κώδικα. Το AST για ένα αρχείο περιλαμβάνει όλα τα στοιχεία που μπορούν να οριστούν στο αρχείο: πακέτα (packages), δηλώσεις βιβλιοθηκών (imports), κλάσεις (classes), μεθόδους (methods), κ.λπ. Ένα απλοποιημένο παράδειγμα AST (χωρίς modifiers) για το αρχείο του Σχήματος 6.2 φαίνεται στο Σχήμα 6.3.

---

```

<class>
  <name>Stack</name>
  <method>
    <name>push</name>
    <parameters>
      <type>Object</type>
    </parameters>
    <return>void</return>
  </method>
  <method>
    <name>pop</name>
    <return>Object</return>
  </method>
</class>

```

---

Σχήμα 6.3: Παράδειγμα AST για τον πηγαίο κώδικα του Σχήματος 6.2

### 6.3.3 Downloader

Ο Downloader χρησιμοποιείται για την αναζήτηση στις CSEs με τις οποίες συνδέεται το σύστημα, για την ανάκτηση σχετικών αρχείων και τη δημιουργία ενός προσωρινού τοπικού αποθετηρίου (local repository), έτσι ώστε τα αρχεία να αναλυθούν στη συνέχεια από το τμήμα του Miner. Από τον Downloader, παρέχεται η δυνατότητα αναζήτησης σε δύο CSEs, τη CSE του GitHub και την AGORA που αναλύθηκε στο προηγούμενο κεφάλαιο. Παρουσιάζουμε την ενσωμάτωση των δύο CSEs στην παρακάτω υποενότητα.

#### 6.3.3.1 Σύνδεση με Μηχανές Αναζήτησης Κώδικα

Όσον αφορά τη σύνδεση του συστήματός μας με το GitHub, το API του GitHub για αναζήτηση κώδικα έχει δύο σημαντικούς περιορισμούς: α) τον περιορισμό σε ένα μέγιστο αριθμό αιτήσεων ανά ώρα, και β) τον περιορισμό της δυνατότητας αναζήτησης κώδικα σε συγκεκριμένα αποθετήρια ή λογαριασμούς, μη επιτρέποντας έτσι την αναζήτηση σε ολόκληρο το ευρετήριο. Για το λόγο αυτό χρειάστηκε να προσαρμόσουμε τον Downloader σε αυτήν την περίπτωση για να ξεπεραστεί ο δεύτερος περιορισμός.

Η υλοποίησή μας χρησιμοποιεί το Searchcode<sup>7</sup>, προκειμένου να εντοπίσει αρχικά τους λογαριασμούς του GitHub των οποίων τα αποθετήρια είναι πιθανό να είναι σχετικοί με το

<sup>7</sup><https://searchcode.com/>



ερώτημα. Ειδικότερα, ο Downloader χρησιμοποιεί το AST της υπογραφής εισόδου προκειμένου να κατασκευάσει έναν σύνολο ερωτημάτων για το Searchcode. Αυτά τα ερωτήματα εκτελούνται το ένα μετά το άλλο μέχρις ότου επιστραφεί ένα προκαθορισμένο πλήθος αποτελεσμάτων από διαφορετικούς λογαριασμούς του GitHub (οι δοκιμές έδειξαν ότι η ανάκτηση των πρώτων 30 λογαριασμών είναι επαρκής). Το τμήμα του Downloader εκτελεί τους εξής τρεις τύπους ερωτημάτων:

- το όνομα της κλάσης και ο συνδυασμός των μεθόδων, συμπεριλαμβανομένων των ονομάτων τους και των τύπων επιστροφής τους,
- το όνομα της κλάσης ως έχει καθώς και χωρισμένο σε όρους (tokenized) εάν μπορεί να χωριστεί σε περισσότερους από έναν όρους, και
- ο συνδυασμός των μεθόδων (χωρίς κάποιο όνομα κλάσης), μαζί με το όνομα και τον τύπο επιστροφής κάθε μεθόδου.

Στο Σχήμα 6.4 φαίνεται μια λίστα ερωτημάτων για την υπογραφή τμήματος “Stack” του Σχήματος 6.2.

---

Query 1: Stack void push Object pop  
 Query 2: Stack void push  
 Query 3: Stack Object pop  
 Query 4: Stack  
 Query 5: void push Object pop  
 Query 6: void push  
 Query 7: Object pop

---

Σχήμα 6.4: Παράδειγμα αλληλουχίας ερωτημάτων για το Searchcode

Μετά την εκτέλεση των ερωτημάτων στο Searchcode, ο Downloader προσδιορίζει 30 λογαριασμούς χρηστών που είναι κατάλληλοι για το ερώτημα. Δεδομένου όμως ότι το GitHub επιτρέπει την αναζήτηση περισσότερων από 30 χρηστών, αποφασίσαμε να το εκμεταλλευτούμε προσθέτοντας και τους πιο δημοφιλείς χρήστες σε αυτή τη λίστα (όπου η δημοτικότητα καθορίζεται με βάση τον αριθμό των stars των αποθετηρίων τους). Συγκεκριμένα, το GitHub επιτρέπει την εκτέλεση ερωτημάτων στον πηγαίο κώδικα έως και 900 λογαριασμών χρηστών ανά λεπτό, δηλαδή 30 αιτήματα (requests) ανά λεπτό κάθε ένα εκ των οποίων περιλαμβάνει 30 χρήστες.

Όμοια με το Searchcode, το GitHub δεν υποστηρίζει κανονικές εκφράσεις, έτσι ο Downloader κατασκευάζει μια λίστα από ερωτήματα με σκοπό την ανάκτηση όσο το δυνατόν περισσότερων αποτελεσμάτων. Η λίστα ερωτημάτων είναι ελαφρώς διαφορετική από αυτή που χρησιμοποιήθηκε για το Searchcode, καθώς στην περίπτωση του GitHub απαιτούνται τα ίδια τα αρχεία και όχι μόνο τα αποθετήρια των χρηστών του GitHub. Συγκεκριμένα, τα ερωτήματα προς το GitHub περιλαμβάνουν:

- τις μεθόδους του τμήματος, όπου για κάθε μέθοδο περιλαμβάνεται ο τύπος επιστροφής της, το όνομά της και οι τύποι των παραμέτρων της,

- τα ονόματα των μεθόδων,
- τα ονόματα των μεθόδων διαχωρισμένα σε όρους (tokenized), και
- το όνομα της κλάσης διαχωρισμένο σε όρους (tokenized).

Ένα παράδειγμα μιας λίστας ερωτημάτων για το τμήμα “Stack” του Σχήματος 6.2 φαίνεται στο Σχήμα 6.5.

---

Query 1: void push(Object) Object pop()

Query 2: push(Object) pop()

Query 3: push pop

Query 4: Stack

---

Σχήμα 6.5: Παράδειγμα αλληλουχίας ερωτημάτων για το GitHub

Τέλος, αν το ερώτημα είναι μεγαλύτερο από 128 χαρακτήρες, τότε οι όροι του καταργούνται ένα κάθε φορά ξεκινώντας από τον τελευταίο έως ότου ικανοποιηθεί αυτός ο περιορισμός του GitHub.

Όπως φαίνεται από τα παραπάνω, το GitHub έχει διάφορους περιορισμούς στο API του. Έτσι, χρησιμοποιούμε επίσης την AGORA, η οποία είναι καλύτερα προσανατολισμένη στην επαναχρησιμοποίηση τμημάτων. Η αναζήτηση με βάση τη σύνταξη που προσφέρει η AGORA επιτρέπει την εύρεση αντικειμένων Java με συγκεκριμένα ονόματα μεθόδων, ονόματα παραμέτρων κ.λπ., ενώ το API της δεν έχει περιορισμούς όσον αφορά την αναζήτηση σε όλο το ευρετήριο. Στην περίπτωση μας, ο Downloader χρησιμοποιεί τη δυνατότητα αναζήτησης με βάση τη σύνταξη της AGORA για να δημιουργήσει ερωτήματα όπως αυτά που εμφανίζονται στο σενάριο αναζήτησης επαναχρησιμοποιήσιμων τμημάτων της AGORA (βλέπε υποενότητα 5.4.2.1).

### 6.3.3.2 Snippets Miner

Για λόγους εύρους ζώνης (bandwidth), οι περισσότερες CSEs συνήθως περιορίζουν τους χρήστες όσον αφορά τον αριθμό των αποτελεσμάτων που μπορούν να ληφθούν μέσα σε ένα χρονικό περιθώριο. Για παράδειγμα, στην περίπτωση του GitHub, η λήψη όλων των αποτελεσμάτων ένα προς ένα δεν είναι αποδοτική. Ωστόσο, οι CSEs συνήθως επιστρέφουν μια λίστα αποτελεσμάτων, συμπεριλαμβανομένων και αποσπασμάτων για κάθε αποτέλεσμα για να δείξουν σε ποιο σημείο το αποτέλεσμα είναι σχετικό με το ερώτημα. Για να επιταχύνουμε το σύστημά μας και να διασφαλίσουμε ότι το αποτέλεσμα είναι τουλάχιστον κάπως σχετικό πριν το κατέβασμά του, έχουμε δημιουργήσει έναν εξορυκτή αποσπασμάτων (*Snippets Miner*) μέσα στον Downloader. Ο Snippets Miner αναλύει τα αποσπάσματα που παρέχονται από τη CSE και καθορίζει ποια αποτελέσματα είναι σχετικά με το ερώτημα, διασφαλίζοντας ότι ο Downloader θα κατεβάσει μόνο αυτά και θα αγνοήσει τα υπόλοιπα αρχεία.

Σημειώστε ότι ο Snippets Miner μπορεί να προσαρμοστεί σε οποιαδήποτε CSE που επιστρέφει αποσπάσματα για τα αποτελέσματα. Καθώς το τμήμα αυτό εφαρμόζει αντιστοίχιση

με βάση τη σύνταξη, δεν ήταν απαραίτητη η χρήση του για την AGORA που έχει ήδη δυνατότητες αντιστοίχισης. Ο Snippets Miner, ωστόσο, έπρεπε να ενεργοποιηθεί για τα αποτελέσματα του GitHub.

Αρχικά, ο Snippets Miner αφαιρεί τα διπλότυπα χρησιμοποιώντας το πλήρες όνομα αρχείου για κάθε αποτέλεσμα (συμπεριλαμβανομένου του ονόματος του έργου) καθώς και το *sha hash* που επιστρέφεται από το GitHub API για κάθε αποτέλεσμα. Μετά από αυτό, εξάγονται οι μέθοδοι για κάθε αποτέλεσμα και ελέγχεται αν συμφωνούν με το αρχικό ερώτημα. Σημειώστε ότι η χρήση του Parser δεν είναι δυνατή αφού τα αποσπάσματα είναι ελλιπή. Έτσι, οι δηλώσεις μεθόδων εξάγονται χρησιμοποιώντας την κανονική έκφραση του Σχήματος 6.6.

$$([\w\<\>\[\[\]]+)?\s+(\w+)?\((.*?)\)\s*(throws\s*\w+)*\{$$

τύπος επιστροφής
όνομα
παράμετροι
σώμα

Σχήμα 6.6: Κανονική έκφραση για την εξαγωγή δηλώσεων μεθόδων από τμήματα κώδικα

Μετά την εξαγωγή των δηλώσεων μεθόδων, ο Snippets Miner πραγματοποιεί αντιστοίχιση μεταξύ των μεθόδων του αποσπάσματος που εξετάζεται και των μεθόδων που ζητούνται στο ερώτημα του προγραμματιστή. Ο μηχανισμός αντιστοίχισης ελέγχει αν υπάρχουν παρόμοια ονόματα μεθόδων, τύποι επιστροφής μεθόδων, καθώς και παρόμοιες παράμετροι (ονόματα και τύποι). Οι τύποι επιστροφής αντιστοιχίζονται μόνο όταν είναι ακριβώς ίδιοι, ενώ για τις παραμέτρους λαμβάνονται υπόψη οι πιθανές αντιμεταθέσεις (permutations), καθώς μπορεί να είναι σε διαφορετική σειρά. Σε κάθε αποτέλεσμα, δίνεται μια συνολική βαθμολογία χρησιμοποιώντας την τιμές αντιστοίχισης για τα παραπάνω στοιχεία. Ο τρόπος βαθμολόγησης περιγράφεται λεπτομερώς στην ενότητα 6.3.4, καθώς είναι ίδιος με αυτόν του Miner του Mantissa. Ο Snippets Miner χρησιμοποιείται για να διατηρεί τα αποτελέσματα που με βαθμολογία μεγαλύτερη του 0 και να απορρίπτει τα υπόλοιπα αρχεία. Αν, ωστόσο, το πλήθος των αποτελεσμάτων που ανακτώνται είναι μικρότερο από ένα όριο (στην περίπτωση μας 100), τότε αποτελέσματα με τιμή 0 επίσης κατεβαίνουν μέχρι το πλήθος των αποτελεσμάτων να φτάσει το όριο.

## 6.3.4 Miner

Μετά το σχηματισμό ενός τοπικού αποθετηρίου με τα αποτελέσματα που ανακτήθηκαν, ο Parser εξάγει τα AST τους, όπως περιγράφεται στην ενότητα 6.3.2, ενώ ο Miner τα κατατάσσει σύμφωνα με το ερώτημα του προγραμματιστή. Ο Miner αποτελείται από τρία υποτμήματα: τον *Preprocessor*, τον *Scorer* και τον *Postprocessor*. Αυτά αναλύονται στις ακόλουθες υποενότητες.

### 6.3.4.1 Preprocessor

Το πρώτο βήμα πριν την κατάταξη των αποτελεσμάτων περιλαμβάνει την ανίχνευση και την αφαίρεση διπλότυπων (duplicates). Υπάρχουν αρκετοί αλγόριθμοι για την ανίχνευση διπλότυπων, ενώ η τρέχουσα βιβλιογραφία περιλαμβάνει επίσης διάφορες εξελιγμένες τεχνικές ανίχνευσης κλώνων κώδικα (code clone detection). Ωστόσο, αυτές οι μέθοδοι συνήθως δεν προσανατολίζονται στην επαναχρησιμοποίηση του κώδικα· κατά την αναζήτηση για ένα

τμήμα λογισμικού, η εύρεση παρόμοιων αποτελεσμάτων είναι στην πραγματικότητα ενθαρρυντική, και αυτό που θεωρείται ως πλεονάζουσα πληροφορία είναι τα *ακριβή διπλότυπα* (*exact duplicates*). Έτσι, ο Preprocessor εξαλείφει τα διπλότυπα χρησιμοποιώντας τον αλγόριθμο MD5 [207] για να διασφαλίσει ότι η διαδικασία είναι όσο το δυνατόν γρηγορότερη. Αρχικά, τα αρχεία σαρώνονται και δημιουργείται ένας MD5 hash για καθένα από αυτά και στη συνέχεια συγκρίνονται τα hashes για τον εντοπισμό διπλότυπων.

### 6.3.4.2 Scorer

Ο Scorer είναι το βασικό υποτμήμα του Miner, που βαθμολογεί και κατατάσσει τα αρχεία. Λαμβάνει τα ASTs των ληφθέντων αρχείων ως είσοδο και υπολογίζει την ομοιότητα μεταξύ καθενός από τα αποτελέσματα με το AST του ερωτήματος. Η μεθοδολογία βαθμολόγησης είναι παρόμοια με το *Μοντέλο Διανυσματικού Χώρου* (*Vector Space Model - VSM*) [72], όπου τα *έγγραφα* είναι αρχεία κώδικα, ενώ οι *όροι* είναι τιμές των κόμβων του AST (XML tags), έτσι όπως αυτό ορίζεται στην ενότητα 6.3.2. Ο Scorer δημιουργεί ένα διάνυσμα (vector) για κάθε αποτέλεσμα και ένα διάνυσμα για το ερώτημα και πραγματοποιεί μια σύγκριση μεταξύ των δύο διανυσμάτων.

Δεδομένου ότι κάθε αρχείο έχει μια κλάση, αρχικά δημιουργείται ένα διάνυσμα κλάσης για κάθε αρχείο. Το διάνυσμα κλάσης έχει την ακόλουθη μορφή:

$$\vec{c} = [score(name), score(\vec{m}_1), score(\vec{m}_2), \dots, score(\vec{m}_n)] \quad (6.1)$$

όπου *name* είναι το όνομα της κλάσης και  $\vec{m}_i$  είναι το διάνυσμα της *i*-ης μεθόδου της κλάσης, από τις *n* μεθόδους συνολικά. Η βαθμολογία του ονόματος είναι η τιμή ενός μέτρου ομοιότητάς του με το όνομα του ερωτήματος, ενώ η βαθμολογία κάθε διανύσματος μεθόδου είναι η τιμή ενός μέτρου ομοιότητάς του με το διάνυσμα μεθόδου του ερωτήματος, το οποίο περιέχει το όνομα της μεθόδου, τον τύπο επιστροφής της και τους τύπους των παραμέτρων της. Ένα διάνυσμα μεθόδου  $\vec{m}$  ορίζεται ως:

$$\vec{m} = [score(name), score(type), score(p_1), score(p_2), \dots, score(p_m)] \quad (6.2)$$

όπου *name* είναι το όνομα της μεθόδου, *type* είναι ο τύπος επιστροφής της, και  $p_j$  είναι ο τύπος της *j*-ης παραμέτρου της μεθόδου, από τις *m* παραμέτρους συνολικά.

Χρησιμοποιώντας τις εξισώσεις (6.1) και (6.2), είμαστε σε θέση να αναπαραστήσουμε την υπογραφή ενός τμήματος ως ένα διάνυσμα κλάσης. Επομένως, για τη σύγκριση δύο τμημάτων αρκεί να συγκρίνουμε αυτά τα διανύσματα που περιέχουν αριθμητικές τιμές στο εύρος [0, 1]. Οι συναρτήσεις *score* υπολογίζονται για κάθε έγγραφο-αρχείο σε σχέση με την υπογραφή του ερωτήματος. Για τον υπολογισμό των τιμών των διανυσμάτων των εξισώσεων (6.1) και (6.2) απαιτείται μια συνάρτηση που να υπολογίζει την ομοιότητα μεταξύ συνόλων, για τις μεθόδους και τις παραμέτρους, και μια συνάρτηση που να υπολογίζει την ομοιότητα μεταξύ συμβολοσειρών, για ονόματα κλάσεων, ονόματα μεθόδων και τύπους επιστροφής, και τύπους παραμέτρων.

Σημειώνουμε ότι η χρήση μιας μεθόδου αντιστοίχισης συμβολοσειρών (string matching method) δεν είναι επαρκής, καθώς η μέγιστη βαθμολογία μεταξύ δύο διανυσμάτων όπως αυτά των εξισώσεων (6.1) και (6.2) εξαρτάται επιπλέον από τη σειρά των στοιχείων. Οι μέθοδοι, για παράδειγμα, δεν είναι βέβαιο ότι έχουν την ίδια σειρά στα δύο αρχεία, όπως επίσης και οι παράμετροι δεν έχουν την ίδια σειρά στις μεθόδους. Επομένως, ο Scorer πρέπει

να βρει την καλύτερη δυνατή αντιστοίχιση μεταξύ δύο συνόλων από μεθόδους ή παραμέτρους αντίστοιχα. Η εύρεση της καλύτερης δυνατής αντιστοίχισης μεταξύ δύο συνόλων είναι ένα πρόβλημα παρόμοιο με αυτό του *Προβλήματος του Σταθερού Γάμου (Stable Marriage Problem - SMP)* [208], όπου δύο σύνολα στοιχείων πρέπει να αντιστοιχηθούν έτσι ώστε να μεγιστοποιηθεί το κέρδος με βάση της προτιμήσεις των στοιχείων του πρώτου συνόλου προς τα στοιχεία του δεύτερου συνόλου και το αντίστροφο<sup>8</sup>. Ωστόσο, στην περίπτωση μας, το πρόβλημα είναι απλούστερο, καθώς οι λίστες προτιμήσεων για τα δύο σύνολα είναι συμμετρικές. Συνεπώς, κατασκευάσαμε έναν αλγόριθμο που βρίσκει την καλύτερη δυνατή αντιστοίχιση μεταξύ των στοιχείων δύο συνόλων. Ο αλγόριθμος *SetsMatching* παρουσιάζεται στο Σχήμα 6.7. Ο αλγόριθμος λαμβάνει ως είσοδο τα σύνολα  $U$  και  $V$  και δίνει ως έξοδο το *MatchedPairs*, δηλαδή την καλύτερη αντιστοίχιση μεταξύ των δύο συνόλων.

---

```

SetsMatching( $U, V$ )
    MatchedU = {}
    MatchedV = {}
    MatchedPairs = {}
    Pairs = {( $u, v, score(u, v)$ )  $\forall u \in U, v \in V$ }
    Sort Pairs in descending order according to their score
    for each ( $u, v, score(u, v)$ )  $\in$  Pairs:
        if  $u \notin$  MatchedU and  $v \notin$  MatchedV
            MatchedPairs = MatchedPairs  $\cup$  {( $u, v, score(u, v)$ )}
    return MatchedPairs

```

---

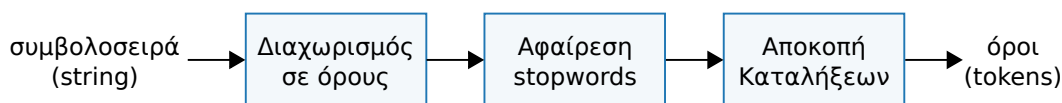
Σχήμα 6.7: Αλγόριθμος που υπολογίζει την ομοιότητα μεταξύ δύο συνόλων από στοιχεία,  $U$  και  $V$ , και επιστρέφει την καλύτερη δυνατή αντιστοίχιση ως ένα σύνολο από ζεύγη *MatchedPairs*

Αρχικά, ορίζονται δύο νέα σύνολα *MatchedU* και *MatchedV*, τα οποία θα περιέχουν τα αντιστοιχισμένα στοιχεία των δύο αρχικών συνόλων. Στη συνέχεια, για όλους τους πιθανούς συνδυασμούς των στοιχείων των δύο συνόλων υπολογίζονται οι βαθμολογίες και αποθηκεύονται στο σύνολο *Pairs* στη μορφή ( $u, v, score(u, v)$ ), όπου τα  $u$  και  $v$  είναι στοιχεία των συνόλων  $U$  και  $V$ , αντίστοιχα. Το σύνολο *Pairs* ταξινομείται σε φθίνουσα σειρά σύμφωνα με τις βαθμολογίες, και στη συνέχεια ο αλγόριθμος διατρέχει κάθε ζεύγος (pair) του συνόλου. Για κάθε ζεύγος, αν κανένα από τα στοιχεία του δεν ανήκει ήδη στα σύνολα *MatchedU* και *MatchedV*, τότε το ζεύγος αντιστοιχίζεται και προστίθεται στο σύνολο *MatchedPairs*.

Αφού αντιμετωπίσαμε το πρόβλημα της αντιστοίχισης μεταξύ κλάσεων και μεταξύ μεθόδων, αυτό που απομένει είναι να σχεδιάσουμε μια συνάρτηση ομοιότητας συμβολοσειρών. Η συνάρτηση θα χρησιμοποιηθεί για τον υπολογισμό της ομοιότητας μεταξύ των ονομάτων και των τύπων. Σημειώστε ότι η μεθοδολογία μας πρέπει να αντικατοπτρίζει τα κύρια χαρακτηριστικά της γλώσσας Java, π.χ. να υποστηρίζει συμβολοσειρές camelCase, ενώ συγχρόνως να χρησιμοποιεί σημασιολογικές τεχνικές (π.χ. stemming). Έτσι, το πρώτο βήμα εί-

<sup>8</sup>Σύμφωνα με τον ορισμό του SMP, υπάρχουν  $N$  άνδρες και  $N$  γυναίκες, και κάθε άτομο κατατάσσει τα μέλη του αντίθετου φύλου σε μια αυστηρή σειρά προτίμησης. Το πρόβλημα είναι η εύρεση μιας αντιστοίχισης μεταξύ ανδρών και γυναικών έτσι ώστε να μην υπάρχουν δύο άτομα του αντίθετου φύλου που και τα δύο να προτιμούσαν να έχουν κάποιο ταίρι που είναι διαφορετικό από το υπάρχον ταίρι τους.

ναι η προεπεξεργασία των συμβολοσειρών. Τα βήματα της προεπεξεργασίας απεικονίζονται στο Σχήμα 6.8.



Σχήμα 6.8: Βήματα προεπεξεργασίας συμβολοσειρών

Αρχικά, η συμβολοσειρά διαχωρίζεται σε όρους (tokenization) σύμφωνα με τις συμβάσεις (naming conventions) της Java. Ο διαχωρισμός γίνεται σε κεφαλαία γράμματα (για συμβολοσειρές camelCase), σε αριθμούς και στον χαρακτήρα underscore (\_). Επίσης, αφαιρούνται αριθμοί και σύμβολα που χρησιμοποιούνται στους τύπος αντικειμένων (< και >) και στους πίνακες ([ και ]), ενώ τέλος οι όροι μετατρέπονται σε πεζά (lowercase).

Το επόμενο βήμα περιλαμβάνει την αφαίρεση *stopwords* και όρων με λιγότερους από 3 χαρακτήρες, καθώς δεν συνεισφέρουν στη σύγκριση. Χρησιμοποιήσαμε τη λίστα με *stopwords* του NLTK [209]. Τέλος, αποκόπτονται οι καταλήξεις των όρων (stemming) χρησιμοποιώντας τον *PortStemmer* του NLTK [209], από τον οποίο εφαρμόζονται ευριστικοί κανόνες κριτικής μορφολογίας (inflectional morphology) και παραγωγικής μορφολογίας (derivational morphology). Στον Πίνακα 6.2 δίνονται παραδείγματα εφαρμογής αυτών των βημάτων προεπεξεργασίας. Στην αριστερή στήλη φαίνεται η αρχική συμβολοσειρά, ενώ οι επόμενες στήλες δείχνουν το αποτέλεσμα μετά την εκτέλεση κάθε βήματος, οπότε η δεξιά στήλη δείχνει τους όρους που προκύπτουν.

Πίνακας 6.2: Παραδείγματα προεπεξεργασίας συμβολοσειρών

Συμβολοσειρά (string)	Διαχωρισμός σε όρους	Αφαίρεση stopwords	Αποκοπή καταλήξεων
pushing_item	pushing item	pushing item	push item
add2Numbers	add numbers	add numbers	add number
test1Me_Now	test me now	test now	test now

Τέλος, μετά την προεπεξεργασία των συμβολοσειρών, χρησιμοποιούμε μεθόδους ομοιότητας συμβολοσειρών (string similarity methods) για να υπολογίσουμε μια τιμή ομοιότητας μεταξύ δύο συμβολοσειρών. Χρησιμοποιήσαμε δύο διαφορετικές μετρικές: την *ομοιότητα Levenshtein* [210] και το *δείκτη Jaccard* [211]. Η ομοιότητα Levenshtein χρησιμοποιείται από τον *Scorer* ώστε η αντιστοίχιση μεταξύ των συμβολοσειρών να είναι αυστηρή (να λαμβάνεται υπόψη και η σειρά των όρων), ενώ ο δείκτης Jaccard χρησιμοποιείται από το υπο-μήμα *Snippets Miner* του *Downloader* (βλέπε υποενότητα 6.3.3.2) ώστε η αντιστοίχιση να είναι λιγότερο αυστηρή.

Καθώς η απόσταση Levenshtein υπολογίζεται για συμβολοσειρές, τα διανύσματα από όρους αρχικά συγχωνεύονται. Η *απόσταση Levenshtein* μεταξύ δύο συμβολοσειρών ορίζεται ως ο ελάχιστος αριθμός εισαγωγών (insertions) χαρακτήρων, διαγραφών (deletions) χαρακτήρων και αντικαταστάσεων (replacements) χαρακτήρων που απαιτούνται για να μετατραπεί η μία συμβολοσειρά στην άλλη. Για την υλοποίησή μας, το κόστος της αντικατάστασης ενός χαρακτήρα τέθηκε στην τιμή 2, ενώ τα κόστη της εισαγωγής και της διαγραφής τέθηκαν

στην τιμή 1. Έτσι, για δύο συμβολοσειρές  $s_1$  και  $s_2$ , η ομοιότητα Levenshtein υπολογίζεται ως:

$$L(s_1, s_2) = 1 - \frac{\text{distance}(s_1, s_2)}{\max\{|s_1|, |s_2|\}} \quad (6.3)$$

Όπως φαίνεται στην εξίσωση (6.3), η απόσταση μεταξύ των δύο συμβολοσειρών κανονικοποιείται στο εύρος  $[0, 1]$  διαιρώντας με το συνολικό πλήθος των χαρακτήρων που υπάρχουν στις δύο συμβολοσειρές, και αφαιρώντας στη συνέχεια το αποτέλεσμα από τη μονάδα για να προκύψει μια μετρική ομοιότητας.

Ο δείκτης Jaccard χρησιμοποιείται για τον υπολογισμό της ομοιότητας μεταξύ δύο συνόλων. Για δύο σύνολα  $U$  και  $V$ , ο δείκτης Jaccard ορίζεται ως το πηλίκο του μεγέθους της τομής (intersection) τους με το μέγεθος της ένωσής (union) τους:

$$J(U, V) = \frac{|U \cap V|}{|U \cup V|} \quad (6.4)$$

Στην περίπτωσή μας, τα σύνολα περιέχουν όρους, οπότε η τομή τους περιέχει τους όρους που ανήκουν και στα δύο σύνολα, ενώ η ένωσή τους περιέχει τους όρους και των δύο συνόλων αποκλείοντας διπλότυπους όρους.

Μετά την κατασκευή των αλγορίθμων και των μετρικών ομοιότητας συμβολοσειρών που απαιτούνται για τη δημιουργία του διάνυσματος για κάθε αποτέλεσμα, επιστρέφουμε πλέον στο VSM που ορίζεται από τις εξισώσεις (6.1) και (6.2).

Σημειώστε ότι όλες οι μετρικές που ορίστηκαν στις προηγούμενες παραγράφους είναι κανονικοποιημένες, έτσι ώστε όλες οι διαστάσεις για το διάνυσμα ενός εγγράφου να είναι στο εύρος  $[0, 1]$ . Το διάνυσμα του ερωτήματος είναι ένα διάνυσμα που όλες οι τιμές του είναι ίσες με τη μονάδα. Το τελικό βήμα είναι να υπολογιστεί μια βαθμολογία για κάθε διάνυσμα εγγράφου-αποτελέσματος και να ταξινομηθούν αυτά τα διανύσματα με βάση την ομοιότητά τους με το διάνυσμα ερωτήματος.

Καθώς ο συντελεστής Tanimoto [212] είναι αποτελεσματικός για τη σύγκριση διανυσμάτων δυαδικών ψηφίων (bit vectors), χρησιμοποιήσαμε τη συνεχή εκδοχή του για τον Scorer. Έτσι, η ομοιότητα μεταξύ του διανύσματος ερωτήματος  $\vec{Q}$  και ενός διανύσματος εγγράφου  $\vec{D}$  δίνεται από την εξίσωση:

$$T(\vec{Q}, \vec{D}) = \frac{\vec{Q} \cdot \vec{D}}{\|\vec{Q}\|^2 + \|\vec{D}\|^2 - \vec{Q} \cdot \vec{D}} \quad (6.5)$$

Ο αριθμητής της εξίσωσης (6.5) είναι το εσωτερικό γινόμενο (dot product) των δύο διανυσμάτων, ενώ ο παρονομαστής είναι το άθροισμα των ευκλείδειων νορμών (Euclidean norms) τους από το οποίο αφαιρείται το εσωτερικό τους γινόμενο. Τέλος, ο Scorer δίνει ως έξοδο τα αρχεία αποτελεσμάτων, μαζί με τις βαθμολογίες τους που υποδεικνύουν τη σχετικότητά τους με το ερώτημα.

Ως ένα παράδειγμα, μπορούμε να χρησιμοποιήσουμε την υπογραφή του Σχήματος 6.2. Η κλάση “Stack” αυτής της υπογραφής έχει δύο μεθόδους “push” και “pop”, έτσι το διάνυσμά της θα ήταν ένα διάνυσμα με τρία στοιχεία (όλα ίσα με τη μονάδα), ένα για το όνομα της κλάσης και από ένα για κάθε μέθοδο:

$$\vec{c}_{Stack} = [1, 1, 1] \quad (6.6)$$

και τα δύο διανύσματα για τις μεθόδους “push” και “pop” θα ήταν:

$$\vec{m}_{push} = [1, 1, 1] \text{ και } \vec{m}_{pop} = [1, 1] \quad (6.7)$$

αντίστοιχα, αφού η “push” έχει ένα όνομα, έναν τύπο επιστροφής (“void”) και έναν τύπο μιας παραμέτρου (“Object”), ενώ η “pop” έχει ένα όνομα και έναν τύπο μιας παραμέτρου (“Object”).

Για το παράδειγμά μας, θεωρήστε ότι εκτελείται το ερώτημα και κατεβαίνει ένα αποτέλεσμα που έχει την υπογραφή του Σχήματος 6.9.

---

```
public class MyStack {
    public bool pushObject(Object o){}
    public Object popObject(){}
```

---

Σχήμα 6.9: Παράδειγμα υπογραφής για μια κλάση στοιβάς με δύο μεθόδους για εισαγωγή και εξαγωγή στοιχείων από τη στοιβά

Αυτό το αντικείμενο έχει τα διανύσματα μεθόδων  $\vec{m}_{pushObject}$  και  $\vec{m}_{popObject}$ . Στη περίπτωση αυτή, ο αλγόριθμος αντιστοίχισης του Σχήματος 6.7 αντιστοιχίζει το  $\vec{m}_{push}$  στο  $\vec{m}_{pushObject}$  και το  $\vec{m}_{pop}$  στο  $\vec{m}_{popObject}$ . Εκτελώντας τα βήματα προεπεξεργασίας του Σχήματος 6.8 (π.χ. το “pushObject” διαχωρίζεται σε “push” και “object”, όλα τα ονόματα γίνονται πεζά κ.λπ.) και χρησιμοποιώντας την ομοιότητα Levenshtein για τις συμβολοσειρές, τα διανύσματα μεθόδων  $\vec{m}_{pushObject}$  και  $\vec{m}_{popObject}$  θα ήταν:

$$\vec{m}_{pushObject} = [0.429, 0, 1] \text{ και } \vec{m}_{popObject} = [0.429, 1] \quad (6.8)$$

αντίστοιχα. Σύμφωνα με την εξίσωση (6.5), οι βαθμολογίες για τα  $\vec{m}_{pushObject}$  και  $\vec{m}_{popObject}$  θα ήταν 0.519 και 0.814 αντίστοιχα. Επομένως, το διάνυσμα κλάσης  $\vec{c}_{MyStack}$  θα ήταν:

$$\vec{c}_{MyStack} = [1, 0.519, 0.814] \quad (6.9)$$

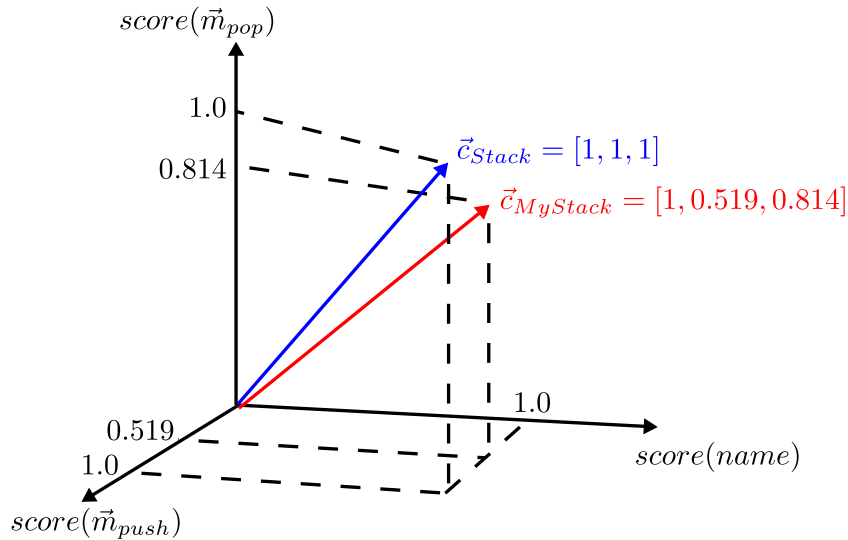
Τέλος, χρησιμοποιώντας την εξίσωση 6.5, τα διανύσματα των εξισώσεων 6.6 και 6.9 θα παρείχαν ως αποτέλεσμα μια βαθμολογία ίση με 0.898 για το αρχείο που ανακτήθηκε. Στο Σχήμα 6.10 αναπαρίστανται γραφικά τα διανύσματα κλάσεων για αυτό το παράδειγμα.

### 6.3.4.3 Postprocessor

Μετά την βαθμολόγηση των αποτελεσμάτων, ο Postprocessor χρησιμοποιείται για την κατασκευή της τελικής κατάταξης. Όταν ένα ή περισσότερα αρχεία είναι λειτουργικά ισοδύναμα, υποθέτουμε ότι ο προγραμματιστής θα επιλέξει την απλούστερη λύση. Συνεπώς, χρησιμοποιούμε τη μετρική του συνολικού αριθμού των γραμμών κώδικα (Physical Lines of Code)<sup>9</sup>, μετρώντας όλες τις γραμμές που έχουν περισσότερους από 3 μη κενούς χαρακτήρες, για να προσδιορίσουμε το μέγεθος και την πολυπλοκότητα κάθε αρχείου. Στη συνέχεια, τα αποτελέσματα κατατάσσονται σύμφωνα με τη βαθμολογία τους όπως υπολογίστηκε από τον Scorer, σε φθίνουσα σειρά, και σε περίπτωση που υπάρχουν ισοπαλίες, τα αποτελέσματα που ισοβαθούν κατατάσσονται με βάση το πλήθος των γραμμών κώδικα σε αύξουσα σειρά.

<sup>9</sup><https://java.net/projects/loc-counter/pages/Home>





Σχήμα 6.10: Παράδειγμα αναπαράστασης σε διανυσματικό χώρο για το ερώτημα “Stack” και το αποτέλεσμα “MyStack”

### 6.3.5 Transformer

Ο Transformer δέχεται ως είσοδο τα αρχεία των αποτελεσμάτων και το ερώτημα και εκτελεί μια σειρά από μετασχηματισμούς ώστε τα αρχεία να αντιστοιχίζονται με το ερώτημα. Οι μετασχηματισμοί γίνονται στο XML δένδρο του srcML χρησιμοποιώντας εκφράσεις XPath, και από το μετασχηματισμένο XML παράγεται το νέο τμήμα κώδικα.

Αρχικά, οι κλάσεις και οι μέθοδοι μετονομάζονται για να αντιστοιχούν στα στοιχεία του ερωτήματος. Οι κλήσεις μεθόδων επίσης εξετάζονται για την υποστήριξη περιπτώσεων όπου μια μέθοδος καλεί μια άλλη μέθοδο που πρόκειται να μετονομαστεί. Επιπλέον, οι τύποι επιστροφής των μεθόδων μετασχηματίζονται στους αντίστοιχους τύπους των μεθόδων του ερωτήματος, χρησιμοποιώντας μετατροπές τύπων (type casts)<sup>10</sup> όπου αυτό είναι εφικτό, ενώ οι εντολές return επίσης μετασχηματίζονται κατάλληλα.

Μετά το μετασχηματισμό των ονομάτων και των τύπων επιστροφής της κάθε μεθόδου, το επόμενο βήμα περιλαμβάνει το μετασχηματισμό των παραμέτρων. Οι παράμετροι των αποτελεσμάτων έχουν ήδη αντιστοιχιστεί στις παραμέτρους του ερωτήματος χρησιμοποιώντας τον αλγόριθμο του Σχήματος 6.7. Επομένως, ο Transformer διατρέχει όλες της παραμέτρους μιας μεθόδου του ερωτήματος και μετασχηματίζει τις αντίστοιχες παραμέτρους της σχετικής μεθόδου του αποτελέσματος. Ο μετασχηματισμός περιλαμβάνει την αλλαγή του ονόματος κάθε παραμέτρου, και την αλλαγή του τύπου κάθε παραμέτρου χρησιμοποιώντας μετατροπές τύπων (type casts). Όσες παράμετροι δεν υπάρχουν στη μέθοδο του αποτελέσματος προστίθενται, ενώ οι πλεονάζουσες παράμετροι αφαιρούνται από τη λίστα των παραμέτρων και τοποθετούνται μέσα στο σώμα της μεθόδου ως δηλώσεις μεταβλητών που αρχικοποιούνται στις προεπιλεγμένες τιμές όπως ορίζονται από τα πρότυπα της Java<sup>11</sup>. Σημειώστε επίσης ότι η σειρά των παραμέτρων ενημερώνεται για να αντιστοιχεί με τη σειρά των παραμέτρων στη μέθοδο τους ερωτήματος. Οι παραπάνω αλλαγές εφαρμόζονται επίσης σε όλες τις εντολές που κάνουν κλήση της μεθόδου.

<sup>10</sup>[http://imagejdocu.tudor.lu/doku.php?id=howto:java:how\\_to\\_convert\\_data\\_type\\_x\\_into\\_type\\_y\\_in\\_java](http://imagejdocu.tudor.lu/doku.php?id=howto:java:how_to_convert_data_type_x_into_type_y_in_java)

<sup>11</sup><https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Τέλος, η νέα έκδοση του τμήματος κώδικα εξάγεται από το μετασχηματισμένο δένδρο XML χρησιμοποιώντας το srcML. Η μορφοποίηση του κώδικα επίσης βελτιώνεται χρησιμοποιώντας τον Artistic Style formatter<sup>12</sup> ώστε να παραχθεί τελικώς ένα τμήμα που είναι έτοιμο για χρήση από τον προγραμματιστή.

Στο Σχήμα 6.11 παρέχεται ένα παράδειγμα υπογραφής ερωτήματος για μια κλάση “Customer” με δύο μεθόδους “setAddress” και “getAddress”, που χρησιμοποιούνται για να θέσουν και να ανακτήσουν τη διεύθυνση του πελάτη, αντίστοιχα. Το παράδειγμα αυτό χρησιμοποιείται σε αυτή την ενότητα για να δείξουμε τη λειτουργικότητα του Transformer.

---

```
public class Customer {
    public void setAddress(String i){}
    public String getAddress(){}
}
```

---

Σχήμα 6.11: Παράδειγμα ερωτήματος για μια κλάση “Customer” με μεθόδους “setAddress” και “getAddress”

Το Σχήμα 6.12 απεικονίζει ένα αποτέλεσμα για την κλάση “Customer” που περιέχει τις απαιτούμενες μεθόδους με ελαφρώς διαφορετικά ονόματα, διαφορετικές παραμέτρους και διαφορετικούς τύπους επιστροφής.

---

```
public class Customer {
    private String address;
    public Customer() {
    }

    public boolean setTheAddress(String address, String postCode) {
        this.address = address;
        if (postCode != null) {
            this.address += ", " + postCode;
            return true;
        }
        return false;
    }

    public String getTheAddress() {
        return address;
    }
}
```

---

Σχήμα 6.12: Παράδειγμα αποτελέσματος για την κλάση “Customer”

<sup>12</sup><http://astyle.sourceforge.net/>

Ο Transformer αρχικά μετονομάζει τις μεθόδους `setTheAddress` και `getTheAddress` σε `setAddress` και `getAddress` αντίστοιχα. Στη συνέχεια, ο τύπος επιστροφής της `setAddress` μετατρέπεται από `int` σε `void` και οι αντίστοιχοι τύποι επιστροφής τροποποιούνται. Τέλος, η πλεονάζουσα παράμετρος `postCode` μεταφέρεται μέσα στη μέθοδο και τίθεται στην προκαθορισμένη τιμή της (στην περίπτωση αυτή `null`). Το τελικό αποτέλεσμα φαίνεται στο Σχήμα 6.13.

---

```
public class Customer {
    private String address;
    public Customer() {
    }

    public void setAddress(String address) {
        String postCode = null;
        this.address = address;
        if (postCode != null) {
            this.address += ", " + postCode;
            return;
        }
        return;
    }

    public String getAddress() {
        return address;
    }
}
```

---

Σχήμα 6.13: Παράδειγμα μετασχηματισμού για το αρχείο του Σχήματος 6.12

### 6.3.6 Tester

Ο Tester του Mantissa ελέγχει εάν τα αποτελέσματα ικανοποιούν την επιθυμητή λειτουργικότητα που έχει ορίσει ο προγραμματιστής. Για να λειτουργήσει αυτό το τμήμα, ο προγραμματιστής πρέπει να παράσχει περιπτώσεις ελέγχου (test case) για το τμήμα λογισμικού του ερωτήματος. Αρχικά, για κάθε αρχείο-αποτέλεσμα, δημιουργείται ένας φάκελος. Το αρχείο και η περίπτωση ελέγχου που έχει γράψει ο προγραμματιστής αντιγράφονται στον φάκελο. Στη συνέχεια, ο Tester χρησιμοποιεί κανονικές εκφράσεις για να τροποποιήσει το αρχείο πηγαίου κώδικα και το αρχείο ελέγχου, έτσι ώστε να είναι δυνατή η εκτέλεση των περιπτώσεων ελέγχου. Τα αρχεία μεταγλωττίζονται (αν αυτό είναι δυνατό) και ο έλεγχος εκτελείται. Το αποτέλεσμα αυτής της διαδικασίας υποδεικνύει αν το αρχείο που έχει ληφθεί είναι μεταγλωττίσιμο και αν περνάει τον έλεγχο. Το αρχείο ελέγχου που παρέχεται από τον προγραμματιστή για το τμήμα του Σχήματος 6.11 φαίνεται στο Σχήμα 6.14.

---

```

import static org.junit.Assert.assertEquals;

import org.junit.Test;
import java.lang.reflect.*;

public class CustomerTest {

    @Test
    public void testAddress() throws Exception {
        int m_index = 0;
        Class<?> clazz = Class.forName("Customer");
        Object c = clazz.newInstance();
        Method m = clazz.getDeclaredMethod("setAddress", String.class);
        m.invoke(c,"test");
        m_index = 1;
        m = clazz.getDeclaredMethod("getAddress");
        assertEquals("Wrong string!", "test", m.invoke(c));
    }
}

```

---

Σχήμα 6.14: Παράδειγμα περίπτωσης ελέγχου για την κλάση “Customer” του Σχήματος 6.11

Ο έλεγχος δημιουργείται χρησιμοποιώντας το εργαλείο ελέγχου JUnit<sup>13</sup> και τη δυνατότητα *reflection* της Java. Το *reflection* χρειάζεται για την παροχή μιας δομής ελέγχου που μπορεί να τροποποιηθεί ώστε να συμφωνεί με διαφορετικά αρχεία. Στον κώδικα του Σχήματος 6.14, η μεταβλητή `m_index` είναι ένας μετρητής που αποθηκεύει τη θέση της μεθόδου που θα κληθεί στη συνέχεια από το αρχείο αποτελέσματος. Χρησιμοποιώντας το μετρητή αποφεύγονται προβλήματα με μεθόδους που γίνονται *overload*<sup>14</sup>. Ένα αντικείμενο κλάσης δημιουργείται με την ανάκτηση ενός *constructor* χρησιμοποιώντας τη συνάρτηση `getDeclaredConstructor` και καλώντας τη συνάρτηση `newInstance`. Στη συνέχεια, οι μέθοδοι ανακτώνται χρησιμοποιώντας τη συνάρτηση `getDeclaredMethod` και καλούνται με τη συνάρτηση `invoke`.

Πριν από τη μεταγλώττιση ενός νέου αρχείου μαζί με την περίπτωση ελέγχου, ο *Tester* πρέπει να τροποποιήσει τα αρχεία για να είναι συμβατά. Η τροποποίηση των αποτελεσμάτων και των περιπτώσεων ελέγχου περιλαμβάνει τη μετονομασία των πακέτων και τη μετατροπή όλων των μεθόδων από ιδιωτικές/προστατευμένες (*private/protected*) σε δημόσιες (*public*) προκειμένου να είναι δυνατή η μεταγλώττιση του ανακτημένου αρχείου μαζί με το αρχείο ελέγχου. Δεν χρειάζονται περαιτέρω μετασχηματισμοί από τον *Tester*, αφού ο *Transformer* έχει ήδη εκτελέσει τις απαραίτητες μετατροπές σε επίπεδο κλάσης, μεθόδου και παραμέτρων.

---

<sup>13</sup><https://junit.org/>

<sup>14</sup>Η Java υποστηρίζει *overload* μεθόδων, που σημαίνει ότι επιτρέπει την ύπαρξη μεθόδων με το ίδιο όνομα που διαφέρουν μόνο στις παραμέτρους τους.

Το επόμενο βήμα είναι η μεταγλώττιση των αρχείων. Χρησιμοποιείται ο μεταγλωττιστής του Eclipse<sup>15</sup>, καθώς επιτρέπει τη μεταγλώττιση πολλών αρχείων, χωρίς να σταματάει σε περίπτωση σφαλμάτων σε ένα μόνο αρχείο. Τέλος, για τα αρχεία που μεταγλωττίζονται επιτυχώς, εκτελούνται οι περιπτώσεις ελέγχου. Τα τελικά αποτελέσματα, συμπεριλαμβανομένης της κατάστασής τους, εάν δηλαδή μεταγλωττίστηκαν και εάν πέρασαν επιτυχώς τον έλεγχο, παρέχονται στον προγραμματιστή.

### 6.3.7 Flow Extractor

Ο Flow Extractor εξάγει τη ροή του πηγαίου κώδικα των μεθόδων για κάθε αποτέλεσμα. Οι τύποι που χρησιμοποιούνται σε κάθε μέθοδο και η ακολουθία των εντολών μπορεί να αποτελέσουν χρήσιμη πληροφορία για τον προγραμματιστή, καθώς μπορεί εύκολα να κατανοήσει την εσωτερική λειτουργία της μεθόδου χωρίς να χρειάζεται να διαβάσει τον πηγαίο κώδικα. Χρησιμοποιούμε τον εξαγωγέα ακολουθιών εντολών που προτείνεται στο [213] και τον επεκτείνουμε περαιτέρω για να υπολογίσουμε τις συνθήκες (conditions), τους βρόχους (loops) και τις δηλώσεις try-catch.

Συγκεκριμένα, για κάθε τμήμα, εξάγουμε αρχικά όλες τις δηλώσεις (declarations) από τον πηγαίο κώδικα (συμπεριλαμβανομένων των κλάσεων, πεδίων, μεθόδων και μεταβλητών) για να δημιουργήσουμε έναν ιεραρχικό πίνακα αναζήτησης (hierarchical lookup table) σύμφωνα με τα scopes, όπως αυτά ορίζονται στη γλώσσα Java. Στη συνέχεια, διατρέχουμε τις εντολές κάθε μεθόδου και εξάγουμε τρεις τύπους εντολών: αναθέσεις (assignments), κλήσεις συναρτήσεων (functions calls) και δημιουργίες αντικειμένων (class instantiations). Για παράδειγμα, η εντολή `this.address = address` του Σχήματος 6.13 είναι μια εντολή ανάθεσης τύπου String. Χρησιμοποιούνται όλες οι διαφορετικές εναλλακτικές διαδρομές. Εκτός από τις συνθήκες, διαφορετικές ροές ορίζονται και από τις δηλώσεις try-catch (που μπορεί να έχουν και τμήματα finally που εκτελούνται σε όλες τις περιπτώσεις), ενώ οι βρόχοι (loops) είτε εκτελούνται είτε όχι, δηλαδή θεωρούνται ως συνθήκες.

Τέλος, για κάθε μέθοδο, οι πιθανές ροές της απεικονίζονται με τη μορφή γράφου, όπου κάθε κόμβος αντιπροσωπεύει μια εντολή. Οι γράφοι κατασκευάζονται σε μορφή Graphviz dot<sup>16</sup>, και οπτικοποιούνται με τη βιβλιοθήκη Viz.js<sup>17</sup>. Επιπλέον, οι τύποι των κόμβων εμφανίζονται με κόκκινη γραμματοσειρά όταν αναφέρονται σε εξωτερικές εξαρτήσεις. Ένα παράδειγμα χρήσης του Flow Extractor θα δοθεί στην ενότητα 6.4.2.

## 6.4 Περιβάλλον Διεπαφής και Σενάριο Αναζήτησης

### 6.4.1 Περιβάλλον Διεπαφής του Mantissa

Το περιβάλλον διεπαφής του Mantissa είναι μια εφαρμογή πελάτη-διακομιστή (client-server). Ο διακομιστής υλοποιείται σε Python χρησιμοποιώντας το Flask<sup>18</sup>, ενώ HTML και Javascript έχουν χρησιμοποιηθεί για την ανάπτυξη της διεπαφής χρήστη της υπηρεσίας. Η αρχική σελίδα του Mantissa φαίνεται στο Σχήμα 6.15.

<sup>15</sup><http://www.eclipse.org/jdt/core/>

<sup>16</sup><http://www.graphviz.org/>

<sup>17</sup><http://viz-js.com/>

<sup>18</sup><http://flask.pocoo.org/>

The screenshot shows the Mantissa search interface. At the top, there is a dark blue header with the 'Mantissa' logo on the left and a hamburger menu icon on the right. Below the header, the word 'Code' is centered. Underneath, a light gray box contains the following Java code snippet:

```
public class Stack{
    public void push(Object o){}
    public Object pop(){}
```

Below the code box, the word 'Test' is centered. Underneath, another light gray box contains the following Java code snippet:

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;
import java.lang.Object;
import java.lang.reflect.*;

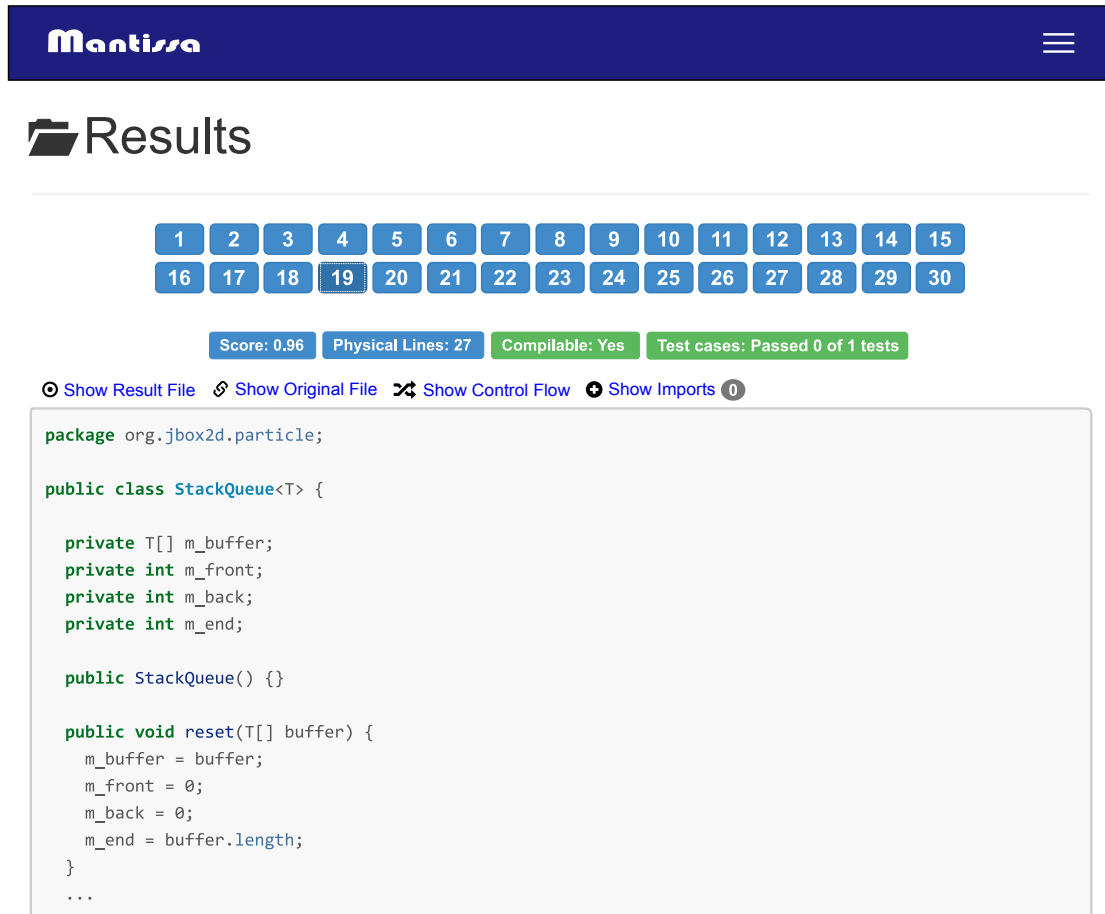
public class StackTest{
    ...
```

At the bottom of the interface, there is a 'Source' section with a dropdown menu currently set to 'AGORA' and a blue 'Search' button to its right.

Σχήμα 6.15: Σελίδα αναζήτησης του Mantissa

Παρέχονται δύο περιοχές κειμένου στο κέντρο της οθόνης που επιτρέπουν στο χρήστη να εισάγει ένα ερώτημα σε μορφή διεπαφής (interface) και τον κώδικα ελέγχου που θα χρησιμοποιηθεί για την αξιολόγηση των ανακτημένων τμημάτων κώδικα. Υπάρχει επίσης η επιλογή να μην δοθεί αρχείο ελέγχου, το ενεργοποιώντας το checkbox “No test case”. Τέλος, στο κάτω μέρος της οθόνης, ο χρήστης μπορεί να επιλέξει τη CSE που θα χρησιμοποιηθεί για αναζήτηση, είτε AGORA είτε GitHub, χρησιμοποιώντας ένα μενού dropdown, πριν τελικά πατήσει το κουμπί “Search”. Στο παράδειγμα του Σχήματος 6.15, ο χρήστης έχει εισάγει ένα ερώτημα για ένα τμήμα “Stack” και την αντίστοιχη περίπτωση ελέγχου.

Με την εκτέλεση μιας αναζήτησης, παρουσιάζονται στο χρήστη τα σχετικά αποτελέσματα στη σελίδα αποτελεσμάτων του Σχήματος 6.16. Όπως φαίνεται σε αυτό το Σχήμα, ο χρήστης μπορεί να πλοηγηθεί στα αποτελέσματα καθώς και να ενσωματώσει ενδεχομένως κάποιο χρήσιμο τμήμα κώδικα στον κώδικα του. Κάθε αποτέλεσμα συνοδεύεται από πληροφορίες σχετικά με τη βαθμολογία που δίνεται από το σύστημα, τις γραμμές κώδικα, καθώς και αν μεταγλωττίστηκε και αν πέρασε με επιτυχία τον έλεγχο. Για παράδειγμα, το Σχήμα 6.16 απεικονίζει το 19ο αποτέλεσμα για το ερώτημα “Stack”, το οποίο έχει βαθμολογία ίση με 0.96, αποτελείται από 27 γραμμές κώδικα, μεταγλωττίστηκε αλλά δεν πέρασε τον έλεγχο. Τέλος, οι διάφοροι σύνδεσμοι επιτρέπουν την προβολή του αρχικού αρχείου (δηλαδή πριν από οποιονδήποτε μετασχηματισμό από τον Transformer), την προβολή της ροής ελέγχου για κάθε μέθοδο του αρχείου, καθώς και την προβολή των εξωτερικών εξαρτήσεων με τη μορφή δηλώσεων βιβλιοθηκών (imports).



The screenshot shows the Mantis web interface. At the top, there is a dark blue header with the 'Mantissa' logo and a hamburger menu icon. Below the header, the word 'Results' is displayed with a folder icon. A navigation bar contains 30 numbered buttons, with button 19 highlighted. Below the navigation bar, there are four status indicators: 'Score: 0.96', 'Physical Lines: 27', 'Compilable: Yes', and 'Test cases: Passed 0 of 1 tests'. Below these indicators are four links: 'Show Result File', 'Show Original File', 'Show Control Flow', and 'Show Imports'. The main content area displays a code snippet for a Java class named 'StackQueue'.

```

package org.jbox2d.particle;

public class StackQueue<T> {

    private T[] m_buffer;
    private int m_front;
    private int m_back;
    private int m_end;

    public StackQueue() {}

    public void reset(T[] buffer) {
        m_buffer = buffer;
        m_front = 0;
        m_back = 0;
        m_end = buffer.length;
    }
    ...

```

Σχήμα 6.16: Σελίδα αποτελεσμάτων του Mantissa

## 6.4.2 Παράδειγμα Σεναρίου Αναζήτησης

Σε αυτή την ενότητα παρέχουμε ένα παράδειγμα σεναρίου χρήσης για να δείξουμε πώς το Mantissa μπορεί να βοηθήσει τον προγραμματιστή. Το σενάριο περιλαμβάνει τη δημιουργία μιας εφαρμογής ελέγχου διπλότυπων αρχείων (duplicate file checker). Η εφαρμογή αρχικά θα διατρέξει τα αρχεία ενός συγκεκριμένου φακέλου και θα αποθηκεύσει την αναπαράσταση MD5 για τα περιεχόμενα κάθε αρχείου σε ένα σύνολο (set). Στη συνέχεια, κάθε νέο αρχείο θα ελέγχεται χρησιμοποιώντας το σύνολο προτού προστεθεί στο φάκελο.

Το πρώτο βήμα είναι ο διαχωρισμός της εφαρμογής σε διαφορετικά τμήματα. Υποθέτουμε ότι ο προγραμματιστής θα χώριζε την εφαρμογή σε ένα τμήμα ανάγνωσης/γραφής αρχείων (file reader/writer), μια δομή δεδομένων τύπου σύνολο (set data structure) και μια υλοποίηση του αλγορίθμου MD5. Για το τμήμα ανάγνωσης/γραφής αρχείων, κατασκευάζουμε το ερώτημα που φαίνεται στο Σχήμα 6.17.

---

```

public class FileTools {
    public String read(String filename){}
    public void write(String filename, String content){}
}

```

---

Σχήμα 6.17: Παράδειγμα υπογραφής για ένα τμήμα που διαβάζει και γράφει αρχεία

Μια ματιά στη λίστα των αποτελεσμάτων αποκαλύπτει ότι πολλά τμήματα παρέχουν τη ζητούμενη λειτουργικότητα. Σε αυτή την περίπτωση, η επιλογή μας θα είναι το μικρότερο από όλα τα τμήματα που είναι όμως ταυτόχρονα μεταγλωττίσιμο. Ενδεικτικά, η μέθοδος εγγραφής αρχείου αυτού του τμήματος φαίνεται στο Σχήμα 6.18.

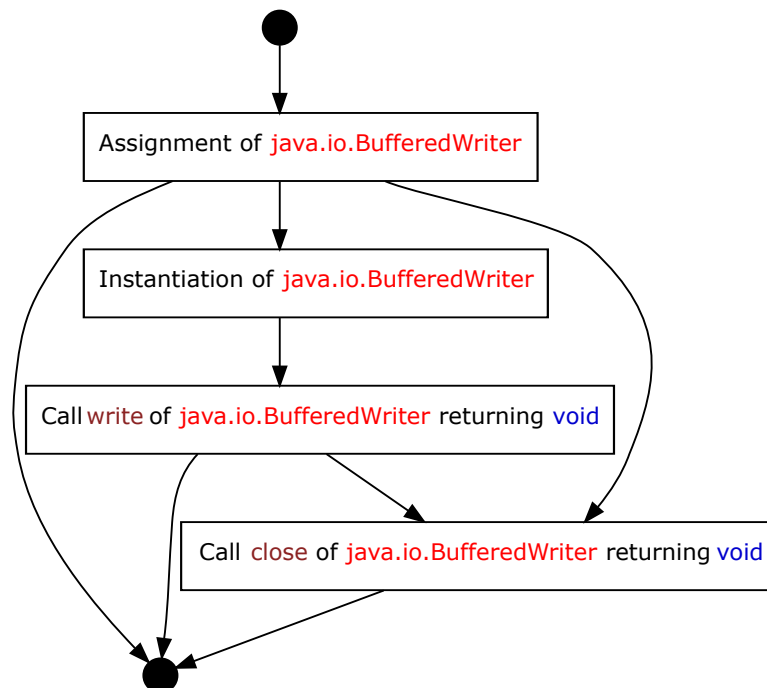
---

```
public static void write(String content, String filePath) throws IOException {
    BufferedWriter out = null;
    try {
        out = new BufferedWriter(new FileWriter(filePath));
        out.write(content);
    } finally {
        if (out != null)
            out.close();
    }
}
```

---

Σχήμα 6.18: Παράδειγμα μεθόδου γραφής σε αρχείο για ένα τμήμα που διαχειρίζεται αρχεία

Εξετάζοντας τη ροή του κώδικα των μεθόδων για το επιλεγμένο αποτέλεσμα, παρατηρούμε ότι χρησιμοποιεί μεθόδους της βασικής βιβλιοθήκης της Java, επομένως ο έλεγχος σε αυτήν την περίπτωση μπορεί να παραλειφθεί για λόγους απλότητας. Για παράδειγμα, στο Σχήμα 6.19 απεικονίζεται η ροή κώδικα για τη μέθοδο του Σχήματος 6.18, όπου είναι σαφές ότι ο βασικός τύπος αντικειμένου που εμπλέκεται είναι ο `BufferedWriter`.



Σχήμα 6.19: Ροή κώδικα για μια μέθοδο γραφής σε αρχείο ενός τμήματος που διαχειρίζεται αρχεία



Το ερώτημα για τον αλγόριθμο MD5 φαίνεται στο Σχήμα 6.20, ενώ η αντίστοιχη περίπτωση ελέγχου φαίνεται στο Σχήμα 6.21. Ελέγχεται αν ο κατακερματισμός (hashing) μιας γνωστής συμβολοσειράς είναι σωστός. Στην περίπτωση αυτή, το δεύτερο αποτέλεσμα περνάει τον έλεγχο και δεν έχει εξωτερικές εξαρτήσεις.

---

```
public class Md5 {
    public String md5Sum(String i){}
}
```

---

Σχήμα 6.20: Παράδειγμα υπογραφής για ένα τμήμα που υπολογίζει το MD5 hash μιας συμβολοσειράς

---

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import java.lang.reflect.*;

public class Md5Test {

    @Test
    public void testMd5Sum() throws Exception {
        int m_index = 0;
        Class<?> c = Class.forName("Md5");
        Method m = c.getDeclaredMethod("md5Sum", String.class);
        assertEquals("c2a9ce57e8df081b4baad80d81868bbb",
            m.invoke(null, "This is my string"));
    }
}
```

---

Σχήμα 6.21: Παράδειγμα περίπτωσης ελέγχου για το τμήμα “MD5” του Σχήματος 6.20

Το επόμενο τμήμα είναι η δομή δεδομένων συνόλου. Το ερώτημα φαίνεται στο Σχήμα 6.22 και η αντίστοιχη περίπτωση ελέγχου φαίνεται στο Σχήμα 6.23. Σημειώστε ότι το τμήμα πρέπει επίσης να είναι serializable, κάτι που ελέγχεται κι από την περίπτωση ελέγχου.

---

```
public class SerializableSet {
    public String serialize(){}
    public SerializableSet deserialize(String s){}
}
```

---

Σχήμα 6.22: Παράδειγμα υπογραφής για μια serializable δομή συνόλου

---

```

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import java.lang.reflect.*;

public class SerializableSetTest {

    @Test
    public void testSerialization() throws Exception {
        int m_index = 0;
        Class<?> clazz = Class.forName("SerializableSet");
        Object c = clazz.newInstance();
        Method m = clazz.getDeclaredMethod("serialize");
        Object serializedSet = m.invoke(c);
        m_index = 1;
        m = clazz.getDeclaredMethod("deserialize", String.class);
        Object deserializedSet = m.invoke(c, serializedSet);
        assertEquals(c, deserializedSet);
    }
}

```

---

Σχήμα 6.23: Παράδειγμα περίπτωσης ελέγχου για τη serializable δομή συνόλου του Σχήματος 6.22

Τέλος, μετά το κατέβασμα των τμημάτων, ο προγραμματιστής μπορεί πλέον εύκολα να γράψει τον κώδικα που απομένει για τη δημιουργία του συνόλου με τις τιμές MD5, την ανάγνωση ή γραφή του συνόλου αυτού στο δίσκο, καθώς και τον έλεγχο αν κάποιο νέο αρχείο αποτελεί διπλότυπο (έχει δηλαδή ίδιες πληροφορίες). Ο κώδικας για τη δημιουργία των παραπάνω φαίνεται στο Σχήμα 6.24.

---

```

SerializableSet set = new SerializableSet();
for (File string : new File(folderPath).listFiles()) {
    String content = FileTools.read(string.getAbsolutePath());
    String md5String = Md5.md5Sum(content);
    set.add(md5String);
}
FileTools.write(set.serialize(), setFilename);

```

---

Σχήμα 6.24: Παράδειγμα για τη δημιουργία ενός συνόλου από συμβολοσειρές MD5 για τα αρχεία του φακέλου folderPath και αποθήκευση του συνόλου σε ένα αρχείο με όνομα setFilename

Συνολικά, ο προγραμματιστής χρειάζεται να γράψει λιγότερες από 50 γραμμές κώδικα για να κατασκευάσει τα ερωτήματα και τους ελέγχους για αυτήν την εφαρμογή. Μετά την

έγρευση εκτελέσιμων τμημάτων που καλύπτουν την απαιτούμενη λειτουργικότητα, η ενσωμάτωση στον κώδικα είναι επίσης σχετικά εύκολη. Τέλος, η βασική ροή της εφαρμογής (ένα μέρος της οποίας φαίνεται στο Σχήμα 6.24) είναι αρκετά απλή, καθώς όλες οι λεπτομέρειες της υλοποίησης των τμημάτων βρίσκονται εντός τους (είναι abstracted). Μέσα από το παραπάνω σενάριο γίνεται σαφές ότι ένα μοντέλο ανάπτυξης βασισμένο σε επαναχρησιμοποίηση τμημάτων μπορεί να έχει ως αποτέλεσμα μειωμένο χρόνο και προσπάθεια για τον προγραμματιστή και συνολικά καλύτερη σχεδίαση. Επιπλέον, χρησιμοποιώντας τις δυνατότητες του Mantissa και με τους απαραίτητους ελέγχους, ο προγραμματιστής μπορεί να είναι βέβαιος ότι τα τμήματα που έχει επιλέξει καλύπτουν την απαιτούμενη λειτουργικότητα.

## 6.5 Αξιολόγηση

Αξιολογούμε το Mantissa ενάντια σε δημοφιλείς CSEs (αυτά που αξιολογήθηκαν και στο προηγούμενο κεφάλαιο, δηλαδή το GitHub Search, το BlackDuck Open Hub και την AGORA), καθώς επίσης και σε σχέση με δύο γνωστά RSSEs (FAST και Code Conjurer). Για κάθε ένα από τα πειράματα παρουσιάζουμε το σύνολο δεδομένων που χρησιμοποιείται για την αξιολόγηση, ενώ τα αποτελέσματα συνοψίζονται σε πίνακες και απεικονίζονται στα σχήματα. Σημειώστε ότι κάθε πείραμα έχει εκτελεστεί με διαφορετικό σύνολο δεδομένων, καθώς ορισμένα από τα συστήματα δεν είναι πλέον διαθέσιμα, έτσι μόνο τα δικά τους σύνολα δεδομένων μπορούσαν να χρησιμοποιηθούν για κοινή αξιολόγηση.

### 6.5.1 Μηχανισμός Αξιολόγησης

Η αξιολόγησή μας βασίζεται στο σενάριο αναζήτησης ενός επαναχρησιμοποιήσιμου τμήματος λογισμικού, όπως περιγράφηκε στην αξιολόγηση του προηγούμενου κεφαλαίου. Στο σενάριό μας, ο προγραμματιστής δημιουργεί ένα ερώτημα για κάποιο τμήμα λογισμικού και το στέλνει σε κάποια CSE ή κάποιο RSSE. Το σύστημα επιστρέφει σχετικά αποτελέσματα και ο προγραμματιστής τα εξετάζει και προσδιορίζει αν είναι σχετικά με το ερώτημα που υπέβαλε. Όπως και προηγουμένως, αποφεύγουμε την εξέταση των αποτελεσμάτων, καθώς θα είχαμε ζητήματα με την εγκυρότητα της μεθοδολογίας αξιολόγησης. Αντίθετα, προσδιορίζουμε αν ένα αποτέλεσμα είναι σχετικό με τη χρήση αυτοματοποιημένων ελέγχων (όπως στον Tester του Mantissa, τον οποίο και δεν αξιολογούμε). Τα σύνολα δεδομένων που χρησιμοποιούνται για την αξιολόγηση αποτελούνται από ερωτήματα για τα τμήματα κώδικα και τις αντίστοιχες περιπτώσεις ελέγχου τους.

Για κάθε ερώτημα, χρησιμοποιούμε ως μετρικές το πλήθος των αποτελεσμάτων που μεταγλωττίστηκαν επιτυχώς και το πλήθος των αποτελεσμάτων που πέρασαν τους ελέγχους. Επιπλέον, όπως και στο προηγούμενο κεφάλαιο, χρησιμοποιούμε ως μετρική το μήκος αναζήτησης (search length), το οποίο ορίζουμε ως τον αριθμό των αποτελεσμάτων που δε μεταγλωττίστηκαν (ή που δεν πέρασαν τους ελέγχους) που πρέπει να εξετάσει ο προγραμματιστής ώστε να βρει έναν αριθμό μεταγλωττισμένων αποτελεσμάτων (ή αποτελεσμάτων που πέρασαν τους ελέγχους). Υπολογίζουμε τις τιμές των μετρικών για τα πρώτα 30 αποτελέσματα κάθε συστήματος, καθώς η διατήρηση περισσότερων αποτελεσμάτων δεν είχε καμία επίδραση στη βαθμολογία είτε των μεταγλωττίσιμων αποτελεσμάτων είτε των αποτελεσμάτων που περνάνε τους ελέγχους. Άλλωστε, 30 αποτελέσματα είναι συνήθως περισσότερα από όσα θα εξέταζε ένας προγραμματιστής σε ένα τέτοιο σενάριο.

## 6.5.2 Σύγκριση του Mantissa με CSEs

Στην ενότητα αυτή, επεκτείνουμε την αξιολόγηση που παρουσιάστηκε στο προηγούμενο κεφάλαιο για να δείξουμε πως το Mantissa είναι σε θέση να επιστρέψει καλύτερα αποτελέσματα σε σχέση με τη χρήση μόνο μιας CSE. Έτσι, συγκρίνουμε το Mantissa με τις γνωστές CSEs του GitHub και του BlackDuck, καθώς και με τη δική μας CSE, την AGORA.

### 6.5.2.1 Σύνολο Δεδομένων

Το σύνολο δεδομένων μας αποτελείται από 16 τμήματα κώδικα και τις αντίστοιχες περιπτώσεις ελέγχου. Τα τμήματα αυτά έχουν διαφορετική πολυπλοκότητα ώστε να αξιολογηθούν με πληρότητα τα αποτελέσματα των CSEs έναντι αυτών του Mantissa στο σενάριο επαναχρησιμοποίησης τμημάτων. Το σύνολο δεδομένων μας φαίνεται στον Πίνακα 6.3.

Πίνακας 6.3: Σύνολο δεδομένων για την αξιολόγηση του Mantissa ενάντια σε Μηχανές Αναζήτησης Κώδικα

Κλάση	Μέθοδοι
Account	deposit, withdraw, getBalance
Article	setId, getId, setName, getName, setPrice, getPrice
Calculator	add, subtract, divide, multiply
ComplexNumber	ComplexNumber, add, getRealPart, getImaginaryPart
CreditCardValidator	isValid
Customer	setAddress, getAddress
Gcd	gcd
Md5	md5Sum
Mortgage	setRate, setPrincipal, setYears, getMonthlyPayment
Movie	Movie, getTitle
Prime	checkPrime
Sort	sort
Spreadsheet	put, get
Stack	push, pop
Stack2	pushObject, popObject
Tokenizer	tokenize

Τα ερωτήματα για τις τρεις CSEs διαμορφώνονται ακριβώς όπως στο προηγούμενο κεφάλαιο. Ωστόσο, το σύνολο δεδομένων ενημερώνεται για να περιλαμβάνει πλέον 16 τμήματα. Τα προϋπάρχοντα 13 αναφέρονται σε κλάσεις, ενώ τα 3 νέα, “Prime”, “Sort” και “Tokenizer” αναφέρονται σε συναρτήσεις. Αυτά τα ερωτήματα προστέθηκαν για να αξιολογηθούν περαιτέρω τα συστήματα στην περίπτωση που το απαιτούμενο τμήμα είναι μια συνάρτηση αντί για κλάση. Αναλύοντας περαιτέρω το σύνολο δεδομένων, παρατηρούμε ότι υπάρχουν απλά ερωτήματα, μερικά από τα οποία σχετίζονται με τα μαθηματικά, για τα οποία αναμένουμε αρκετά αποτελέσματα (π.χ. “Gcd” ή “Prime”) και πιο σύνθετα ερωτήματα (π.χ. “Mortgage”) ή ερωτήματα με λιγότερο συχνά ονόματα (π.χ. “Spreadsheet”) που είναι πιο δύσκολα για τα συστήματα. Τέλος, αν και ορισμένα ερωτήματα μπορεί να φαίνονται απλά (π.χ. “Calculator”), ενδέχεται να έχουν εξαρτήσεις από βιβλιοθήκες που θα οδηγήσουν σε σφάλματα σύνταξης.

### 6.5.2.2 Αποτελέσματα

Τα αποτελέσματα της σύγκρισης του Mantissa με τις CSEs φαίνονται στον Πίνακα 6.4. Αρχικά, παρατηρούμε ότι το GitHub, η AGORA, καθώς και οι δύο εκδόσεις του Mantissa επέστρεψαν τουλάχιστον 30 αποτελέσματα για κάθε ερώτημα. Αντίθετα, το BlackDuck επέστρεψε λιγότερα αποτελέσματα για τα μισά ερωτήματα, υποδεικνύοντας ότι ο μηχανισμός ανάκτησής του εκτελεί ακριβή αντιστοίχιση (exact matching), συνεπώς είναι πιο ευαίσθητος σε αλλαγές ονομάτων στο ερώτημα. Αυτό είναι ιδιαίτερα προφανές στην τροποποιημένη εκδοχή του ερωτήματος για μια στοίβα (“Stack2”), όπου το BlackDuck δεν επέστρεψε κανένα αποτέλεσμα.

Πίνακας 6.4: Αποτελέσματα που μεταγλωττίστηκαν και αποτελέσματα που πέρασαν τους ελέγχους για τις τρεις CSEs και τις δύο εκδόσεις του Mantissa, για κάθε ερώτημα και κατά μέσο όρο, όπου η μορφή για κάθε τιμή είναι Πλήθος Ελεγμένων Αποτελεσμάτων / Πλήθος Μεταγλωττισμένων Αποτελεσμάτων / Συνολικό Πλήθος Αποτελεσμάτων

Ερώτημα	Μηχανές Αναζήτησης Κώδικα			Mantissa	
	GitHub	AGORA	BlackDuck	AGORA	GitHub
Account	0 / 10 / 30	1 / 4 / 30	2 / 9 / 30	1 / 5 / 30	6 / 7 / 30
Article	1 / 6 / 30	0 / 5 / 30	2 / 3 / 15	5 / 9 / 30	2 / 8 / 30
Calculator	0 / 5 / 30	0 / 5 / 30	1 / 3 / 22	0 / 4 / 30	0 / 9 / 30
ComplexNumber	2 / 2 / 30	2 / 12 / 30	1 / 1 / 3	4 / 7 / 30	7 / 11 / 30
CreditCardValidator	0 / 1 / 30	1 / 4 / 30	2 / 3 / 30	0 / 0 / 30	0 / 4 / 30
Customer	2 / 3 / 30	13 / 15 / 30	7 / 7 / 30	20 / 21 / 30	5 / 6 / 30
Gcd	1 / 3 / 30	5 / 9 / 30	12 / 16 / 30	5 / 9 / 30	20 / 22 / 30
Md5	3 / 7 / 30	2 / 10 / 30	0 / 0 / 0	4 / 7 / 30	1 / 3 / 30
Mortgage	0 / 6 / 30	0 / 4 / 30	0 / 0 / 0	0 / 7 / 30	0 / 1 / 30
Movie	1 / 3 / 30	2 / 2 / 30	3 / 7 / 30	2 / 2 / 30	1 / 5 / 30
Prime	4 / 13 / 30	1 / 4 / 30	1 / 2 / 15	1 / 1 / 30	4 / 14 / 30
Sort	0 / 7 / 30	0 / 1 / 30	0 / 4 / 30	1 / 4 / 30	4 / 5 / 30
Spreadsheet	0 / 1 / 30	1 / 2 / 30	0 / 0 / 6	1 / 2 / 30	1 / 3 / 30
Stack	0 / 3 / 30	7 / 8 / 30	8 / 12 / 30	14 / 15 / 30	10 / 11 / 30
Stack2	0 / 0 / 30	7 / 8 / 30	0 / 0 / 0	14 / 15 / 30	1 / 1 / 30
Tokenizer	0 / 8 / 30	0 / 4 / 30	2 / 7 / 30	0 / 3 / 30	1 / 4 / 30
# Αποτελέσματα	30.0	30.0	18.8	30.0	30.0
# Μεταγλωττισμένα	4.875	6.062	4.625	6.938	7.125
# Ελεγμένα	0.875	2.625	2.562	4.500	3.938

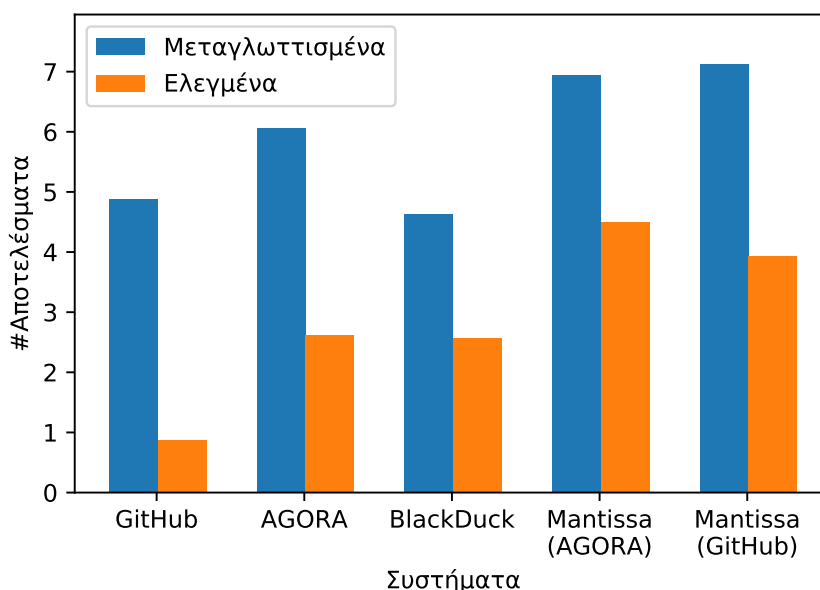
Το GitHub φαίνεται να επιστρέφει αρκετά αποτελέσματα, ωστόσο λίγα από αυτά είναι μεταγλωττίσιμα, ενώ τα περισσότερα ερωτήματα επιστρέφουν αποτελέσματα που δεν περνάνε τους ελέγχους. Αυτό αναμένεται από τη στιγμή που το GitHub δεν είναι σχεδιασμένο για την ανάκτηση επαναχρησιμοποιήσιμων τμημάτων. Η AGORA, από την άλλη πλευρά, παράγει μεταγλωττίσιμα και ελεγμένα αποτελέσματα για τα περισσότερα ερωτήματα. Ανακτά περίπου 6 μεταγλωττίσιμα αποτελέσματα ανά ερώτημα, ενώ είναι επίσης η μόνη CSE που παράγει τουλάχιστον ένα μεταγλωττίσιμο αποτέλεσμα για κάθε ερώτημα, ακόμη και για το τροποποιημένο “Stack”.

Όπως μπορεί κανείς να δει, η σύνδεση του συστήματός μας με την AGORA σχεδόν διπλασίασε τον αριθμό των αποτελεσμάτων που περνάνε τους ελέγχους, ενώ η σύνδεση με

το GitHub τετραπλασίασε την προαναφερθείσα τιμή. Και οι δύο υλοποιήσεις του Mantissa είναι αρκετά αποτελεσματικές, ανακτώντας τουλάχιστον ένα μεταγλωττίσιμο τμήμα κώδικα για σχεδόν όλα τα ερωτήματα και τουλάχιστον ένα που περνάει τους ελέγχους για 13 από τα 16 ερωτήματα.

Αναλύοντας περαιτέρω τα αποτελέσματα που παρουσιάζονται σε αυτόν τον Πίνακα, παρατηρούμε ότι και οι δύο υλοποιήσεις του Mantissa επέστρεψαν πολυάριθμα αποτελέσματα για ερωτήματα μεμονωμένων μεθόδων (π.χ. “Gcd” και “Sort”) ή για σχετικά απλά ερωτήματα (π.χ. “Customer” και “Stack”). Η έκδοση με το GitHub είναι πιο αποτελεσματική στα ερωτήματα μαθηματικού τύπου, όπως τα “Gcd” και “Prime”, πιθανώς λόγω του ότι το ευρετήριο της AGORA είναι αρκετά μικρότερο από αυτό του GitHub. Τα μαθηματικά τμήματα κώδικα μπορεί επίσης να έχουν εξαρτήσεις από βιβλιοθήκες που οδηγούν σε σφάλματα σύνταξης, όπως π.χ. συμβαίνει και στις δύο CSEs για το ερώτημα “Calculator”. Σημειώστε, ωστόσο, ότι το Mantissa παρέχει συνδέσμους στο gpercode για τις εξαρτήσεις, που θα μπορούσαν να είναι χρήσιμοι για τέτοιου είδους ερωτήματα. Η υλοποίηση του Mantissa που συνδέεται με την AGORA είναι πιο αποτελεσματική για δύσκολα ερωτήματα, όπως το ερώτημα “Stack2”, κάτι που οφείλεται στην αναλυτική δυνατότητα της AGORA που βασίζεται στη σύνταξη του πηγαίου κώδικα.

Τέλος, ο μέσος αριθμός των μεταγλωττισμένων αποτελεσμάτων και ο μέσος αριθμός των αποτελεσμάτων που πέρασαν τους ελέγχους (ελεγμένων αποτελεσμάτων) για κάθε ένα από τα πέντε συστήματα απεικονίζονται στο Σχήμα 6.25. Με βάση αυτό το Σχήμα, είναι επίσης σαφές ότι οι υλοποιήσεις του Mantissa είναι πιο αποτελεσματικές από τις τρεις CSEs.



Σχήμα 6.25: Διάγραμμα που απεικονίζει το μέσο πλήθος αποτελεσμάτων που μεταγλωττίστηκαν και αυτών που πέρασαν τους ελέγχους για τις τρεις Μηχανές Αναζήτησης Κώδικα και τις δύο υλοποιήσεις του Mantissa

Εκτός από την απόδοση των πέντε συστημάτων για την ανάκτηση χρήσιμων αποτελεσμάτων, είναι επίσης σημαντικό να καθοριστεί εάν αυτά τα αποτελέσματα κατατάσσονται βέλτιστα. Δεδομένου ότι ένα ελεγμένο αποτέλεσμα είναι ίσως το πιο χρήσιμο για τον προγραμματιστή, το μήκος αναζήτησης υπολογίζεται για την εύρεση του πρώτου αποτελέσμα-

τος που πέρασε τους ελέγχους. Τα αποτελέσματα για το μήκος αναζήτησης εμφανίζονται στον Πίνακα 6.5 για τα πέντε συστήματα.

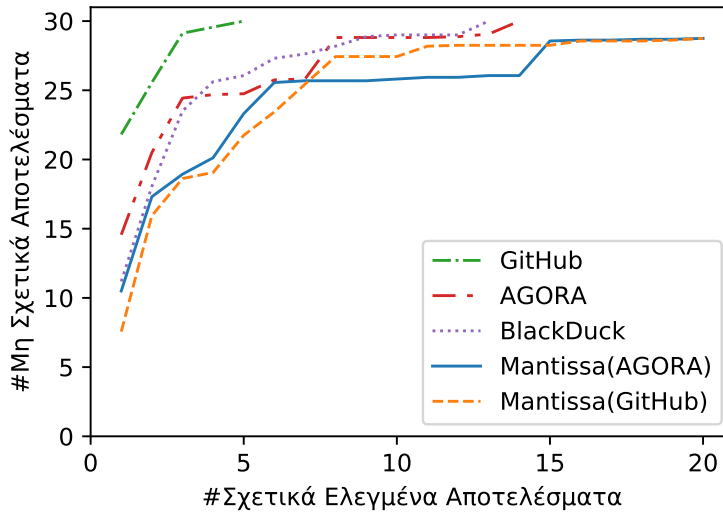
Πίνακας 6.5: Μήκη Αναζήτησης για τα Ελεγμένα Αποτελέσματα, για τις τρεις Μηχανές Αναζήτησης Κώδικα και τις δύο υλοποιήσεις του Mantissa, για κάθε Αποτέλεσμα και ως Μέσο Όροι

Ερώτημα	Μηχανές Αναζήτησης Κώδικα			Mantissa	
	GitHub	AGORA	BlackDuck	AGORA	GitHub
Account	30	2	0	0	0
Article	21	30	2	11	1
Calculator	30	30	0	30	30
ComplexNumber	0	0	0	0	0
CreditCardValidator	30	15	2	30	30
Customer	0	0	1	0	0
Gcd	25	8	5	0	0
Md5	9	0	30	1	2
Mortgage	30	30	30	30	30
Movie	23	21	1	22	7
Prime	1	10	0	6	0
Sort	30	30	30	0	1
Spreadsheet	30	23	30	8	9
Stack	30	2	4	0	3
Stack2	30	2	30	0	0
Tokenizer	30	30	14	30	8
Μέσο Μήκος Αναζήτησης Ελεγμένων	21.812	14.562	11.188	10.500	7.562

Και οι δύο υλοποιήσεις του Mantissa τοποθέτησαν ένα επιτυχώς ελεγμένο αποτέλεσμα στην πρώτη θέση για τα μισά σχεδόν ερωτήματα του συνόλου δεδομένων. Ακόμα και σε ερωτήματα όπου μόνο ένα αποτέλεσμα πέρασε του ελέγχους, όπως τα “Prime” και “Sort”, αυτό κατατάχθηκε σε υψηλή θέση. Συνεπώς, και οι δύο υλοποιήσεις είναι πιο αποτελεσματικές από οποιαδήποτε CSE. Όσον αφορά τα τρία CSEs, το GitHub είναι το πιο αδύναμο από τα τρία συστήματα, ενώ η AGORA και το BlackDuck φαίνεται να παρέχουν αποτελεσματική κατάταξη για τα μισά περίπου ερωτήματα.

Το μέσο μήκος αναζήτησης για την εύρεση περισσότερων του ενός ελεγμένων αποτελεσμάτων εμφανίζεται στο Σχήμα 6.26. Όπως φαίνεται σε αυτό το Σχήμα, το BlackDuck είναι αποτελεσματικό για την εύρεση του πρώτου σχετικού αποτελέσματος, ενώ το AGORA είναι καλύτερο όταν απαιτούνται περισσότερα αποτελέσματα. Όλα τα συστήματα υπερέχουν του GitHub σε αυτήν την περίπτωση. Συγκρίνοντας όλα τα συστήματα, είναι σαφές ότι χρησιμοποιώντας το Mantissa απαιτείται η εξέταση λιγότερων άσχετων αποτελεσμάτων για να βρεθούν τα πρώτα σχετικά. Αν, για παράδειγμα, ένας προγραμματιστής θέλει να βρει τα πρώτα 3 αποτελέσματα που πέρασαν τους ελέγχους, τότε κατά μέσο όρο θα πρέπει να εξετάσει 18.6 αποτελέσματα από αυτά που επιστρέφει το Mantissa με CSE το GitHub και 18.9 αποτελέσματα από αυτά που επιστρέφει το Mantissa με CSE την AGORA. Αντίστοιχα, θα έπρεπε να εξετάσει 23.5 αποτελέσματα του BlackDuck, 24.4 αποτελέσματα της AGORA και 29.13 αποτελέσματα του GitHub. Η υλοποίηση του Mantissa που συνδέεται με το GitHub

υπερέχει όλων των άλλων συστημάτων για την εύρεση των πρώτων αποτελεσμάτων, ενώ η υλοποίηση που συνδέεται με την AGORA είναι πιο αποτελεσματική για την εύρεση περισσότερων σχετικών αποτελεσμάτων.



Σχήμα 6.26: Διάγραμμα που απεικονίζει το μέσο μήκος αναζήτησης για την εύρεση αποτελεσμάτων που περνάνε τους ελέγχους, για τις τρεις Μηχανές Αναζήτησης Κώδικα και τις δύο υλοποιήσεις του Mantissa

### 6.5.3 Σύγκριση του Mantissa με το FAST

Στην ενότητα αυτή συγκρίνουμε το Mantissa με το σύστημα FAST [204]. Χρησιμοποιούμε το σύνολο δεδομένων που παρέχεται για την αξιολόγηση του FAST καθώς το plugin χρησιμοποιεί την CSE Merobase της οποίας η λειτουργία έχει διακοπεί.

#### 6.5.3.1 Σύνολο Δεδομένων

Το σύνολο δεδομένων του FAST αποτελείται από 15 τμήματα και 15 αντίστοιχες περιπτώσεις ελέγχου. Τα τμήματα εμφανίζονται στον Πίνακα 6.6. Τα περισσότερα ερωτήματα περιέχουν μόνο μία μέθοδο και σχετίζονται κυρίως με μαθηματικές συναρτήσεις.

Τα ερωτήματα με περισσότερες από μία μεθόδους, όπως η κλάση “FinancialCalculator”, είναι αρκετά περίπλοκα, επομένως δεν αναμένεται να έχουν επαρκή αποτελέσματα. Ένα ενδιαφέρον ερώτημα είναι το “StackInit” που διαφέρει από “Stack” στο ότι περιέχει έναν constructor. Επιπλέον, τα ερωτήματα “Sort” και “SumArray” ενδέχεται να επιστρέψουν κλάσεις των οποίων οι μέθοδοι διαφέρουν ως προς τους τύπους παραμέτρων (π.χ. List<Integer> αντί για int[]). Καθώς πλέον το FAST δε λειτουργεί, δεν μπορούσαμε να υπολογίσουμε τη μετρική μήκους αναζήτησης. Συνεπώς, αξιολογούμε τα δύο συστήματα με βάση τον αριθμό των αποτελεσμάτων που μεταγλωττίστηκαν και με βάση τον αριθμό των αποτελεσμάτων που πέρασαν τους ελέγχους.



Πίνακας 6.6: Σύνολο δεδομένων για την αξιολόγηση του Mantissa και του FAST

Κλάση	Μέθοδοι
CreditCardValidator	isValid
Customer	setName, getName, setId, getId, setBirthday, getBirthday
Fibonacci	Fibonacci
FinancialCalculator	setPaymentsPerYear, setNumPayments, setInterest, setPresentValue, setFutureValue, getPayment
Gcd	gcd
isDigit	isDigit
isLeapYear	isLeapYear
isPalindrome	isPalindrome
Md5	md5Sum
Mortgage	calculateMortgage
Prime	isPrime
Sort	sort
Stack	push, pop
StackInit	Stack, push, pop
SumArray	sumArray

### 6.5.3.2 Αποτελέσματα

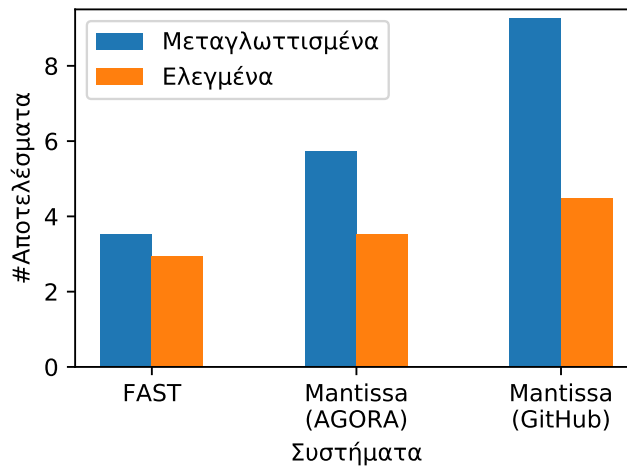
Τα αποτελέσματα για το Mantissa και το FAST παρουσιάζονται στον Πίνακα 6.7.

Πίνακας 6.7: Αποτελέσματα που μεταγλωττίστηκαν και αποτελέσματα που πέρασαν τους ελέγχους για το FAST και τις δύο εκδόσεις του Mantissa, για κάθε ερώτημα και κατά μέσο όρο, όπου η μορφή για κάθε τιμή είναι Πλήθος Ελεγμένων Αποτελεσμάτων / Πλήθος Μεταγλωττισμένων Αποτελεσμάτων / Συνολικό Πλήθος Αποτελεσμάτων

Ερώτημα	FAST	Mantissa	
		AGORA	GitHub
CreditCardValidator	1 / 1 / 1	0 / 0 / 30	0 / 4 / 30
Customer	0 / 0 / 5	10 / 10 / 30	0 / 3 / 30
Fibonacci	4 / 11 / 30	2 / 2 / 30	4 / 12 / 30
FinancialCalculator	0 / 0 / 0	0 / 5 / 30	0 / 2 / 30
Gcd	12 / 12 / 30	5 / 9 / 30	20 / 22 / 30
isDigit	2 / 2 / 30	6 / 6 / 30	10 / 12 / 30
isLeapYear	4 / 4 / 30	1 / 2 / 30	2 / 3 / 30
isPalindrome	4 / 4 / 9	1 / 3 / 30	6 / 18 / 30
Md5	0 / 0 / 24	4 / 7 / 30	1 / 3 / 30
Mortgage	0 / 0 / 0	0 / 2 / 30	0 / 5 / 30
Prime	3 / 4 / 30	1 / 1 / 30	5 / 14 / 30
Sort	1 / 1 / 30	1 / 4 / 30	4 / 5 / 30
Stack	10 / 10 / 30	12 / 14 / 30	9 / 14 / 30
StackInit	3 / 4 / 30	5 / 12 / 30	5 / 14 / 30
SumArray	0 / 0 / 30	5 / 9 / 30	1 / 8 / 30
# Αποτελέσματα	20.6	30.0	30.0
# Μεταγλωττισμένα	3.533	5.733	9.267
# Ελεγμένα	2.933	3.533	4.467

Το Mantissa επιστρέφει περισσότερα μεταγλωττίσιμα αποτελέσματα καθώς και περισσότερα αποτελέσματα που περνάνε τους ελέγχους από το σύστημα FAST. Αναλύοντας περαιτέρω τα αποτελέσματα, παρατηρούμε ότι το FAST είναι αποτελεσματικό για τα μισά ερωτήματα, ενώ υπάρχουν πολλά ερωτήματα για τα οποία τα αποτελέσματά του δεν είναι μεταγλωττίσιμα. Και οι δύο υλοποιήσεις του Mantissa επέστρεψαν πολλά ελεγμένα αποτελέσματα, ακόμη και σε περιπτώσεις όπου το FAST δεν ήταν αποτελεσματικό (π.χ. κλάσεις “Md5” και “SumArray”). Όπως και στο προηγούμενο πείραμά μας, τα μαθηματικά ερωτήματα, όπως τα “FinancialCalculator” και “Prime”, αποτελούν πρόκληση για όλα τα συστήματα, πιθανώς λόγω της ύπαρξης εξαρτήσεων από βιβλιοθήκες. Αξίζει να σημειωθεί ότι η υλοποίηση του Mantissa που συνδέεται με το GitHub φαίνεται καλύτερη από αυτήν που συνδέεται με την AGORA. Ωστόσο, αυτό είναι αναμενόμενο, καθώς το GitHub έχει εκατομμύρια αποθετήρια, ενώ το ευρετήριο της AGORA περιέχει λιγότερα αποθετήρια που μπορεί να μην περιέχουν μαθηματικές συναρτήσεις, όπως αυτές του συνόλου δεδομένων αυτής της ενότητας.

Ο μέσος αριθμός των αποτελεσμάτων που μπορούν να μεταγλωττιστούν και των αποτελεσμάτων που πέρασαν τους ελέγχους παρουσιάζονται επίσης στο Σχήμα 6.27, όπου είναι σαφές ότι οι υλοποιήσεις του Mantissa υπερέρχουν του συστήματος FAST.



Σχήμα 6.27: Διάγραμμα που απεικονίζει το μέσο πλήθος αποτελεσμάτων που μεταγλωττίστηκαν και αυτών που πέρασαν τους ελέγχους για το FAST και τις δύο υλοποιήσεις του Mantissa

#### 6.5.4 Σύγκριση του Mantissa με τον Code Conjurer

Στην ενότητα αυτή συγκρίνουμε το Mantissa με τον Code Conjurer [96]. Ο Code Conjurer έχει δύο εκδοχές αναζήτησης: την “Interface-based” αναζήτηση και την “Adaptation” αναζήτηση. Η πρώτη εκδοχή πραγματοποιεί ακριβή αντιστοίχιση (exact matching) μεταξύ του ερωτήματος και των ανακτημένων αρχείων, ενώ η δεύτερη πραγματοποιεί μια ευέλικτη αντιστοίχιση, αγνοώντας τα ονόματα μεθόδων, τους τύπους κ.λπ. Όπως και με το FAST, ο Code Conjurer δε λειτουργεί πλέον, επομένως χρησιμοποιούμε το σύνολο δεδομένων που παρέχεται από τη σχετική δημοσίευση για να το συγκρίνουμε με το σύστημά μας. Τέλος, δεδομένου ότι ο αριθμός των μεταγλωττίσιμων αποτελεσμάτων καθώς και η κατάταξη των αποτελε-

σμάτων δεν παρέχεται για τον Code Conjurer, συγκρίνουμε τα συστήματα μόνο σε σχέση με τον αριθμό των αποτελεσμάτων που περνάνε τους ελέγχους.

#### 6.5.4.1 Σύνολο δεδομένων

Το σύνολο δεδομένων, που φαίνεται στον Πίνακα 6.8, αποτελείται από 6 τμήματα κώδικα και 6 αντίστοιχες περιπτώσεις ελέγχου. Σε σχέση με το σύνολο δεδομένων που παρουσιάζεται στο [96], παραλείψαμε το τμήμα “ShoppingCart” επειδή εξαρτάται από την εξωτερική κλάση “Product”, επομένως δεν θα μπορούσε να εκτελεστεί. Το σύνολο δεδομένων είναι σχετικά περιορισμένο, ωστόσο περιλαμβάνει διαφορετικούς τύπους ερωτημάτων που θα μπορούσαν να οδηγήσουν σε ενδιαφέροντα συμπεράσματα. Σύνθετα ερωτήματα με πολλές μεθόδους, όπως τα “ComplexNumber” ή “MortgageCalculator”, αναμένεται να είναι δύσκολα για τα συστήματα. Επιπλέον, τα ερωτήματα που περιλαμβάνουν μαθηματικά στοιχεία, όπως το “Calculator” ή το “Matrix”, μπορεί να περιλαμβάνουν εξαρτήσεις από βιβλιοθήκες, επομένως ενδέχεται επίσης και για αυτά να προκύψουν δυσκολίες.

Πίνακας 6.8: Σύνολο δεδομένων για την αξιολόγηση του Mantissa και του Code Conjurer

Κλάση	Μέθοδοι
Calculator	add, sub, div, mult
ComplexNumber	ComplexNumber, add, getRealPart, getImaginaryPart
Matrix	Matrix, set, get, multiply
MortgageCalculator	setRate, setPrincipal, setYears, getMonthlyPayment
Spreadsheet	put, get
Stack	push, pop

#### 6.5.4.2 Αποτελέσματα

Τα αποτελέσματα της σύγκρισής μας, που φαίνονται στον Πίνακα 6.9, εξαρτώνται σε μεγάλο βαθμό από τα συγκεκριμένα ερωτήματα για όλα τα συστήματα. Ένα από τα βασικά πλεονεκτήματα του Code Conjurer, ειδικά της προσέγγισης “Adaptation”, είναι η ικανότητά του να βρει πολλά αποτελέσματα. Η προσέγγιση ανακτά τον μέγιστο αριθμό αρχείων που σχετίζονται με το ερώτημα από το MergoBase και επιχειρεί να εκτελέσει τους ελέγχους κάνοντας ορισμένους μετασχηματισμούς. Σε αυτό το πλαίσιο, δεν είναι συγκρίσιμο με το σύστημά μας, δεδομένου ότι το Mantissa χρησιμοποιεί ορισμένα όρια ως προς τον αριθμό των αρχείων που ανακτώνται από τις CSEs. Αυτό επιτρέπει στο Mantissa να παρουσιάσει αποτελέσματα για ένα ερώτημα εγκαίρως, δηλαδή μέσα σε λιγότερα από 3 λεπτά.

Η χρήση του μέγιστου αριθμού αρχείων έχει ισχυρό αντίκτυπο στον χρόνο απόκρισης του Code Conjurer. Η προσέγγιση Adaptation σε ορισμένες περιπτώσεις απαιτεί αρκετές ώρες για την επιστροφή ελεγμένων τμημάτων κώδικα. Η προσέγγιση Interface-based παρέχει αποτελέσματα πιο έγκαιρα, συνεχίζει ωστόσο να έχει απροσδόκητες αποκλίσεις στον χρόνο απόκρισης μεταξύ διαφορετικών ερωτημάτων (π.χ. χρειάζεται 26 λεπτά για το ερώτημα “Stack”).

Όσον αφορά την αποτελεσματικότητα της προσέγγισης Interface-based του Code Conjurer σε σχέση με το Mantissa, παρατηρούμε ότι κάθε σύστημα λειτουργεί καλύτερα σε διαφορετικά ερωτήματα. Ο Code Conjurer είναι πιο αποτελεσματικός για την εύρεση ελεγμέ-

Πίνακας 6.9: Αποτελέσματα που πέρασαν τους ελέγχους και χρόνοι απόκρισης για τις δύο εκδοχές του Code Conjurer (Interface-based και Adaptation) και τις δύο υλοποιήσεις του Mantissa (με την AGORA και με το GitHub), για κάθε ερώτημα και κατά μέσο όρο

Ερώτημα	Code Conjurer				Mantissa			
	Interface		Adaptation		AGORA		GitHub	
	Ελεγμένα	Χρόνος	Ελεγμένα	Χρόνος	Ελεγμένα	Χρόνος	Ελεγμένα	Χρόνος
Calculator	1	19s	22	20h 24m	0	25s	0	2m 35s
ComplexNumber	0	3s	30	1m 19s	4	35s	7	2m 8s
Matrix	2	23s	26	5m 25s	0	31s	0	2m 51s
MortgageCalculator	0	4s	15	3h 19m	0	31s	0	2m 18s
Spreadsheet	0	3s	4	15h 13m	1	23s	1	2m 28s
Stack	30	26m	30	18h 23m	14	30s	10	2m 53s
Μέσοι όροι	5.500	4m 28s	21.167	9h 34m 17s	3.167	29s	3.000	2m 32s

νων τμημάτων κώδικα για τα ερωτήματα “Calculator” και “Matrix”, ενώ οι δύο υλοποιήσεις του Mantissa ανακτούν περισσότερα χρήσιμα τμήματα για τα ερωτήματα “ComplexNumber” και “Spreadsheet”. Όλα τα συστήματα βρήκαν αρκετά αποτελέσματα για το ερώτημα “Stack”, ενώ το ερώτημα “MortgageCalculator” ήταν αρκετά περίπλοκο οπότε δεν υπήρξαν αποτελέσματα που να περνάνε τους ελέγχους για κανένα από τα συστήματα. Σημειώστε, ωστόσο, ότι ο Code Conjurer εξέτασε 692 τμήματα κώδικα για αυτό το ερώτημα (απαιτώντας 26 λεπτά), ενώ οι υλοποιήσεις του Mantissa περιορίστηκαν σε 200 τμήματα κώδικα.

Τέλος, σημειώνουμε ότι το σύνολο δεδομένων περιέχει μόνο 6 ερωτήματα, ενώ δεν ήταν δυνατή η χρήση κάποιου άλλου συνόλου δεδομένων, καθώς η CSE Merobase δε λειτουργεί. Για αυτό το σύνολο δεδομένων, είναι δύσκολο να καταλήξουμε σε ασφαλή συμπεράσματα σχετικά με την αποτελεσματικότητα των συστημάτων. Ωστόσο, το παραπάνω πείραμα δείχνει ότι το Mantissa έχει σταθερό χρόνο απόκρισης. Όπως φαίνεται στον Πίνακα 6.9, ο χρόνος απόκρισης της υλοποίησης του Mantissa που συνδέεται με την AGORA είναι περίπου 35 δευτερόλεπτα, ενώ για την υλοποίηση που συνδέεται με το GitHub επιστρέφονται τα ελεγμένα τμήματα κώδικα σε περίπου 2.5 λεπτά. Και οι δύο υλοποιήσεις είναι συνεπείς, έχοντας πολύ μικρές αποκλίσεις από αυτούς τους μέσους όρους.

## 6.6 Συμπεράσματα

Σε αυτό το κεφάλαιο, σχεδιάσαμε και υλοποιήσαμε το Mantissa, ένα RSSE που εξάγει το ερώτημα από τον κώδικα του προγραμματιστή και προτείνει επαναχρησιμοποιήσιμα τμήματα κώδικα. Το μοντέλο εξόρυξης του συστήματός μας συνδυάζεται με έναν μετασχηματιστή που πραγματοποιεί μετατροπές σε τμήματα κώδικα για να αντιστοιχίζονται στο ερώτημα του προγραμματιστή, ενώ χρησιμοποιείται και μια μονάδα ελέγχου για τον έλεγχο της λειτουργικότητας των αρχείων πηγαίου κώδικα. Από την αξιολόγησή μας προέκυψε ότι το Mantissa είναι πιο αποτελεσματική από τις σύγχρονες CSEs, ενώ η σύγκρισή του με δύο RSSEs δείχνει ότι παρουσιάζει χρήσιμα αποτελέσματα εγκαίρως και υπερτερεί των υπόλοιπων συστημάτων σε διάφορα σενάρια.

Αναγνωρίζουμε πιθανές μελλοντικές επεκτάσεις για το Mantissa σε διάφορες κατευθύνσεις. Όσον αφορά τον Tester, θα ήταν ενδιαφέρον να εργαστούμε για την επίλυση εξαρ-

τήσεων από βιβλιοθήκες, προκειμένου να αυξηθεί ο αριθμός των αποτελεσμάτων που θα μπορούσαν να μεταγλωττιστούν και εν συνεχεία να περάσουν τους ελέγχους. Επιπλέον, η διενέργεια μιας έρευνας σε προγραμματιστές θα μπορούσε να οδηγήσει σε καλύτερη αξιολόγηση του Mantissa και κατανόηση των λειτουργιών της που έχουν μεγαλύτερη αξία.



# 7

## Εξόρυξη Κώδικα για Επαναχρησιμοποίηση API Snippets

### 7.1 Επισκόπηση

Όπως ήδη αναφέρθηκε στα προηγούμενα κεφάλαια, οι σύγχρονες πρακτικές ανάπτυξης λογισμικού βασίζονται όλο και περισσότερο στην επαναχρησιμοποίηση. Τα συστήματα κατασκευάζονται χρησιμοποιώντας τμήματα που βρίσκονται σε βιβλιοθήκες λογισμικού και ενσωματώνοντάς τα με εντολές πηγαίου κώδικα που συνιστούν αυτά που αποκαλούνται *snippets*. Έτσι, ένα μεγάλο μέρος της προσπάθειας κατά την ανάπτυξη λογισμικού δαπάνεται για την εύρεση των κατάλληλων snippets που αφορούν διάφορες εργασίες (π.χ. ανάγνωση ενός αρχείου CSV, αποστολή ενός αρχείου μέσω ftp κ.α.) και την ενσωμάτωσή τους στον πηγαίο κώδικα του προγραμματιστή. Η χρήση εργαλείων όπως οι μηχανές αναζήτησης, τα προγραμματιστικά forums κ.α., δεν είναι βέλτιστη, δεδομένου ότι απαιτεί την απομάκρυνση από το IDE και την περιήγηση σε ένα πλήθος από σελίδες στο διαδίκτυο. Ο στόχος αυτού είναι να εντοπίσει κανείς τους διαφορετικούς τρόπους επίλυσης του προβλήματος που αντιμετωπίζει πριν επιλέξει και ενσωματώσει την κατάλληλη υλοποίηση.

Καθώς η παραπάνω διαδικασία μπορεί να είναι χρονοβόρα, έχουν προταθεί αρκετές μεθοδολογίες για την αυτοματοποίησή της, οι περισσότερες εκ των οποίων επικεντρώνονται στα προβλήματα της *εξόρυξης χρήσεων API (API usage mining)* και της *εξόρυξης μικρών τμημάτων κώδικα (snippet mining)*. Τα συστήματα εξόρυξης χρήσεων API εξάγουν και παρουσιάζουν παραδείγματα για συγκεκριμένα API βιβλιοθηκών [100, 214–218]. Παρόλο που μπορεί να είναι αποτελεσματικά, τα συστήματα αυτά επικεντρώνονται μόνο στην ανακάλυψη του τρόπου χρήσης ενός API, χωρίς να παρέχουν λύσεις σε γενικές περιπτώσεις ή σε περιπτώσεις κατά τις οποίες ο καθορισμός της βιβλιοθήκης που θα χρησιμοποιηθεί αποτελεί μέρος του ερωτήματος. Επιπλέον, αρκετές από αυτές τις προσεγγίσεις [100, 214, 215] επιστρέφουν ακολουθίες κλήσεων API (return API call sequences) αντί για snippets που είναι έτοιμα προς χρήση.

Από την άλλη πλευρά, τα πιο γενικά συστήματα εξόρυξης snippets [103, 104, 219] χρησιμοποιούν μηχανισμούς ευρετηριοποίησης κώδικα (code indexing) και συνεπώς περιλαμ-

βάνουν λύσεις για πολλά διαφορετικά ερωτήματα. Ωστόσο, έχουν κι αυτά σημαντικούς περιορισμούς. Όσον αφορά τα συστήματα με τοπικά ευρητήρια [103], η ποιότητα και η ποικιλία των αποτελεσμάτων τους συνήθως περιορίζεται από το μέγεθος του ευρητηρίου. Επιπλέον, τα snippets που προτείνονται παρουσιάζονται με τη μορφή λίστας [103, 104, 219], μην επιτρέποντας έτσι την εύκολη διάκριση μεταξύ διαφορετικών υλοποιήσεων (π.χ. χρήση διαφορετικών βιβλιοθηκών για τη διαχείριση αρχείων). Επίσης, η ποιότητα και η επαναχρησιμοποιησιμότητα των αποτελεσμάτων συνήθως δεν αξιολογούνται. Τέλος, ένας κοινός περιορισμός σε ορισμένα συστήματα είναι ότι περιλαμβάνουν κάποια εξειδικευμένη γλώσσα ερωτημάτων, συνεπώς μπορεί να απαιτείται προσπάθεια για να εξοικειωθεί ο προγραμματιστής.

Σε αυτό το κεφάλαιο, σχεδιάζουμε και αναπτύσσουμε το *CodeCatch*, ένα σύστημα που λαμβάνει ερωτήματα σε φυσική γλώσσα και χρησιμοποιεί τη μηχανή αναζήτησης Google για να εξάγει χρήσιμα τμήματα κώδικα από πολλαπλές πηγές στο διαδίκτυο. Το σύστημά μας αξιολογεί περαιτέρω την αναγνωσιμότητα (readability) των ανακτηθέντων snippets, καθώς και την προτίμησή τους/αποδοχή από την κοινότητα προγραμματιστών, χρησιμοποιώντας πληροφορίες από online αποθετήρια. Επιπλέον, το CodeCatch ομαδοποιεί τα snippets με βάση τις κλήσεις API τους, επιτρέποντας στον προγραμματιστή να επιλέξει αρχικά την επιθυμητή υλοποίηση (δηλαδή το επιθυμητό API) και, στη συνέχεια, να επιλέξει το snippet που θα χρησιμοποιήσει.

## 7.2 Βιβλιογραφία για τα Συστήματα Εξόρυξης Snippets και Χρήσεων API

Όπως ήδη αναφέρθηκε, σε αυτό το κεφάλαιο επικεντρώναστε σε συστήματα που λαμβάνουν ερωτήματα για την επίλυση συχνών προγραμματιστικών προβλημάτων και προτείνουν μικρά τμήματα πηγαίου κώδικα που είναι κατάλληλα για επαναχρησιμοποίηση στον πηγαίο κώδικα του προγραμματιστή. Κάποια από τα πρώτα συστήματα αυτού του τύπου, όπως ο Prospector [94] ή το PARSEWeb [95], εστιάζουν στο πρόβλημα της εύρεσης μιας διαδρομής μεταξύ ενός αντικειμένου πηγής και ενός αντικειμένου προορισμού στον πηγαίο κώδικα. Για τον Prospector [94], οι διαδρομές αυτές ορίζονται ως jungloids και η τελική ροή του προγράμματος είναι ένας γράφος jungloid (jungloid graph). Το σχετικό εργαλείο είναι αρκετά αποτελεσματικό για ορισμένα σενάρια επαναχρησιμοποίησης και έχει επίσης τη δυνατότητα να παράγει κώδικα (code generation). Ωστόσο, απαιτεί τη συντήρηση μιας τοπικής βάσης δεδομένων, η οποία μπορεί εύκολα να μείνει ανενημέρωτη. Μια κάπως πιο ευρεία λύση δόθηκε από το PARSEWeb [95], που χρησιμοποιούσε τη μηχανή αναζήτησης κώδικα της Google<sup>1</sup> κι έτσι τα αποτελέσματά του ήταν πάντα ενημερωμένα. Και τα δύο συστήματα, ωστόσο, θεωρούν δεδομένο ότι ο προγραμματιστής γνωρίζει ακριβώς ποια APIs θέλει να χρησιμοποιήσει και ότι αντιμετωπίζει μόνο το πρόβλημα της ενσωμάτωσής τους.

Μια άλλη κατηγορία συστημάτων είναι εκείνα που παράγουν παραδείγματα χρήσεων API με τη μορφή ακολουθιών κλήσεων (call sequences). Τα συστήματα αυτά λαμβάνουν ως είσοδο client code, κώδικα δηλαδή που χρησιμοποιεί το υπό ανάλυση API. Ένα τέτοιο

<sup>1</sup>Όπως ήδη αναφέρθηκε στα προηγούμενα κεφάλαια, η μηχανή αναζήτησης κώδικα της Google, που βρισκόταν στην ηλεκτρονική διεύθυνση <http://www.google.com/codesearch>, σταμάτησε να λειτουργεί το 2013.



σύστημα είναι το MAPO [100], που χρησιμοποιεί τεχνικές εξόρυξης συχνών ακολουθιών (*frequent sequence mining*) προκειμένου να εντοπίσει συχνά πρότυπα χρήσης. Όπως σημειώνουν, ωστόσο, ο Wang και οι συνεργάτες του [214], το MAPO δεν λαμβάνει υπόψη την ποικιλία των προτύπων χρήσης και έτσι εξάγει ένα μεγάλο αριθμό παραδειγμάτων API, πολλά από τα οποία είναι περιττά. Προς τη βελτίωση σε αυτόν τον άξονα, οι συγκεκριμένοι ερευνητές προτείνουν το UP-Miner [214], ένα σύστημα που στοχεύει στην υψηλή κάλυψη (coverage) του API και ταυτόχρονα στην περιεκτικότητα (succinctness) των παραδειγμάτων. Το UP-Miner μοντελοποιεί τον κώδικα που χρησιμοποιεί το API χρησιμοποιώντας γράφους και στη συνέχεια εξάγει από αυτούς συχνές κλειστές διαδρομές/ακολουθίες κλήσεων API (closed API call paths/sequences), χρησιμοποιώντας τον αλγόριθμο BIDE [220]. Το PAM [215] είναι άλλο ένα παρόμοιο σύστημα που εφαρμόζει πιθανοτικές τεχνικές μηχανικής μάθησης για να εξάγει ακολουθίες κλήσεων μεθόδων, οι οποίες αποδεικνύεται ότι είναι πιο αντιπροσωπευτικές από αυτές των MAPO και UP-Miner. Μια ενδιαφέρουσα καινοτομία του PAM [215] είναι η χρήση ενός αυτοματοποιημένου μηχανισμού αξιολόγησης που βασίζεται σε παραδείγματα χρήσεων από τους προγραμματιστές του υπό ανάλυση API.

Εκτός από τα παραπάνω συστήματα, που εξάγουν ακολουθίες κλήσεων API, υπάρχουν επίσης συστήματα που προτείνουν snippets που είναι έτοιμα προς χρήση. Ενδεικτικά, αναφέρουμε τον APIMiner [216], που εφαρμόζει τεχνικές code slicing για να απομονώσει τις εντολές από snippets που αναφέρονται σε κάποιο API. Οι Buse και Weimer [217] χρησιμοποιούν επιπλέον τεχνικές ανάλυσης της ροής δεδομένων (path-sensitive data flow analysis) και τεχνικές αφαίρεσης προτύπων (pattern abstraction) για να προτείνουν snippets σε υψηλότερο επίπεδο αφαίρεσης (abstraction). Ένα άλλο σημαντικό πλεονέκτημα της υλοποίησής τους είναι ότι χρησιμοποιούν τεχνικές ομαδοποίησης για να χωρίσουν τα snippets σε κατηγορίες. Ένα σύστημα που είναι παρόμοιο σε αυτό τον άξονα είναι το eXoaDocs [218], καθώς και αυτό ομαδοποιεί snippets, χρησιμοποιώντας όμως ένα σύνολο σημασιολογικών χαρακτηριστικών από τον αλγόριθμο ανίχνευσης κλώνων κώδικα DECKARD [221].

Αν και έχουν ενδιαφέρον, όλες οι προαναφερθείσες προσεγγίσεις παρέχουν παραδείγματα χρήσης για κάποιο συγκεκριμένο API βιβλιοθήκης και δεν αντιμετωπίζουν την πρόκληση της επιλογής της βιβλιοθήκης που θα χρησιμοποιηθεί. Επιπλέον, αρκετές από αυτές τις προσεγγίσεις δίνουν ως έξοδο ακολουθίες κλήσεων API, αντί για έτοιμες λύσεις με τη μορφή τμημάτων κώδικα. Τέλος, κανένα από τα προαναφερθέντα συστήματα δεν δέχεται ερωτήματα σε φυσική γλώσσα, τα οποία είναι σίγουρα προτιμότερα όταν ο προγραμματιστής θέλει να διατυπώσει ένα πρόβλημα χωρίς να γνωρίζει εκ των προτέρων ποιο API θέλει να χρησιμοποιήσει για την επίλυσή του.

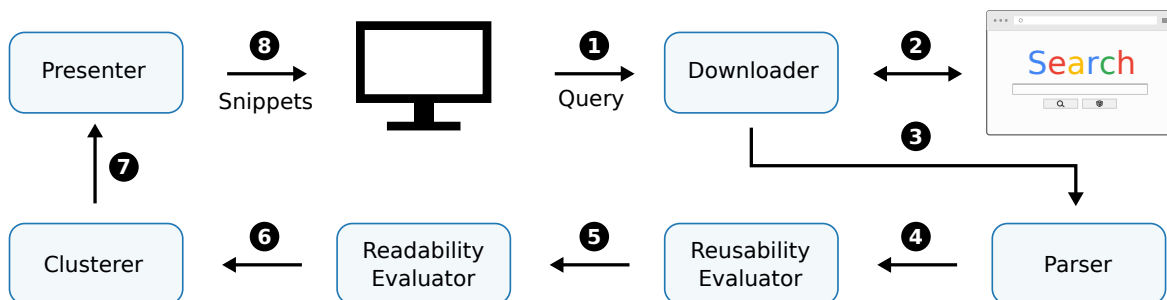
Για την αντιμετώπιση των παραπάνω προκλήσεων, τα σύγχρονα συστήματα επικεντρώνονται σε snippets γενικού τύπου και χρησιμοποιούν κάποιου τύπου επεξεργασία για ερωτήματα φυσικής γλώσσας. Ένα παράδειγμα τέτοιου συστήματος είναι το SnipMatch [103], το σύστημα προτάσεων κώδικα του Eclipse IDE, που έχει αρκετά ενδιαφέρουσες δυνατότητες, όπως π.χ. τη μετονομασία μεταβλητών για την ευκολότερη ενσωμάτωση των snippets. Ωστόσο, το ευρετήριο του πρέπει να δημιουργηθεί από τον προγραμματιστή ο οποίος πρέπει να παρέχει τα snippets και τις αντίστοιχες περιγραφές σε φυσική γλώσσα. Ένα σχετικό σύστημα που επίσης προσφέρεται ως plugin του Eclipse είναι το Blueprint [219]. Το Blueprint χρησιμοποιεί τη μηχανή αναζήτησης της Google για την εύρεση και την κατάταξη μικρών τμημάτων κώδικα, εξασφαλίζοντας έτσι ότι ανακτώνται χρήσιμα αποτελέσματα για σχεδόν όλα τα ερωτήματα. Ένα ακόμα πιο εξελιγμένο σύστημα είναι το Bing Code Search [104], που

χρησιμοποιεί τη μηχανή αναζήτησης Bing για την εύρεση χρήσιμων snippets, και επιπλέον εισάγει ένα πολυπαραμετρικό μοντέλο κατάταξης, καθώς κι ένα σύνολο μετασχηματισμών για την προσαρμογή των snippets στον πηγαίο κώδικα του προγραμματιστή.

Τα παραπάνω συστήματα, ωστόσο, δεν επιτρέπουν στον προγραμματιστή να επιλέξει τη βιβλιοθήκη και ουσιαστικά την υλοποίηση που θα χρησιμοποιηθεί. Επιπλέον, τα περισσότερα από αυτά δεν αξιολογούν τα snippets που ανακτώνται ως προς το βαθμό επαναχρησιμοποίησής τους από την κοινότητα. Αυτό είναι ιδιαίτερα σημαντικό, καθώς οι βιβλιοθήκες που προτιμώνται από τους προγραμματιστές παρουσιάζουν συνήθως κώδικα υψηλής ποιότητας και καλή τεκμηρίωση, ενώ υποστηρίζονται προφανώς από μια ευρύτερη κοινότητα [149, 184]. Σε αυτό το κεφάλαιο, παρουσιάζουμε το CodeCatch, ένα σύστημα εξόρυξης snippets που έχει σχεδιαστεί για να ξεπεράσει τους παραπάνω περιορισμούς. Το CodeCatch χρησιμοποιεί τη μηχανή αναζήτησης της Google για να λαμβάνει ερωτήματα σε φυσική γλώσσα και να κατεβάζει snippets από διαφορετικές πηγές στο διαδίκτυο. Σε αντίθεση με τα υπάρχοντα συστήματα, το εργαλείο μας αξιολογεί όχι μόνο την ποιότητα (αναγνωσιμότητα) των αποσπασμάτων, αλλά και την επαναχρησιμοποιησιμότητα (reusability)/προτίμηση τους από τους προγραμματιστές. Επιπλέον, το CodeCatch χρησιμοποιεί τεχνικές ομαδοποίησης (clustering) για να ομαδοποιήσει τα snippets σύμφωνα με τις κλήσεις API και έτσι επιτρέπει στον προγραμματιστή να διακρίνει εύκολα μεταξύ διαφορετικών υλοποιήσεων.

### 7.3 Το Σύστημα Προτάσεων Snippets CodeCatch

Η αρχιτεκτονική του CodeCatch φαίνεται στο Σχήμα 7.1. Η είσοδος του συστήματος είναι ένα ερώτημα που δίνεται σε φυσική γλώσσα στο υποσύστημα του Downloader, το οποίο στέλνεται στη μηχανή αναζήτησης της Google, για να κατέβουν τμήματα κώδικα από τις σελίδες των αποτελεσμάτων. Στη συνέχεια, ο Parser εξάγει τις κλήσεις API των snippets, ενώ ο Reusability Evaluator βαθμολογεί τα snippets σύμφωνα με το αν χρησιμοποιούνται ευρέως/προτιμώνται από τους προγραμματιστές. Επιπλέον, η αναγνωσιμότητα των τμημάτων κώδικα αξιολογείται από τον Readability Evaluator. Τέλος, ο Clusterer ομαδοποιεί τα τμήματα κώδικα σύμφωνα με τις κλήσεις API, ενώ ο Presenter τα κατατάσσει και τα παρουσιάζει στον προγραμματιστή. Τα υποσυστήματα αυτά αναλύονται στις επόμενες ενότητες.



Σχήμα 7.1: Αρχιτεκτονική του CodeCatch

### 7.3.1 Downloader

Το πρόγραμμα λήψης λαμβάνει ως είσοδο το ερώτημα του προγραμματιστή σε φυσική γλώσσα και επιστρέφει τμήματα κώδικα από διαφορετικές πηγές. Ένα παράδειγμα ερωτήματος που θα χρησιμοποιηθεί σε αυτό το υποκεφάλαιο είναι ένα ερώτημα για την ανάγνωση αρχείων CSV (“How to read a CSV file”). Ο Downloader λαμβάνει το ερώτημα και το επαυξάνει (query augmentation) πριν το στείλει στη μηχανή αναζήτησης της Google. Σημειώστε ότι η μεθοδολογία μας είναι εφαρμόσιμη για οποιαδήποτε γλώσσα προγραμματισμού· ωστόσο, χωρίς βλάβη της γενικότητας, εστιάζουμε ως εφαρμογή στη γλώσσα προγραμματισμού Java. Προκειμένου να διασφαλιστεί ότι τα αποτελέσματα που επιστρέφει η μηχανή αναζήτησης θα στοχεύουν στη γλώσσα Java, το ερώτημα επαυξάνεται χρησιμοποιώντας τους εξής όρους που σχετίζονται με την Java: `java`, `class`, `interface`, `public`, `protected`, `abstract`, `final`, `static`, `import`, `if`, `for`, `void`, `int`, `long` και `double`. Αντίστοιχες λίστες από όρους μπορούν να δημιουργηθούν για την υποστήριξη άλλων γλωσσών προγραμματισμού.

Διαχειριζόμαστε τα αποτελέσματα που επιστρέφονται από τη Google με το Scrapy<sup>2</sup>. Συγκεκριμένα, μετά την ανάκτηση των πρώτων 40 ιστοσελίδων των αποτελεσμάτων, εξάγεται το περιεχόμενο των HTML ετικετών `<pre>` και `<code>`. Η πλειοψηφία του κώδικα που βρίσκεται σε κάποια ιστοσελίδα είναι εντός αυτών των δύο ετικετών. Εκτός από τον κώδικα, συλλέγουμε επίσης πληροφορίες σχετικές με την ιστοσελίδα, που περιλαμβάνουν τη διεύθυνση URL του αποτελέσματος, την κατάταξή του στη λίστα αποτελεσμάτων της Google και τη σχετική θέση κάθε τμήματος κώδικα μέσα στη σελίδα.

### 7.3.2 Parser

Στη συνέχεια, τα snippets αναλύονται χρησιμοποιώντας το εργαλείο εξαγωγής ακολουθιών που περιγράφεται στο [213], το οποίο εξάγει το AST ενός τμήματος κώδικα και το διασχίζει δύο φορές. Την πρώτη φορά εξάγει όλες τις δηλώσεις τύπων (type declarations, για τα πεδία - fields και τις μεταβλητές - variables), και τη δεύτερη φορά εξάγει όλες τις κλήσεις μεθόδων (method invocations), και πιο συγκεκριμένα τις κλήσεις API (API calls). Για παράδειγμα, για το τμήμα κώδικα του Σχήματος 7.2, κατά το πρώτο πέρασμα εξάγονται οι δηλώσεις `line: String`, `br: BufferedReader`, `data: String[]` και `e: Exception`. Στη συνέχεια, και εφόσον αφαιρούνται οι γενικές δηλώσεις (δηλαδή οι δηλώσεις από τους βασικούς τύπους της Java, `int`, `String`, `Exception`, κ.λπ.), γίνεται το δεύτερο πέρασμα και εξάγονται οι σχετικές κλήσεις μεθόδων, που επισημαίνονται στο Σχήμα 7.2. Το αντικείμενο που εκτελεί κάθε κλήση μεθόδου (caller) αντικαθίσταται από τον τύπο του (εκτός από τις κλήσεις constructor για τις οποίες είναι ήδη γνωστοί οι τύποι), για να παραχθούν τελικά οι κλήσεις API `FileReader.__init__`, `BufferedReader.__init__`, `BufferedReader.readLine` και `BufferedReader.close`, όπου το `__init__` χρησιμοποιείται για να δείξει ότι η σχετική κλήση είναι η δημιουργία ενός αντικειμένου μέσω ενός constructor.

Σημειώστε ότι ο Parser λειτουργεί ακόμη και σε περιπτώσεις όπου τα τμήματα κώδικα δεν είναι μεταγλωττίσιμα, ενώ παράλληλα απομονώνει αποτελεσματικά τις κλήσεις API που δεν σχετίζονται με κάποιον τύπο (καθώς, κλήσεις με γενικά ονόματα, όπως π.χ. `close`, δε θα είναι χρήσιμες για την εξόρυξη κλήσεων API). Τέλος, όλα τα τμήματα κώδικα που δεν είναι γραμμένα σε Java και/ή δεν παράγουν κλήσεις API απορρίπτονται σε αυτό το στάδιο.

<sup>2</sup><https://scrapy.org/>

```
String line = "";  
BufferedReader br = null;  
try {  
    br = new BufferedReader(new FileReader("test.csv"));  
    while((line = br.readLine()) != null) {  
        String[] data = line.split(",");  
    }  
    br.close();  
} catch (Exception e) {  
    System.err.println("CSV file cannot be read: " + e);  
}
```

Σχήμα 7.2: Παράδειγμα τμήματος κώδικα για το ερώτημα “How to read a CSV file”

### 7.3.3 Reusability Evaluator

Μετά τη συγκέντρωση των τμημάτων κώδικα και την εξαγωγή των κλήσεων API, το επόμενο βήμα είναι να προσδιορίσουμε αν αναμένεται να είναι χρήσιμα για τον προγραμματιστή. Σε αυτό το πλαίσιο *επαναχρησιμοποιησιμότητας (reusability)*, θέλουμε να κατευθύνουμε τον προγραμματιστή προς αυτό που ονομάζουμε *κοινή πρακτική (common practice)*. Για να το επιτύχουμε αυτό, κάνουμε την υπόθεση ότι τα τμήματα κώδικα με κλήσεις API που χρησιμοποιούνται συχνά από άλλους προγραμματιστές είναι πιο πιθανό να είναι κατάλληλα για επαναχρησιμοποίηση. Αυτή είναι μια λογική υπόθεση, δεδομένου ότι οι απαντήσεις σε συχνά προγραμματιστικά ερωτήματα είναι πιθανό να εμφανίζονται συχνά στον κώδικα διαφόρων έργων. Ως αποτέλεσμα, σχεδιάσαμε τον Reusability Evaluator κατεβάζοντας ένα σύνολο από έργα υψηλής ποιότητας και προσδιορίζοντας το εύρος της επαναχρησιμοποίησης για τις κλήσεις API κάθε τμήματος κώδικα.

Για αυτό το σκοπό χρησιμοποιήσαμε τα 1000 δημοφιλέστερα έργα Java του GitHub, όπου η δημοφιλία καθορίζεται από τον αριθμό των stars τους. Τα έργα κατέβηκαν από το ευρετήριο της AGORA (βλέπε Κεφάλαιο 5). Αυτή η επιλογή έργων εκτός από το ότι είναι διαισθητικά ορθή, υποστηρίζεται επιπλέον από την τρέχουσα έρευνα: τα δημοφιλή έργα συνήθως είναι υψηλής ποιότητας [149], ενώ επίσης περιλαμβάνουν επαναχρησιμοποιήσιμο κώδικα [150] και επαρκή τεκμηρίωση [184]. Ως εκ τούτου, αναμένουμε ότι τα έργα αυτά χρησιμοποιούν τα πιο χρήσιμα APIs με αποτελεσματικό τρόπο.

Μετά το κατέβασμα των έργων, κατασκευάσαμε ένα τοπικό ευρετήριο (index) όπου αποθηκεύονται οι κλήσεις API των έργων, που εξάγονται χρησιμοποιώντας τον Parser<sup>3</sup>. Στη συνέχεια, δίνουμε μια βαθμολογία σε κάθε κλήση API διαιρώντας το πλήθος των έργων στα οποία εμφανίζεται η κλήση με το συνολικό πλήθος των έργων. Για τη βαθμολογία ενός τμήματος κώδικα, υπολογίζουμε το μέσο όρο μεταξύ των βαθμολογιών των κλήσεων API

<sup>3</sup>Σημειώνουμε ότι αυτό το τοπικό ευρετήριο χρησιμοποιείται μόνο για την αξιολόγηση της επαναχρησιμοποιησιμότητας των τμημάτων: η αναζήτησή μας, ωστόσο, δεν περιορίζεται σε αυτήν (όπως συμβαίνει με άλλα συστήματα) καθώς χρησιμοποιούμε μια μηχανή αναζήτησης και ανιχνεύουμε πολλαπλές σελίδες. Για να διασφαλίσουμε ότι ο Reusability Evaluator είναι πάντα ενημερωμένος, θα μπορούσαμε να ανακατασκευάζουμε το ευρετήριο του μαζί με τους κύκλους ανακατασκευής του ευρετηρίου της AGORA.

του. Τέλος, το ευρετήριο περιέχει επίσης τα πλήρη ονόματα (qualified names) των αντικειμένων, αποθηκευμένα έτσι ώστε να μπορούν εύκολα να ανακτηθούν (π.χ. `BufferedReader: java.io.BufferedReader`).

### 7.3.4 Readability Evaluator

Για την κατασκευή ενός μοντέλου που θα αξιολογεί την αναγνωσιμότητα (readability) τμημάτων κώδικα, χρησιμοποιήσαμε ένα σύνολο δεδομένων που είναι δημόσια διαθέσιμο [222]. Το σύνολο δεδομένων περιέχει 100 τμήματα κώδικα μαζί με σχολιασμούς (annotations) από 120 άτομα (συνολικά 12000 annotations). Κατασκευάσαμε το μοντέλο μας ως ένα δυαδικό ταξινομητή (binary classifier) που αξιολογεί αν ένα τμήμα κώδικα είναι *περισσότερο αναγνώσιμο (more readable)* ή *λιγότερο αναγνώσιμο (less readable)*. Αρχικά, για κάθε τμήμα κώδικα εξάγεται ένα σύνολο χαρακτηριστικών (feature set) που σχετίζονται με την αναγνωσιμότητα, που περιλαμβάνουν το μέσο μήκος γραμμής (average line length), το μέσο μήκος μεταβλητών (average identifier length), το μέσο αριθμό σχολίων (average number of comments), κ.α. (η πλήρης λίστα των χαρακτηριστικών είναι διαθέσιμη στο [222]). Στη συνέχεια, εκπαιδεύουμε έναν ταξινομητή AdaBoost για το παραπάνω σύνολο δεδομένων. Χρησιμοποιούμε δένδρα απόφασης (decision trees) για τον εκτιμητή βάσης (base estimator) του ταξινομητή, ενώ το πλήθος των εκτιμητών (number of estimators) και ο ρυθμός εκμάθησης (learning rate) τέθηκαν στις 160 και 0.6, αντίστοιχα. Κατασκευάσαμε το μοντέλο μας χρησιμοποιώντας cross-validation με 10 folds και το μέσο F-measure για όλα τα folds προέκυψε ίσο με 85%, συνεπώς θεωρούμε ότι το μοντέλο είναι επαρκές για την παροχή μιας δυαδικής απόφασης σχετικά με την αναγνωσιμότητα νέων τμημάτων κώδικα.

### 7.3.5 Clusterer

Αφού βαθμολογήσαμε τα snippets τόσο για την επαναχρησιμοποιησιμότητά όσο και για την αναγνωσιμότητά τους, το επόμενο βήμα είναι να τα ομαδοποιήσουμε σύμφωνα με τις διάφορες υλοποιήσεις. Για να εφαρμόσουμε τεχνικές ομαδοποίησης (clustering), χρειάζεται αρχικά να εξάγουμε τα κατάλληλα χαρακτηριστικά (features) από τα τμήματα κώδικα. Μια σχετικά απλή προσέγγιση θα ήταν να ομαδοποιήσουμε τα τμήματα κώδικα εξετάζοντάς τα σαν έγγραφα κειμένου· ωστόσο με αυτή η προσέγγιση δε θα μπορούσαν να διακριθούν οι διάφορες υλοποιήσεις. Ως ένα παράδειγμα, θεωρούμε το τμήμα κώδικα του Σχήματος 7.2 μαζί με αυτό του Σχήματος 7.3. Αν αφαιρέσουμε τα σημεία στίξης και συγκρίνουμε τα δύο τμήματα κώδικα, θα διαπιστώσουμε ότι πάνω από το 60% των όρων του δεύτερου τμήματος βρίσκονται επίσης και στο πρώτο τμήμα. Τα δύο τμήματα κώδικα, ωστόσο, είναι πολύ διαφορετικά: έχουν διαφορετικές κλήσεις συναρτήσεων API και αναφέρονται σε διαφορετικές υλοποιήσεις.

Συνεπώς, αποφασίσαμε να ομαδοποιήσουμε τα τμήματα κώδικα με βάση τις κλήσεις API τους. Για το σκοπό αυτό, χρησιμοποιούμε ένα Μοντέλο Διανυσματικού Χώρου (Vector Space Model - VSM) στο οποίο τα snippets αναπαρίστανται ως έγγραφα (documents) και οι κλήσεις API ως διανύσματα (vectors). Έτσι, αρχικά, κατασκευάζουμε ένα έγγραφο για κάθε τμήμα κώδικα με βάση τις κλήσεις API του. Π.χ., το έγγραφο για το τμήμα κώδικα του Σχήματος 7.2 είναι “`FileReader.__init__ BufferedReader.__init__ BufferedReader.readLine BufferedReader.close`”, ενώ το έγγραφο για το τμήμα του Σχήματος 7.3 είναι “`File.__init__ Scanner.__init__ Scanner.hasNext Scanner.next Scanner.close`”.

---

```

Scanner scanner = null;
try{
    scanner = new Scanner(new File("test.csv"));
    scanner.useDelimiter(",");
    while(scanner.hasNext()) {
        System.out.print(scanner.next() + " ");
    }
    scanner.close();
} catch (Exception e) {
    System.err.println("CSV file cannot be read: " + e);
}

```

---

Σχήμα 7.3: Παράδειγμα τμήματος κώδικα για το ερώτημα “How to read a CSV file”

Στη συνέχεια, χρησιμοποιούμε ένα *διανυσματοποιητή tf-idf (tf-idf vectorizer)* για να εξάγουμε τη διανυσματική αναπαράσταση κάθε εγγράφου. Το βάρος (τιμή διανύσματος) κάθε όρου (term)  $t$  σε ένα έγγραφο  $d$  υπολογίζεται από την εξίσωση:

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \quad (7.1)$$

όπου  $tf(t, d)$  είναι η συχνότητα του όρου  $t$  στο έγγραφο  $d$  και αναφέρεται στο πλήθος των εμφανίσεων της κλήσης API μέσα στο snippet, ενώ  $idf(t, D)$  είναι αντίστροφη συχνότητα εγγράφων του όρου  $t$  στο σύνολο όλων των εγγράφων  $D$ , και αναφέρεται στο πόσο συχνά εμφανίζεται η κλήση API σε όλα τα snippets. Συγκεκριμένα, το  $idf(t, D)$  υπολογίζεται ως:

$$idf(t, D) = 1 + \log \frac{1 + |D|}{1 + d_t} \quad (7.2)$$

όπου  $|d_t|$  είναι το πλήθος των εγγράφων που περιέχουν τον όρο  $t$ , δηλαδή το πλήθος των snippets που περιέχουν τη σχετική κλήση API. Το  $idf$  εξασφαλίζει ότι οι πολύ κοινές κλήσεις API (π.χ. `Exception.printStackTrace`) θα έχουν χαμηλά βάρη, έτσι ώστε να μην υπερτερούν έναντι πιο σημαντικών κλήσεων API.

Πριν την ομαδοποίησης, χρειάζεται επιπλέον να ορίσουμε μια μετρική απόστασης που θα χρησιμοποιηθεί για τη μέτρηση της ομοιότητας μεταξύ δύο διανυσμάτων. Χρησιμοποιούμε την ομοιότητα συνημιτόνου, η οποία ορίζεται για δύο διανύσματα εγγράφων  $d_1$  και  $d_2$  σύμφωνα με την παρακάτω σχέση:

$$cosine\_similarity(d_1, d_2) = \frac{d_1 \cdot d_2}{|d_1| \cdot |d_2|} = \frac{\sum_1^N w_{t_i, d_1} \cdot w_{t_i, d_2}}{\sum_1^N w_{t_i, d_1}^2 \cdot \sum_1^N w_{t_i, d_2}^2} \quad (7.3)$$

όπου  $w_{t_i, d_1}$  και  $w_{t_i, d_2}$  είναι οι τιμές tf-idf του όρου  $t_i$  στα έγγραφα  $d_1$  και  $d_2$  αντίστοιχα, και  $N$  είναι το συνολικό πλήθος των όρων.

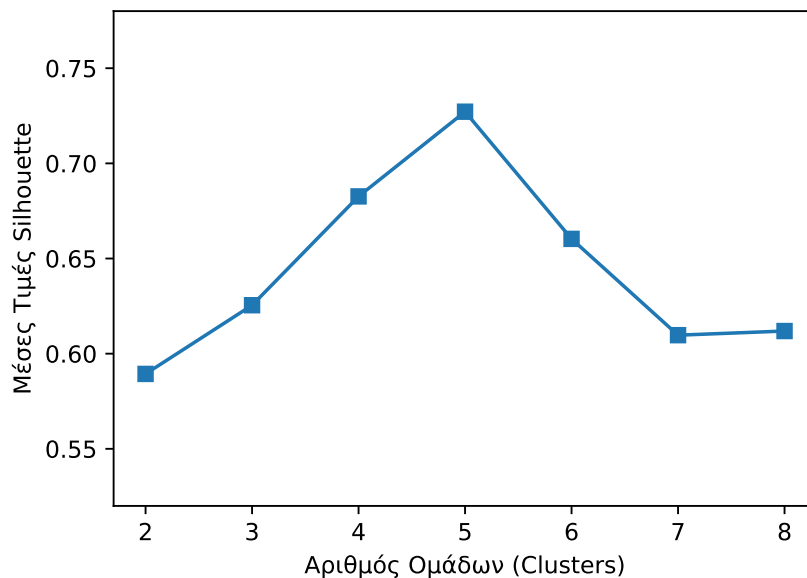
Επιλέγουμε τον K-Means ως αλγόριθμο ομαδοποίησης, καθώς είναι αποτελεσματικός για προβλήματα ομαδοποίησης κειμένου παρόμοια με το δικό μας [223]. Ο αλγόριθμος, ωστόσο, έχει έναν σημαντικό περιορισμό καθώς απαιτεί ως είσοδο τον αριθμό των ομάδων (clusters). Έτσι, για να προσδιοριστεί αυτόματα η καλύτερη τιμή για τον αριθμό των ομάδων,

χρησιμοποιούμε τη μετρική silhouette, καθώς καλύπτει τόσο την ομοιότητα των snippets εντός της ομάδας (cohesion) όσο και τη διαφορετικότητά τους με τα snippets της άλλης ομάδας (separation). Επομένως, αναμένουμε ότι χρησιμοποιώντας την τιμή του silhouette ως παράμετρο βελτιστοποίησης, οι ομάδες που θα δημιουργηθούν θα αντιστοιχούν σε διαφορετικές υλοποιήσεις Εκτελούμε τον K-Means για 2 έως 8 ομάδες (clusters), και υπολογίζουμε κάθε φορά την τιμή του συντελεστή silhouette (silhouette coefficient) για κάθε έγγραφο (snippet) χρησιμοποιώντας την παρακάτω εξίσωση:

$$silhouette(d) = \frac{b(d) - a(d)}{\max(a(d), b(d))} \quad (7.4)$$

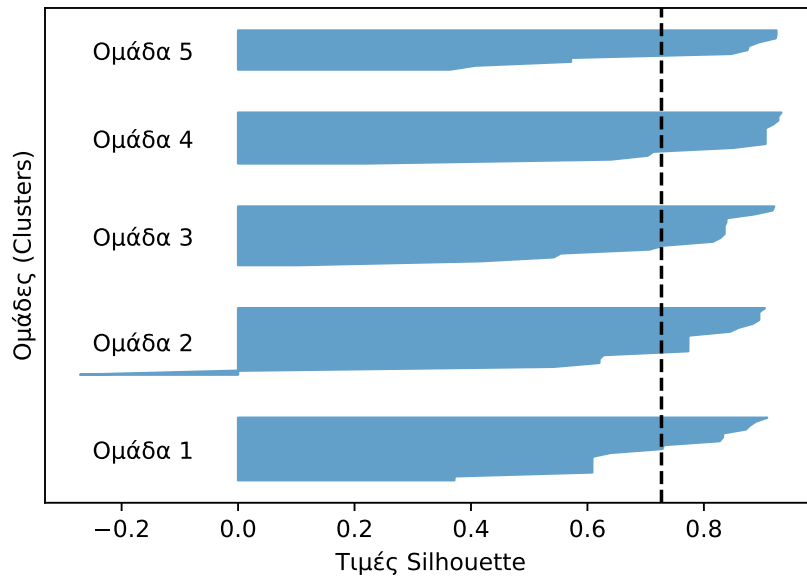
όπου  $a(d)$  είναι η μέση απόσταση του εγγράφου  $d$  από όλα τα άλλα έγγραφα στην ίδια ομάδα, ενώ το  $b(d)$  υπολογίζεται μετρώντας τη μέση απόσταση του  $d$  από τα έγγραφα κάθε μίας από τις άλλες ομάδες και επιλέγοντας τη χαμηλότερη από αυτές τις τιμές (κάθε μία εκ των οποίων αντιστοιχεί σε μια ομάδα). Και για τις δύο παραμέτρους, η απόσταση μεταξύ δύο εγγράφων μετράται χρησιμοποιώντας την εξίσωση (7.3). Τέλος, ο συντελεστής silhouette για μια ομάδα είναι ο μέσος όρος των τιμών silhouette των τμημάτων κώδικα της ομάδας, ενώ το συνολικό silhouette για όλες τις ομάδες είναι ο μέσος όρος των τιμών silhouette όλων των τμημάτων κώδικα.

Ένα παράδειγμα ανάλυσης με το silhouette analysis για το ερώτημα “How to read a CSV file” φαίνεται στα Σχήματα 7.4 και 7.5. Το Σχήμα 7.4 απεικονίζει την τιμή του silhouette για 2 έως 8 ομάδες, όπου είναι σαφές ότι το βέλτιστο πλήθος ομάδων είναι 5.



Σχήμα 7.4: Τιμή silhouette για διαφορετικούς αριθμούς ομάδων για το ερώτημα “How to read a CSV file”

Επιπλέον, οι τιμές του silhouette για τα snippets των πέντε ομάδων απεικονίζονται στο Σχήμα 7.5, όπου επιβεβαιώνεται ότι αυτή η ομαδοποίηση είναι αποτελεσματική καθώς τα περισσότερα δείγματα έχουν υψηλό silhouette και μόνο λίγα έχουν οριακά αρνητικές τιμές.



Σχήμα 7.5: Τιμή silhouette κάθε ομάδας για ομαδοποίηση σε 5 ομάδες για το ερώτημα “How to read a CSV file”

### 7.3.6 Presenter

Ο Presenter διαχειρίζεται την κατάταξη και την παρουσίαση των αποτελεσμάτων. Υλοποιήσαμε το CodeCATCH ως μια διαδικτυακή εφαρμογή (web application). Εφόσον ο προγραμματιστής εισάγει ένα ερώτημα, αρχικά παρουσιάζονται οι ομάδες που αντιστοιχούν σε διαφορετικές υλοποιήσεις. Στο Σχήμα 7.6 φαίνονται ενδεικτικά οι πρώτες τρεις ομάδες που περιέχουν υλοποιήσεις για την ανάγνωση αρχείων CSV.



Σχήμα 7.6: Screenshot του CodeCATCH για το ερώτημα “How to read a CSV file”, όπου απεικονίζονται οι πρώτες τρεις ομάδες

Οι προτεινόμενες υλοποιήσεις περιλαμβάνουν το BufferedReader API (π.χ. όπως στο Σχήμα 7.2), το Scanner API (π.χ. όπως στο Σχήμα 7.3), και το Java CSV reader API<sup>4</sup>. Οι ομάδες κατατάσσονται σύμφωνα με τη βαθμολογία επαναχρησιμοποιησιμότητας API τους, που για κάθε ομάδα ορίζεται ως ο μέσος όρος των βαθμολογιών των snippets της (η βαθμολογία ενός snippet ορίστηκε στην ενότητα 7.3.3). Για κάθε ομάδα, το CodeCATCH παρέχει τις

<sup>4</sup><https://gist.github.com/jaysridhar/d61ea9cbde617606256933378d71751>



5 πιο συχνές δηλώσεις βιβλιοθηκών (imports) και τις 5 πιο συχνά κλήσεις API (API calls), ώστε να μπορεί ο προγραμματιστής να διακρίνει τις διάφορες υλοποιήσεις. Σε περιπτώσεις που δεν υπάρχουν δηλώσεις βιβλιοθηκών μέσα στα τμήματα κώδικα, οι δηλώσεις αυτές εξάγονται χρησιμοποιώντας το ευρετήριο που δημιουργήθηκε στην ενότητα 7.3.3.

Όταν ο προγραμματιστής επιλέξει μια ομάδα, τότε παρουσιάζεται μια λίστα με τα τμήματα κώδικα της ομάδας. Τα τμήματα κώδικα εντός μιας ομάδας κατατάσσονται σύμφωνα με τη βαθμολογία επαναχρησιμοποιησιμότητάς τους, και, σε περίπτωση ισοβαθμίας, σύμφωνα με την απόσταση από το κέντρο της ομάδας (cluster centroid, που υπολογίζεται χρησιμοποιώντας την εξίσωση (7.3)). Έτσι διασφαλίζεται ότι οι πιο συχνές χρήσεις ενός API είναι σε υψηλότερες θέσεις της λίστας. Επιπλέον, για κάθε τμήμα κώδικα, το CodeCatch παρέχει χρήσιμες πληροφορίες. Όπως φαίνεται στο Σχήμα 7.7, οι πληροφορίες περιλαμβάνουν τη βαθμολογία επαναχρησιμοποιησιμότητας (*API Score*) του τμήματος, την απόστασή του από το κέντρο της ομάδας (*Centroid Distance*), την αναγνωσιμότητά του (*Readability*, που μπορεί να είναι είτε χαμηλή - Low, είτε υψηλή - High), τη θέση του αντίστοιχου URL στα αποτελέσματα της Google (*Position*) και της σειράς του snippet στη σελίδα (*Order in page*), το πλήθος των κλήσεων API (*Number of API calls*), και τέλος τον αριθμό των γραμμών κώδικα του τμήματος (*Lines of Code*). Τέλος, ο προγραμματιστής έχει την επιλογή αν θέλει να δει μόνο τα σημεία του κώδικα που περιλαμβάνουν κλήσεις API, ενώ μπορεί επίσης να πλοηγηθεί στη σελίδα από την οποία ανακτήθηκε το snippet.



Σχήμα 7.7: Screenshot του CodeCatch για το ερώτημα “How to read a CSV file”, όπου απεικονίζεται ένα τμήμα κώδικα

## 7.4 Αξιολόγηση

### 7.4.1 Μηχανισμός Αξιολόγησης

Η σύγκριση του CodeCatch με παρόμοιες συστήματα δεν πραγματοποιείται με άμεσο τρόπο, καθώς αρκετά από αυτά τα συστήματα εστιάζουν στην εξόρυξη μεμονωμένων APIs, ενώ άλλα δε συντηρούνται και/ή δεν είναι διαθέσιμα online. Η αξιολόγησή μας επικεντρώνεται κυρίως στη δυνατότητα επαναχρησιμοποίησης των αποτελεσμάτων, ενώ το σύστημα που είναι το πιο παρόμοιο με το δικό μας είναι το Bing Code Search [104], το οποίο ωστόσο στοχεύει τη γλώσσα προγραμματισμού C#. Επομένως, αποφασίσαμε να αξιολογήσουμε το

σύστημά μας σε σχέση με την επαναχρησιμοποιησιμότητα ενάντια στη μηχανή αναζήτησης Google. Για αυτό το σκοπό, κατασκευάζουμε ένα σύνολο δεδομένων με συχνά προγραμματιστικά ερωτήματα που φαίνονται στον Πίνακα 7.1, μαζί με σχετικά στατιστικά.

Πίνακας 7.1: Στατιστικά Ερωτημάτων του Συνόλου Δεδομένων

A/A	Ερώτημα	# Ομάδες	# Snippets
1	How to read CSV file	5	76
2	How to generate MD5 hash code	5	65
3	How to send packet via UDP	5	34
4	How to split string	4	22
6	How to upload file to FTP	4	31
5	How to send email	5	79
7	How to initialize thread	6	51
8	How to connect to a JDBC database	5	42
9	How to read ZIP archive	6	82
10	How to play audio file	6	45

Ο σκοπός της αξιολόγησής μας είναι διττός: επιθυμούμε όχι μόνο να ελέγξουμε αν τα τμήματα κώδικα του συστήματός μας είναι σχετικά, αλλά και να προσδιορίσουμε αν ο προγραμματιστής μπορεί εύκολα να βρει τμήματα κώδικα για όλα τα διαφορετικά APIs που είναι σχετικά με ένα ερώτημα. Έτσι, αρχικά, σημειώνουμε (annotate) τα τμήματα κώδικα για όλα τα ερωτήματα ως σχετικά ή μη σχετικά. Για να είναι η αξιολόγηση όσο το δυνατόν πιο αντικειμενική και συστηματική, η διαδικασία σχολιασμού (annotation) εκτελέστηκε χωρίς γνώση της κατάταξης των αποτελεσμάτων. Για τον ίδιο λόγο, ο σχολιασμός διατηρήθηκε πολύ απλός: ένα snippet σημειώνεται ως σχετικό αν και μόνο αν ο κώδικάς του καλύπτει τη λειτουργικότητα που περιγράφεται από το ερώτημα. Για παράδειγμα, για το ερώτημα “How to read CSV file”, τα τμήματα κώδικα που έχουν ως στόχο την ανάγνωση ενός αρχείου CSV θεωρούνται σχετικά, ανεξάρτητα από το μέγεθος ή την πολυπλοκότητά τους, από πιθανές εξωτερικές εξαρτήσεις κ.λπ.

Όπως ήδη αναφέρθηκε, τα τμήματα κώδικα βρίσκονται σε ομάδες, όπου κάθε ομάδα περιλαμβάνει διαφορετικές κλήσεις API και συνεπώς αντιστοιχεί σε διαφορετική υλοποίηση. Ως εκ τούτου, η αξιολόγηση της σχετικότητας των αποτελεσμάτων θα γίνει ανά ομάδα, υποθέτοντας δηλαδή ότι ο προγραμματιστής θα επιλέξει πρώτα την επιθυμητή υλοποίηση και στη συνέχεια θα πλοηγηθεί στην αντίστοιχη ομάδα. Για το σκοπό αυτό, συγκρίνουμε τα αποτελέσματα κάθε ομάδας (υλοποίησης) με τα αποτελέσματα της μηχανής αναζήτησης Google. Οι ομάδες του CodeCATCH παρέχουν ήδη λίστες από snippets, ενώ για τη Google κατασκευάζουμε μια λίστα από snippets, υποθέτοντας ότι ο προγραμματιστής ανοίγει το πρώτο URL, εξετάζει στη συνέχεια τα snippets του URL από πάνω προς τα κάτω, μετά ανοίγει το δεύτερο URL κ.ο.κ.

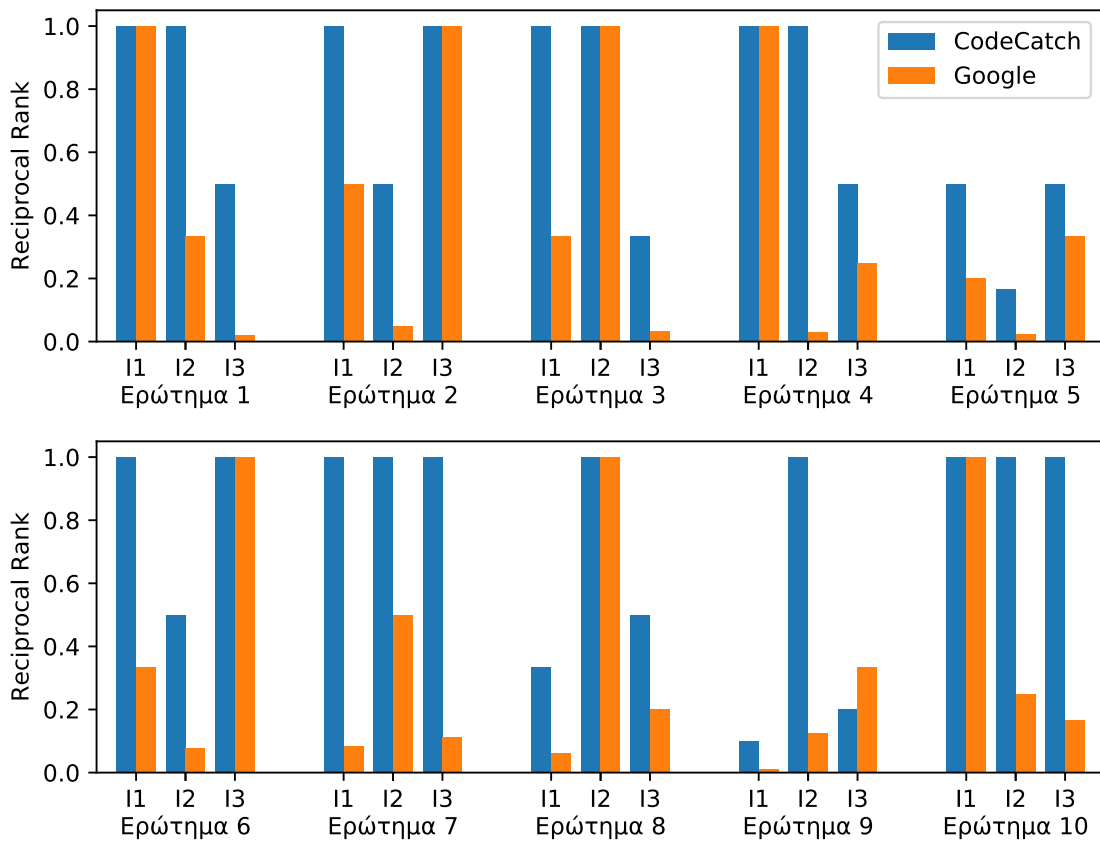
Κατά την αξιολόγηση των αποτελεσμάτων κάθε ομάδας, επιθυμούμε να βρούμε τμήματα κώδικα που να σχετίζονται όχι μόνο με το ερώτημα αλλά και με τις αντίστοιχες κλήσεις API. Ως εκ τούτου, για την αξιολόγηση κάθε ομάδας, σημειώνουμε (annotate) τα αποτελέσματα και των δύο συστημάτων έτσι ώστε να θεωρηθούν σχετικά, όταν ανήκουν και στην αντίστοιχη υλοποίηση. Αυτό, αναμφισβήτητα, παράγει λιγότερο αποτελεσματικές λίστες τμη-

μάτων κώδικα για τη μηχανή αναζήτησης Google· ωστόσο, σημειώστε ότι ο σκοπός μας δεν είναι να αμφισβητήσουμε τα αποτελέσματα της Google όσον αφορά τη συνάφεια με το ερώτημα, αλλά να δείξουμε πόσο εύκολο ή δύσκολο είναι για τον προγραμματιστή να εξετάσει τα αποτελέσματα και να απομονώσει τους διαφορετικούς τρόπους απάντησης στο ερώτημά του.

Για κάθε ερώτημα, αφού κατασκευάσαμε τις λίστες τμημάτων κώδικα για κάθε ομάδα του CodeCatch και για τη Google, συγκρίνουμε τις λίστες χρησιμοποιώντας τη μετρική της αντίστροφης κατάταξης (*reciprocal rank*). Αυτή η μετρική επιλέχθηκε καθώς χρησιμοποιείται συχνά για την αξιολόγηση συστημάτων ανάκτησης πληροφοριών καθώς και για συστήματα παρόμοια με το δικό μας [104]. Έχοντας μια λίστα αποτελεσμάτων, το *reciprocal rank* για ένα ερώτημα υπολογίζεται ως το αντίστροφο της θέσης του πρώτου σχετικού αποτελέσματος. Για παράδειγμα, εάν το πρώτο σχετικό αποτέλεσμα είναι στην πρώτη θέση, τότε το *reciprocal rank* είναι ίσο με  $1/1 = 1$ , αν το πρώτο σχετικό αποτέλεσμα είναι στη δεύτερη θέση, τότε το *reciprocal rank* είναι ίσο με  $1/2 = 0.5$ , κ.λπ.

### 7.4.2 Αποτελέσματα Αξιολόγησης

Στο Σχήμα 7.8 φαίνονται οι τιμές του *reciprocal rank* των CodeCatch και Google για τα τμήματα κώδικα που ανήκουν στις τρεις πιο δημοφιλείς υλοποιήσεις για κάθε ερώτημα.



Σχήμα 7.8: Reciprocal Rank των CodeCatch και Google για τις τρεις πιο δημοφιλείς υλοποιήσεις (I1, I2, I3) για κάθε ερώτημα

Αρχικά, αν ερμηνεύσουμε αυτό το διάγραμμα από τη σκοπιά της σχετικότητας των αποτελεσμάτων, φαίνεται ότι και τα δύο συστήματα είναι πολύ αποτελεσματικά. Συγκεκριμένα, αν θεωρήσουμε ότι ο προγραμματιστής θα απαιτούσε ένα σχετικό τμήμα κώδικα ανεξάρτητα από την υλοποίηση, τότε για τα περισσότερα ερωτήματα, τόσο το CodeCatch όσο και η Google επιστρέφουν ένα σχετικό αποτέλεσμα στην πρώτη θέση (reciprocal rank ίσο με 1).

Εάν, ωστόσο, επικεντρωθούμε σε όλες τις διαφορετικές υλοποιήσεις για κάθε ερώτημα, μπορούμε να εξάγουμε κάποια ενδιαφέροντα συμπεράσματα. Ας εξετάσουμε, για παράδειγμα, το πρώτο ερώτημα (“How to read a CSV file”). Σε αυτήν την περίπτωση, εάν ο προγραμματιστής επιθυμεί την πιο δημοφιλή υλοποίηση, που είναι ο `BufferedReader` (I1), τότε τόσο το CodeCatch όσο και το Google επιστρέφουν ένα σχετικό snippet στην πρώτη θέση. Ομοίως, εάν επιθυμούσαμε να χρησιμοποιήσουμε την υλοποίηση του `Scanner` (I2) ή την υλοποίηση του `Java CSV reader` (I3), τότε το σύστημά μας θα επιστρέφει ένα χρήσιμο snippet στην κορυφή της δεύτερης ομάδας ή αντίστοιχα στη δεύτερη θέση της τρίτης ομάδας (reciprocal rank ίσο με 0.5). Από την άλλη πλευρά, χρησιμοποιώντας τη Google θα χρειαζόταν η εξέταση περισσότερων αποτελεσμάτων (3 και 50 αποτελεσμάτων για τα I2 και I3 αντίστοιχα, καθώς τα αντίστοιχα reciprocal ranks είναι 0.33 and 0.02 αντίστοιχα). Παρόμοια συμπεράσματα μπορούν να εξαχθούν για τα περισσότερα ερωτήματα.

Ένα άλλο σημαντικό σημείο σύγκρισης των δύο συστημάτων είναι το εάν επιστρέφουν τις πιο δημοφιλείς υλοποιήσεις στην κορυφή της λίστας τους. Το CodeCatch είναι σαφώς πιο αποτελεσματικό από τη Google σε αυτή την πτυχή. Ας εξετάσουμε, για παράδειγμα, το έκτο ερώτημα. Σε αυτή την περίπτωση, η πιο δημοφιλής υλοποίηση βρίσκεται στην τρίτη θέση της Google, ενώ το snippet που βρέθηκε στην πρώτη θέση αντιστοιχεί σε μια λιγότερο δημοφιλή υλοποίηση. Αυτό ισχύει επίσης σε αρκετά άλλα ερωτήματα (ερωτήματα 2, 3, 5, 7). Έτσι, θα μπορούσε κανείς να υποστηρίξει ότι το CodeCatch όχι μόνο παρέχει στον προγραμματιστή όλες τις διαφορετικές υλοποιήσεις API για το ερώτημά του, αλλά επιπλέον τον βοηθά να επιλέξει τις πιο δημοφιλείς από αυτές τις υλοποιήσεις, που είναι αυτές που προτιμώνται πιο συχνά.

## 7.5 Συμπεράσματα

Σε αυτό το κεφάλαιο, προτείναμε ένα σύστημα που εξάγει μικρά τμήματα κώδικα (snippets) από το διαδίκτυο και αξιολογεί την αναγνωσιμότητά τους καθώς και την επαναχρησιμοποιησιμότητά τους με βάση την προτίμησή τους από τους προγραμματιστές. Επιπλέον, το σύστημά μας, το CodeCatch, ομαδοποιεί τα τμήματα κώδικα με βάση τις κλήσεις API σε ομάδες που αντιστοιχούν σε διαφορετικές υλοποιήσεις. Με αυτόν τον τρόπο, ο προγραμματιστής μπορεί να επιλέξει μεταξύ πιθανών λύσεων σε συχνά προγραμματιστικά ερωτήματα, ακόμα και σε περιπτώσεις που δεν γνωρίζει εκ των προτέρων ποιο API να χρησιμοποιήσει.

Οι ιδέες για μελλοντική εργασία σχετικά με το CodeCatch είναι αρκετές. Αρχικά, μπορούμε να επεκτείνουμε τη μεθοδολογία βαθμολόγησης ώστε να συμπεριλάβουμε π.χ. τη θέση του URL κάθε τμήματος κώδικα στα αποτελέσματα Google, κ.α. Επιπλέον, τα τμήματα κώδικα μπορούν να συνοψιστούν (summarization) χρησιμοποιώντας πληροφορίες από την ομαδοποίηση (π.χ. αφαιρώντας εντολές που εμφανίζονται σε πολύ λίγα snippets μέσα σε μια ομάδα). Τέλος, μια ενδιαφέρουσα ιδέα θα ήταν η διεξαγωγή μιας έρευνας προκειμένου να αξιολογηθεί περαιτέρω το CodeCatch ως προς την αποτελεσματικότητά του για την ανάκτηση χρηστικών τμημάτων κώδικα.

# 8

## Βελτίωση Ερωταπαντήσεων Υλοποίησης με Snippets

### 8.1 Επισκόπηση

Στο προηγούμενο κεφάλαιο, αναλύσαμε τις προκλήσεις που αντιμετωπίζει ένας προγραμματιστής όταν προσπαθεί να βρει λύσεις στο διαδίκτυο για διάφορα προγραμματιστικά ερωτήματα. Εστιάσαμε στην αναζήτηση υλοποιήσεων που καλύπτουν κάποια επιθυμητή λειτουργικότητα, χωρίς να περιοριστούμε σε αποτελέσματα που κάνουν χρήση συγκεκριμένων βιβλιοθηκών. Σε αυτό το κεφάλαιο, εστιάζουμε στην πρόκληση που προκύπτει από την εξόρυξη τμημάτων κώδικα: τον έλεγχο του αν ένα τμήμα κώδικα που επιλέχθηκε (ή γενικά υλοποιήθηκε) από τον προγραμματιστή είναι ορθό και επίσης αν αποτελεί βέλτιστη λύση στο εκάστοτε πρόβλημα. Συγκεκριμένα, υποθέτουμε ότι ο προγραμματιστής είτε έγραψε ένα τμήμα πηγαίου κώδικα είτε το βρήκε και το ενσωμάτωσε στον πηγαίο κώδικα του, όμως το τμήμα αυτό δεν εκτελέστηκε όπως αναμενόταν. Σε ένα τέτοιο σενάριο, μια διαδικασία που συχνά ακολουθεί είναι η αναζήτηση σε διαδικτυακές πηγές, όπως π.χ. το Stack Overflow, προκειμένου να διερευνηθεί ο τρόπος με τον οποίο άλλοι προγραμματιστές αντιμετώπισαν το συγκεκριμένο πρόβλημα ή αν έχουν ήδη βρει κάποια λύση. Στην περίπτωση που δεν βρεθεί κάποια έτοιμη λύση στο πρόβλημα, ο προγραμματιστής συνήθως υποβάλλει μια νέα ερώτηση, με την ελπίδα ότι η κοινότητα θα ανταποκριθεί.

Η υποβολή μιας νέας ανάρτησης-ερώτησης (question post) στο Stack Overflow απαιτεί αρκετά βήματα: ο προγραμματιστής πρέπει να σχηματίσει ένα ερώτημα με έναν σαφή τίτλο (title), να εξηγήσει περαιτέρω το πρόβλημα στο σώμα (body) της ανάρτησης, να απομονώσει σχετικά τμήματα (snippets) από τον κώδικα, και πιθανώς να προσθέσει κάποιες ετικέτες (tags) στο ερώτημα (π.χ. “swing” ή “android”) προκειμένου να τραβήξει την προσοχή των μελών της κοινότητας που είναι εξοικειωμένα με τις συγκεκριμένες τεχνολογίες. Στις περισσότερες περιπτώσεις, ωστόσο, το πρώτο βήμα πριν το σχηματισμό μιας ερώτησης είναι να διερευνηθεί εάν έχει ήδη αναρτηθεί κάποια σχετική ερώτηση. Για να βρεθούν σχετικές αναρτήσεις, μπορεί κανείς αρχικά να πραγματοποιήσει αναζήτηση στους τίλους των αναρτήσεων, ενώ για την περαιτέρω βελτίωση των αποτελεσμάτων απαιτείται η συμπερίληψη

ετικετών ή ακόμα και η αναζήτηση στα σώματα των αναρτήσεων. Για την πιο αποτελεσματική αναζήτηση απαιτείται πιθανώς η συμπερίληψη του τίτλου, των ετικετών και του σώματος, απαιτείται δηλαδή ο προγραμματιστής να σχηματίσει μια πλήρη ανάρτηση. Αυτή η διαδικασία είναι προφανώς πολύπλοκη (για τους νέους προγραμματιστές) και χρονοβόρα· για παράδειγμα ο κώδικας του προβλήματος ενδέχεται να μην απομονώνεται εύκολα ή ο προγραμματιστής μπορεί να μην γνωρίζει ποιες ετικέτες να χρησιμοποιήσει.

Σε αυτό το κεφάλαιο, εξετάζουμε το πρόβλημα της εύρεσης παρόμοιων αναρτήσεων-ερωτήσεων στο Stack Overflow, χρησιμοποιώντας διαφορετικά στοιχεία για την αναζήτηση. Εκτός από τίτλους, ετικέτες ή σώματα αναρτήσεων, διερευνούμε αν ο προγραμματιστής θα μπορούσε να αναζητήσει παρόμοιες ερωτήσεις χρησιμοποιώντας τμήματα πηγαίου κώδικα, έτσι ώστε να μην απαιτείται ο πλήρης σχηματισμός μιας ερώτησης. Η βασική συνεισφορά μας είναι μια μεθοδολογία υπολογισμού της ομοιότητας ερωτήσεων, που μπορεί επιπλέον να χρησιμοποιηθεί για να προτείνει παρόμοιες ερωτήσεις (question posts) σε χρήστες της κοινότητας που προσπαθούν να βρουν λύση στα προβλήματά τους [213]. Χρησιμοποιώντας τη μεθοδολογία μας, οι προγραμματιστές θα μπορούν να ελέγξουν αν τα snippets τους (είτε είναι γραμμένα από τους ίδιους είτε επαναχρησιμοποιούν κώδικα) αποτελούν καλές επιλογές για τη λειτουργικότητα που πρέπει να καλυφθεί. Αναφέρουμε, τέλος, ότι η μεθοδολογία μας θα μπορούσε επίσης να είναι χρήσιμη για τον εντοπισμό παρόμοιων ερωτήσεων, συνεπώς θα μπορούσε να χρησιμοποιηθεί από τα μέλη της κοινότητας ή τους συντονιστές (moderators), για την αναγνώριση πιθανών συνδεδεμένων ερωτήσεων (linked questions) ή διπλότυπων ερωτήσεων (duplicate questions) ή ακόμα και για τον εντοπισμό παρόμοιων ερωτήσεων στις οποίες μπορούν να δώσουν απάντηση και άρα να συμβάλλουν στην κοινότητα.

## 8.2 Συλλογή και Προεπεξεργασία Δεδομένων

Η μεθοδολογία μας χρησιμοποιεί το επίσημο dump του Stack Overflow από τις 26 Σεπτεμβρίου 2014, που παρέχεται από το [224]. Χρησιμοποιήσαμε το Elasticsearch [178] για την αποθήκευση δεδομένων, καθώς παρέχει κατάλληλα ευρετήρια και υποστηρίζει τη γρήγορη εκτέλεση ερωτημάτων σε μεγάλο όγκο δεδομένων. Η αποθήκευση στο Elasticsearch είναι απλή· κάθε εγγραφή δεδομένων είναι ένα έγγραφο (*document*), τα έγγραφα αποθηκεύονται μέσα σε συλλογές (*collections*), και οι συλλογές αποθηκεύονται σε ένα ευρετήριο (*index*) (βλέπε Κεφάλαιο 5 για περισσότερες πληροφορίες για το Elasticsearch). Ορίσαμε μια συλλογή *posts* για τις αναρτήσεις, η οποία περιέχει τις αναρτήσεις για ερωτήσεις του Stack Overflow σχετικές με Java (αυτές δηλαδή που έχουν την ετικέτα “java”)<sup>1</sup>.

### 8.2.1 Εξαγωγή Δεδομένων από Ερωτήσεις

Ένα παράδειγμα ανάρτησης στο Stack Overflow απεικονίζεται στο Σχήμα 8.1. Τα βασικά στοιχεία της ανάρτησης είναι: ο τίτλος (στο άνω μέρος του Σχήματος 8.1), το σώμα και οι ετικέτες (στο κάτω μέρος του Σχήματος 8.1). Αυτά τα τρία στοιχεία αποθηκεύονται και ευρετηριοποιούνται για κάθε ανάρτηση στο Elasticsearch ώστε να υποστηρίζεται η ανάκτηση των πιο σχετικών ερωτήσεων. Ο τίτλος και το σώμα αποθηκεύονται ως κείμενο, ενώ

<sup>1</sup> Αν και η μεθοδολογία μας είναι ως επί το πλείστον ανεξάρτητη της γλώσσα προγραμματισμού (language-agnostic), εστιάζουμε σε ερωτήσεις Java ως μια εφαρμογή.

οι ετικέτες αποθηκεύονται ως μια λίστα από λέξεις-κλειδιά (keywords). Καθώς το σώμα είναι σε html, χρησιμοποιήσαμε επιπλέον έναν αναλυτή html (html parser) για να εξάγουμε τα τμήματα κώδικα (snippets) για κάθε ερώτηση.

### How to iterate through all buttons of grid layout?

I have a 2x2 grid of buttons

```
JFrame frame = new JFrame("myframe");
JPanel panel = new JPanel();
Container pane = frame.getContentPane();
GridLayout layout = new GridLayout(2,2);
panel.setLayout(layout);
panel.add(upperLeft);
panel.add(upperRight);
panel.add(lowerLeft);
panel.add(lowerRight);
pane.add(panel);
```

where upperLeft, upperRight, etc. are buttons. How can I iterate through all of the buttons?

java swing jbutton

Σχήμα 8.1: Παράδειγμα ανάρτησης στο Stack Overflow

## 8.2.2 Αποθήκευση και Ευρετηριοποίηση Δεδομένων

Σε αυτήν την ενότητα, αναλύεται ο τρόπος με τον οποίο ευρετηριοποιούνται οι αναρτήσεις στο Elasticsearch. Όπως ήδη αναφέρθηκε στο Κεφάλαιο 5, το Elasticsearch δεν αποθηκεύει απλώς τα δεδομένα, αλλά επιπλέον αναλύει κάθε πεδίο (field) με χρήση κάποιου αναλυτή (*analyzers*), ούτως ώστε οι αναζητήσεις στο ευρετήριο να είναι αποδοτικές.

Ο τίτλος μιας ανάρτησης αναλύεται χρησιμοποιώντας τον αναλυτή *standard* (*standard analyzer*) του Elasticsearch. Ο αναλυτής αυτός αποτελείται από έναν τμηματοποιητή (tokenizer) που διαχωρίζει το κείμενο σε όρους και ένα φίλτρο όρων (token filter) που αφαιρεί κάποια stopwords (στην περίπτωση μας τα stopwords της αγγλικής γλώσσας). Οι ετικέτες αποθηκεύονται σε ένα πεδίο τύπου array χωρίς κάποια ανάλυση, καθώς είναι ήδη διαχωρισμένες και σε πεζά. Το σώμα κάθε ανάρτησης αναλύεται χρησιμοποιώντας έναν αναλυτή *html* (*html analyzer*), που είναι παρόμοιος με τον αναλυτή *standard*, ωστόσο πρώτα αφαιρεί την html σύνταξη (html tags) χρησιμοποιώντας το φίλτρο *html\_strip character* του Elasticsearch. Οι τίτλοι, οι ετικέτες και τα σώματα ερωτήσεων αποθηκεύονται στα πεδία “Title”, “Tags” και “Body”, αντίστοιχα.

Όσον αφορά τα τμήματα κώδικα που εξάγονται από τα σώματα των ερωτήσεων, χρειαζόμαστε μια αναπαράσταση που να περιγράφει όχι μόνο τους όρους τους (οι οποίοι ήδη περιλαμβάνονται στο πεδίο “Body”), αλλά και τη δομή τους. Η επιλογή της κατάλληλης αναπαράστασης κώδικα (source code representation) είναι ένα ενδιαφέρον πρόβλημα για το οποίο υπάρχουν πολλές διαφορετικές λύσεις. Οι ‘σύνθετες αναπαραστάσεις, όπως π.χ. τα Αφηρημένα Συντακτικά Δένδρα (Abstract Syntax Trees - ASTs) ή οι Γράφοι Εξάρτησης Προγράμματος (Program Dependency Graphs - PDGs), διατηρούν και τις περισσότερες πληροφορίες, ωστόσο δεν μπορούν να χρησιμοποιηθούν στην περίπτωση τμημάτων κώδικα που

είναι ελλιπή, όπως αυτό του Σχήματος 8.1. Για παράδειγμα, η εξαγωγή πληροφοριών από το AST και η αντιστοίχιση μεταξύ κλάσεων και μεθόδων (όπως π.χ. στο [93]) δεν είναι δυνατή, καθώς μπορεί να μην υπάρχουν πληροφορίες σχετικά με κλάσεις και μεθόδους.

Άλλες ενδιαφέρουσες αναπαραστάσεις χρησιμοποιούνται στο πρόβλημα της εξόρυξης κλήσεων API (API call mining), όπου τα snippets συχνά αναπαρίστανται ως ακολουθίες (sequences) [95, 101]. Ωστόσο, όπως ήδη αναφέρθηκε στο προηγούμενο κεφάλαιο, οι σχετικές προσεγγίσεις αφορούν το πρόβλημα του τρόπου κλήσης μιας μεθόδου API, συνεπώς δεν είναι βέλτιστες για το ευρύτερο πρόβλημα της ομοιότητας τμημάτων κώδικα. Άλλες προσεγγίσεις περιλαμβάνουν την εξαγωγή των τύπων από το τμήμα κώδικα [225, 226] ή την τμηματοποίηση του κώδικα και μετέπειτα την επεξεργασία του ως bag-of-words [227]. Σε κάθε περίπτωση, όμως, με αυτές τις αναπαραστάσεις δε διατηρείται η πληροφορία της δομής του κώδικα. Επομένως, χρειάστηκε η κατασκευή μιας αναπαράστασης που να χρησιμοποιεί τόσο τους τύπους αντικειμένων όσο και τη δομή των snippets.

Η αναπαράσταση που προτείνουμε είναι μια ακολουθία που παράγεται από τρία είδη εντολών (instruction types): αναθέσεις (assignments - AM), κλήσεις συναρτήσεων (function calls - FC) και δημιουργίες αντικειμένων (class instantiations - CI). Αποφεύγουμε πιο σύνθετες αναπαραστάσεις, όπως π.χ. αυτή στο [228], αφού τα ελλιπή τμήματα κώδικα ενδέχεται να μην περιέχουν όλες τις δηλώσεις μεταβλητών και συναρτήσεων. Για παράδειγμα, στο τμήμα κώδικα του Σχήματος 8.1, δε γνωρίζουμε τίποτα για το αντικείμενο “upperLeft”. Θα μπορούσε να είναι ένα πεδίο (field) ή ένα αντικείμενο κλάσης (class object). Η μέθοδος μας χρησιμοποιεί τον αναλυτή (parser) που περιγράφεται στο [213], ο οποίος ήδη χρησιμοποιήθηκε για τη μεθοδολογία του προηγούμενου κεφαλαίου και είναι αρκετά σταθερός σε διαφορετικά σενάρια. Ο αναλυτής αρχικά λαμβάνει το AST (που προκύπτει από το μεταγλωττιστή του Eclipse) και εξάγει μια ακολουθία εντολών, την οποία διασχίζει δύο φορές. Στο πρώτο πέρασμα εξάγονται όλες οι δηλώσεις του κώδικα (κλάσεις, πεδία, μέθοδοι και μεταβλητές) και δημιουργείται έναν πίνακα αναζήτησης (lookup table). Ο πίνακας αναζήτησης για το τμήμα κώδικα του Σχήματος 8.1 φαίνεται στον Πίνακα 8.1.

Πίνακας 8.1: Παράδειγμα Πίνακα Αναζήτησης για το Snippet του Σχήματος 8.1

Μεταβλητή	Τύπος
frame	JFrame
panel	JPanel
pane	Container
layout	GridLayout

Μετά την εξαγωγή των τύπων, στο δεύτερο πέρασμα δημιουργείται μια ακολουθία εντολών για το τμήμα κώδικα. Για παράδειγμα, η εντολή “pane = frame.getContentPane()” αποτελεί ένα στοιχείο FC\_getContentPane. Στη συνέχεια, επεξεργαζόμαστε περαιτέρω αυτό το στοιχείο αντικαθιστώντας το όνομα της μεταβλητής ή της συνάρτησης (σε αυτήν την περίπτωση getContentPane) με τον τύπο της (σε αυτήν την περίπτωση Container), χρησιμοποιώντας τον πίνακα αναζήτησης όπου αυτό είναι απαραίτητο. Αν δε μπορεί να προσδιοριστεί κάποιος τύπος δεδομένων, τότε τίθεται ο τύπος void. Η ακολουθία για το snippet του Σχήματος 8.1 φαίνεται στο Σχήμα 8.2. Στο ευρετήριο του Elasticsearch, η αναπαράσταση αυτή αποθηκεύεται ως μια διατεταγμένη λίστα (ordered list) στο πεδίο “Snippets”.



---

```

CI_JFrame
CI_JPanel
FC_Container
CI_GridLayout
FC_void
FC_void
FC_void
FC_void
FC_void
FC_void
FC_void

```

---

Σχήμα 8.2: Παράδειγμα ακολουθίας για το snippet του Σχήματος 8.1

## 8.3 Μεθοδολογία Αντιστοίχισης

Ένα ερώτημα στο ευρετήριο μπορεί να περιεχθεί ένα ή περισσότερα από τα πεδία “Title”, “Tags”, “Body” και “Snippets”. Κατά την αναζήτηση μιας ανάρτησης-ερώτησης, υπολογίζεται μια βαθμολογία που είναι ο μέσος όρος μεταξύ των βαθμολογιών των επιμέρους πεδίων. Η βαθμολογία για κάθε πεδίο κανονικοποιείται στο εύρος, έτσι ώστε όλα τα πεδία να προσμετρώνται με την ίδια βαρύτητα. Ο τρόπος υπολογισμού της βαθμολογίας για κάθε πεδίο αναλύεται στις ακόλουθες ενότητες.

### 8.3.1 Αντιστοίχιση Κειμένου

Η ομοιότητα μεταξύ δύο κειμένων, είτε τίτλων (titles) είτε σωμάτων (bodies), υπολογίζεται χρησιμοποιώντας το *tf-idf* (*term frequency-inverse document frequency*). Τα κείμενα/έγγραφα είναι ήδη διαχωρισμένα σε όρους (tokens) από το Elasticsearch. Εφαρμόζοντας το μοντέλο διανυσματικού χώρου (vector space model), κάθε έγγραφο-κείμενο αποτελεί ένα διάνυσμα και κάθε όρος αποτελεί μια διάσταση (dimension) του μοντέλου. Έτσι, η συχνότητα ενός όρου σε ένα έγγραφο (term frequency - tf) υπολογίζεται ως η τετραγωνική ρίζα του πλήθους των φορών που ο όρος εμφανίζεται στο έγγραφο, ενώ η αντίστροφη συχνότητα εγγράφων (inverse document frequency - idf) είναι ο λογάριθμος του πηλίκου του συνολικού πλήθους των εγγράφων με το συνολικό πλήθος των εγγράφων που περιέχουν τον όρο. Η τελική βαθμολογία μεταξύ δύο εγγράφων υπολογίζεται από την ομοιότητα συνημιτόνου μεταξύ των διανυσμάτων τους:

$$score(d_1, d_2) = \frac{d_1 \cdot d_2}{|d_1| \cdot |d_2|} = \frac{\sum_1^N w_{t_i, d_1} \cdot w_{t_i, d_2}}{\sum_1^N w_{t_i, d_1}^2 \cdot \sum_1^N w_{t_i, d_2}^2} \quad (8.1)$$

όπου  $d_1, d_2$  είναι τα δύο έγγραφα και  $w_{t_i, d_j}$  είναι η τιμή tf-idf του όρου  $t_i$  στο έγγραφο  $d_j$ .

### 8.3.2 Αντιστοίχιση Ετικετών

Καθώς οι τιμές του πεδίου “Tags” είναι μοναδικές (unique), μπορούμε να υπολογίσουμε την ομοιότητα μεταξύ δύο πεδίων “Tags” θεωρώντας ότι οι λίστες είναι σύνολα (sets). Επομένως, ορίζουμε την ομοιότητα χρησιμοποιώντας το δείκτη *Jaccard* (*Jaccard index*) μεταξύ

των συνόλων. Για δύο σύνολα  $T_1$  και  $T_2$ , ο δείκτης Jaccard ορίζεται ως το πηλίκο του μεγέθους της τομής (intersection) τους με το μέγεθος της ένωσης (union) τους:

$$J(T_1, T_2) = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} \quad (8.2)$$

Τέλος, σημειώστε ότι αφαιρούμε την ετικέτα “java” από τα δεδομένα μας, καθώς δεν είναι χρήσιμη για τη βαθμολόγηση.

### 8.3.3 Αντιστοίχιση Snippets

Καθώς έχουμε εξάγει ακολουθίες από τα snippets (βλέπε ενότητα 8.2.2), μπορούμε να ορίσουμε μια μετρική ομοιότητας για ακολουθίες. Σχεδιάζουμε τη μετρική μας με βάση τη *Μέγιστη Κοινή Υπακολουθία (Longest Common Subsequence - LCS)*. Για δύο ακολουθίες  $S_1$  και  $S_2$ , η LCS τους ορίζεται ως η μέγιστη υπακολουθία που είναι κοινή και στις δύο ακολουθίες. Μια υπακολουθία, ωστόσο, δεν είναι απαραίτητο να περιέχει διαδοχικά στοιχεία κάποιας ακολουθίας. Για παράδειγμα, η μέγιστη κοινή υπακολουθία για τις ακολουθίες  $S_1 = [A, B, D, C, B, D, C]$  και  $S_2 = [B, D, C, A, B, C]$  είναι η  $LCS(S_1, S_2) = [B, D, C, B, C]$ . Το πρόβλημα εύρεσης της μέγιστης κοινής υπακολουθίας για δύο ακολουθίες  $S_1$  και  $S_2$  μπορεί να λυθεί χρησιμοποιώντας δυναμικό προγραμματισμό (dynamic programming). Η υπολογιστική πολυπλοκότητα (computational complexity) της λύσης είναι  $O(m \times n)$ , όπου  $m$  και  $n$  είναι τα μήκη των δύο ακολουθιών [177], οπότε ο αλγόριθμος είναι αρκετά γρήγορος.

Έχοντας ορίσει την LCS μεταξύ δύο ακολουθιών snippets  $S_1$  και  $S_2$ , η τελική βαθμολογία για την ομοιότητα μεταξύ τους υπολογίζεται από τη σχέση:

$$score(S_1, S_2) = 2 \cdot \frac{|LCS(S_1, S_2)|}{|S_1| + |S_2|} \quad (8.3)$$

Καθώς το μήκος της LCS είναι πάντοτε μικρότερο από το μήκος της μικρότερης ακολουθίας, το πηλίκο της LCS με το άθροισμα των μηκών των ακολουθιών είναι στο εύρος  $[0, 0.5]$ . Έτσι, η τιμή της παραπάνω εξίσωσης είναι στο εύρος  $[0, 1]$ .

Για παράδειγμα, ας υπολογίσουμε την τιμή ομοιότητας μεταξύ της ακολουθίας του Σχήματος 8.2 και της ακολουθίας του Σχήματος 8.3. Το μήκος της LCS μεταξύ αυτών των ακολουθιών είναι 5, ενώ τα μήκη των δύο ακολουθιών είναι 10 και 6. Επομένως, χρησιμοποιώντας την εξίσωση (8.3), η βαθμολογία μεταξύ των δύο ακολουθιών είναι 0.625.

---

```

CI_JFrame
FC_Container
AM_float
CI_GridLayout
FC_void
FC_void

```

---

Σχήμα 8.3: Παράδειγμα ακολουθίας που εξήχθη από ένα snippet

## 8.4 Αξιολόγηση

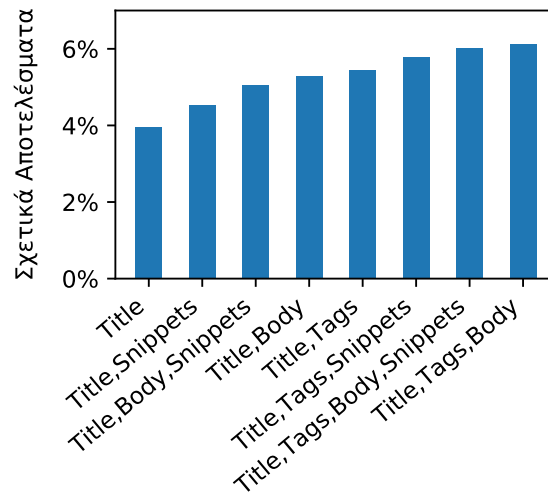
Η αξιολόγηση της ομοιότητας των αναρτήσεων του Stack Overflow δεν είναι απλή διαδικασία, καθώς τα δεδομένα δεν είναι σημειωμένα (annotated) για αυτού του είδους την ανάλυση. Ως μια ένδειξη ότι δύο ερωτήσεις είναι παρόμοιες, ελέγχουμε για την παρουσία μιας σύνδεσης (link) μεταξύ τους (που επίσης δίνεται από το [224]). Αν και αυτή η υπόθεση είναι λογική, η αντίθετη υπόθεση, ότι δηλαδή οι μη συνδεδεμένες ερωτήσεις δεν είναι παρόμοιες, δεν ισχύει απαραίτητα. Στην πράξη, το dump δεδομένων του Stack Overflow έχει περίπου 700000 αναρτήσεις για ερωτήσεις σχετικές με java, από τις οποίες κοντά στις 300000 έχουν snippets από τα οποία προκύπτουν ακολουθίες (δηλαδή snippets με τουλάχιστον μια ανάθεση, κλήση συνάρτησης ή δημιουργία αντικειμένου). Κατά μέσο όρο, κάθε μία από αυτές τις 300000 αναρτήσεις έχει 0.98 συνδέσεις. Στην περίπτωση μας, ωστόσο, το πρόβλημα διαμορφώνεται ως πρόβλημα αναζήτησης, οπότε η βασική πρόκληση είναι να βρούμε συνδεδεμένες (και άρα παρόμοιες) ερωτήσεις.

Στην αξιολόγηση μας, έχουμε συμπεριλάβει ερωτήσεις με τουλάχιστον 5 συνδέσεις (καθώς και διπλότυπες ερωτήσεις που επίσης αποτελούν συνδέσεις σύμφωνα με το σχήμα δεδομένων του Stack Overflow)<sup>2</sup>. Για λόγους απόδοσης, όλα τα ερωτήματα εκτελούνται αρχικά πάνω στον τίτλο της ανάρτησης, ενώ στη συνέχεια τα πρώτα 1000 αποτελέσματα αναζητούνται και πάλι χρησιμοποιώντας τόσο τον τίτλο, όσο και οποιαδήποτε άλλα πεδία. Για τα προαναφερθέντα, κατασκευάζουμε ερωτήματα Elasticsearch τύπου *bool*. Ορίζουμε 8 συνδυασμούς πεδίων για την αξιολόγηση της μεθοδολογίας μας. Καθώς ο τίτλος (title) παρέχεται πάντα, οι συνδυασμοί είναι όλοι οι συνδυασμοί ετικετών (tag), σωμάτων (body) και τμημάτων κώδικα (snippets). Για κάθε ερώτηση-ανάρτηση διατηρούμε τα πρώτα 20 αποτελέσματα, υποθέτοντας ότι αυτό είναι το μέγιστο πλήθος αναρτήσεων που θα εξέταζε ένας προγραμματιστής (η χρήση άλλων τιμών είχε παρόμοια αποτελέσματα).

Το διάγραμμα του Σχήματος 8.4 απεικονίζει το μέσο ποσοστό των σχετικών αποτελεσμάτων (σε σχέση με το πλήθος των σχετικών συνδέσεων κάθε ερώτησης) μέσα στα πρώτα 20 αποτελέσματα κάθε ερωτήματος, ανεξάρτητα από την κατάταξή τους. Τα αποτελέσματα αυτά περιλαμβάνουν όλες τις ερωτήσεις του συνόλου δεδομένων που έχουν τμήματα κώδικα (περίπου 300000), ακόμα κι αν η ακολουθία εντολών που εξήχθη από αυτά έχει μήκος 1 (για μήκος 0 δε μπορεί να υπάρξει κάποια σύγκριση). Παρατηρούμε ότι το ποσοστό σχετικών αποτελεσμάτων για όλους τους συνδυασμούς είναι μικρότερο από 10%· αυτό, ωστόσο, είναι ένας περιορισμός του συνόλου δεδομένων. Πολλές περισσότερες από αυτές τις αναρτήσεις μπορεί να είναι σχετικές, αλλά να μην συνδέονται.

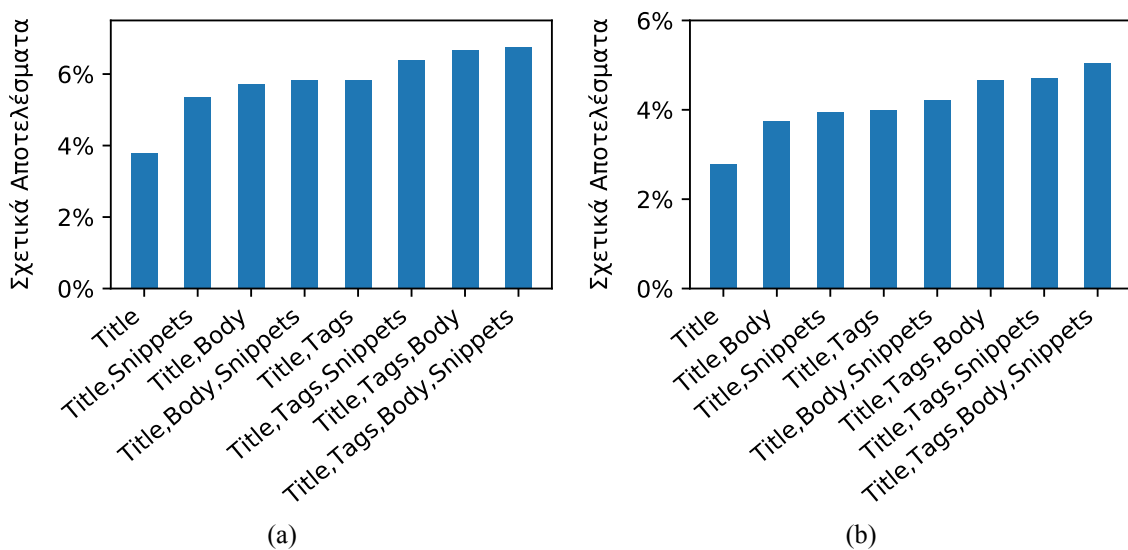
Όπως φαίνεται στο Σχήμα 8.4, όταν τα snippets χρησιμοποιούνται μαζί με τίτλους και/ή ετικέτες, οι ερωτήσεις που επιστρέφονται είναι πιο σχετικές. Ωστόσο, όταν χρησιμοποιείται το σώμα της ερώτησης, η χρήση των snippets είναι πιθανώς περιττή (ή ακόμα και μη προτιμητέα). Αυτό είναι αναμενόμενο καθώς το διάγραμμα του Σχήματος 8.4 περιλαμβάνει πολλά μικρά snippets που έχουν ακολουθίες που είναι δύσκολα συγκρίσιμες. Σε τέτοιες περιπτώσεις, για την εύρεση σχετικών ερωτήσεων είναι προτιμότερη η χρήση περισσότερων (ή μεγαλύτερων) τμημάτων κώδικα ή η προσθήκη περισσότερου κειμένου στο σώμα της ερώτησης.

<sup>2</sup>Η μεθοδολογία μας υποστηρίζει όλες τις ερωτήσεις, ανεξάρτητα από τον αριθμό των συνδέσεών τους. Ωστόσο, στο πλαίσιο της αξιολόγησης, υποθέτουμε ότι οι ερωτήσεις με λιγότερες συνδέσεις μπορεί να είναι πολύ περιορισμένες και/ή να μην έχουν παρόμοιες ερωτήσεις.



Σχήμα 8.4: Ποσοστό σχετικών αποτελεσμάτων μέσα στα πρώτα 20 αποτελέσματα κάθε ερωτήματος, για ερωτήσεις με πλήθος εντολών κώδικα μεγαλύτερο ή ίσο του 1

Για να επιβεβαιώσουμε την εγκυρότητα του παραπάνω ισχυρισμού και να αναλύσουμε περαιτέρω την αποτελεσματικότητα της προσέγγισής μας, πραγματοποιήσαμε δύο ακόμη πειράματα. Τα αποτελέσματά τους φαίνονται στα Σχήματα 8.5a και 8.5b. Όπως και στο Σχήμα 8.4, τα διαγράμματα απεικονίζουν το ποσοστό των σχετικών αποτελεσμάτων μέσα στα πρώτα 20 αποτελέσματα κάθε ερωτήματος, αυτήν τη φορά όμως με snippets που έχουν πλήθος εντολών ίσο ή μεγαλύτερο του 3 ή του 5, για τα Σχήματα 8.5a και 8.5b, αντίστοιχα. Το πλήθος των αναρτήσεων για τις δύο αυτές περιπτώσεις είναι περίπου 200000 και 150000, αντίστοιχα. Η αποτελεσματικότητα της χρήσης snippets για την εύρεση παρόμοιων ερωτήσεων είναι αρκετά ξεκάθαρη από αυτά τα δύο διαγράμματα.



Σχήμα 8.5: Ποσοστό σχετικών αποτελεσμάτων μέσα στα πρώτα 20 αποτελέσματα κάθε ερωτήματος (ανάρτησης), για ερωτήσεις με πλήθος εντολών κώδικα μεγαλύτερο ή ίσο (a) του 3, και (b) του 5

Όπως αναμένεται, η χρήση όλων των διαθέσιμων πηγών πληροφορίας, δηλαδή του τίτλου της ερώτησης, των ετικετών, του σώματος και των τμημάτων κώδικα, είναι βέλτιστη και στις δύο περιπτώσεις. Η μεθοδολογία αντιστοίχισης τμημάτων κώδικα που κατασκευάσαμε είναι αρκετά αποτελεσματική για την εύρεση σχετικών αναρτήσεων, ακόμα και όταν δε χρησιμοποιείται το σώμα της ανάρτησης στο ερώτημα.

Επιπλέον, όταν τα snippets που παρέχονται είναι μεγαλύτερα και άρα προκύπτουν μεγαλύτερες ακολουθίες εντολών, όπως στο Σχήμα 8.5b, τότε είναι πιο αποτελεσματικό να χρησιμοποιήσουμε μόνο αυτά αντί για ολόκληρο το σώμα της ανάρτησης. Ένα άλλο βασικό στοιχείο που φαίνεται να επηρεάζει τη σχετικότητα των αποτελεσμάτων και στα τρία σενάρια αξιολόγησης είναι η χρήση ετικετών. Παρατηρούμε ότι η χρήση ετικετών μαζί με τμήματα κώδικα είναι σχεδόν εξίσου αποτελεσματική με την κατάρτιση ολόκληρου του σώματος της ερώτησης. Έτσι, ο προγραμματιστής θα μπορούσε να γράψει έναν τίτλο και κάποιες ετικέτες για την ερώτησή του και στη συνέχεια να στείλει ένα σχετικό τμήμα του κώδικά του χωρίς να χρειάζεται να διατυπώσει μια πλήρη ερώτηση.

## 8.5 Συμπεράσματα

Σε αυτό το κεφάλαιο, διερευνήσαμε το πρόβλημα της εύρεσης σχετικών ερωτήσεων στο Stack Overflow. Η μεθοδολογία ομοιότητας αναρτήσεων που σχεδιάσαμε χρησιμοποιεί όχι μόνο τον τίτλο, τις ετικέτες και το σώμα μιας ερώτησης, αλλά και τα τμήματα πηγαίου κώδικα (snippets). Τα αποτελέσματα της αξιολόγησής μας υποδηλώνουν ότι τα τμήματα κώδικα αποτελούν πολύτιμη πηγή πληροφοριών για να βρει κανείς συνδεδεμένες ερωτήσεις ή να προσδιορίσει αν μια ερώτηση έχει ήδη αναρτηθεί. Ως εκ τούτου, χρησιμοποιώντας το σύστημά μας, ο προγραμματιστής μπορεί εύκολα να σχηματίσει μια ανάρτηση-ερώτηση ακόμη και χρησιμοποιώντας μόνο ένα υπάρχον snippet ώστε να ελέγξει τον κώδικά του ή να επιβεβαιώσει ότι ο κώδικας που θέλει να επαναχρησιμοποιήσει συμβαδίζει με αυτά που προτείνονται από την κοινότητα.

Αν και η αντιστοίχιση τμημάτων κώδικα είναι ένα δύσκολο έργο, πιστεύουμε ότι η μεθοδολογία μας είναι ένα βήμα προς τη σωστή κατεύθυνση. Η μελλοντική έρευνα σε αυτό τον άξονα περιλαμβάνει την προσαρμογή της μεθοδολογίας μας στα ειδικά χαρακτηριστικά των διαφόρων τμημάτων κώδικα, π.χ. ομαδοποιώντας τα ανάλογα με το μέγεθος ή την πολυπλοκότητά τους. Επιπλέον, μπορούν να δοκιμαστούν διαφορετικές αναπαραστάσεις για τμήματα κώδικα προκειμένου να βελτιωθεί περαιτέρω η μεθοδολογία αντιστοίχισης.



**Μέρος**

**IV**

**ΑΞΙΟΛΟΓΗΣΗ ΠΟΙΟΤΗΤΑΣ**





# 9

## Προτάσεις Επαναχρησιμοποιήσιμου Κώδικα

### 9.1 Επισκόπηση

Στα Κεφάλαια 5 και 6 εστιάσαμε στο πρόβλημα της επαναχρησιμοποίησης τμημάτων λογισμικού και προσεγγίσαμε το πρόβλημα από την άποψη της λειτουργικότητας. Όπως ήδη αναφέρθηκε στη σχετική βιβλιογραφία στο Κεφαλαίο 6, οι περισσότερες ερευνητικές προσπάθειες χρησιμοποιούν μηχανισμούς αντιστοίχισης (matching mechanisms) για να παρέχουν λειτουργικά τμήματα λογισμικού στον προγραμματιστή. Κάποια συστήματα χρησιμοποιούν επίσης περιπτώσεις ελέγχου (test cases) για να εξασφαλίσουν ότι καλύπτεται η επιθυμητή λειτουργικότητα [96, 98], ενώ άλλα συστήματα εφαρμόζουν και μετασχηματισμούς κώδικα (code transformations) για την ενσωμάτωση των τμημάτων στον κώδικα του προγραμματιστή.

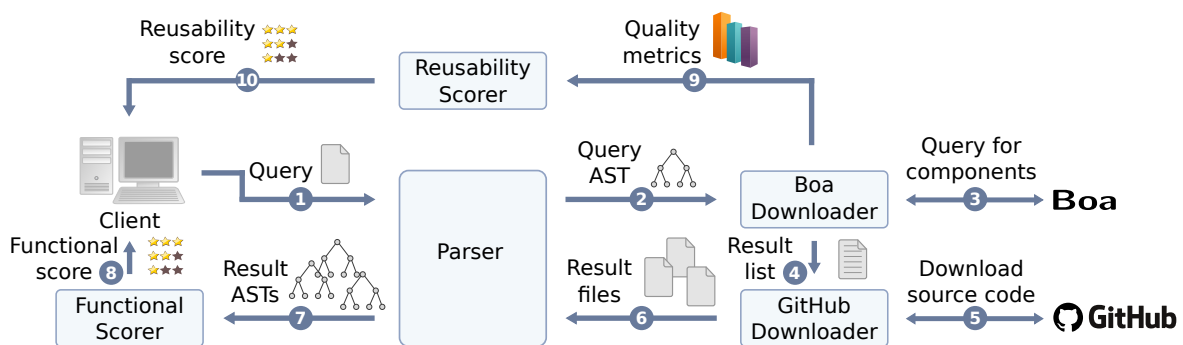
Παρόλο που αυτά τα συστήματα καλύπτουν τα λειτουργικά κριτήρια που τίθενται από τον προγραμματιστή, δεν παρέχουν καμία διαβεβαίωση σχετικά με την επαναχρησιμοποιησιμότητα (reusability) του πηγαίου κώδικα. Η επαναχρησιμοποίηση κώδικα που ανακτήθηκε από online πηγές μπορεί να είναι εσφαλμένη πρακτική, ιδιαίτερα αν ο κώδικας δεν ελεγχθεί από κάποιον ειδικό σε θέματα ποιότητας (quality expert). Μια πιθανή λύση που αξιοποιεί τη δύναμη της κοινότητας των προγραμματιστών προτείνεται από τον Bing Developer Assistant [104], που προάγει τα τμήματα κώδικα τα οποία έχουν επιλεγεί από άλλους προγραμματιστές. Ωστόσο, οι λύσεις που βασίζονται σε crowdsourcing συχνά δεν είναι ακριβείς, καθώς οι προγραμματιστές μπορεί να χρειάζονται εξειδικευμένα τμήματα που να πληρούν συγκεκριμένα κριτήρια ποιότητας. Ο Code Conjuror [96] κινείται προς αυτήν την κατεύθυνση, επιλέγοντας κάθε φορά το λιγότερο πολύπλοκο τμήμα κώδικα όταν κάποια τμήματα είναι λειτουργικά ισοδύναμα, όπου η πολυπλοκότητα καθορίζεται από το πλήθος των γραμμών κώδικα (lines of code). Ωστόσο, ο καθορισμός των κριτηρίων για την πολυπλοκότητα και γενικά για την επαναχρησιμοποιησιμότητα ενός τμήματος δεν είναι απλή διαδικασία.

Σε αυτό το κεφάλαιο, παρουσιάζουμε ένα σύστημα προτάσεων τμημάτων κώδικα που καλύπτει όχι μόνο τα λειτουργικά κριτήρια, αλλά και τα κριτήρια επαναχρησιμοποιησιμότητας κώδικα. Το σύστημά μας λέγεται QualBoa [148] και αξιοποιεί την υπηρεσία ευρετηρίου Boa [229] για τον εντοπισμό χρήσιμων τμημάτων λογισμικού και τον υπολογισμό μετρικών ποιότητας (quality metrics). Το QualBoa κατεβάζει τμήματα κώδικα από το GitHub, και στη συνέχεια παράγει για κάθε τμήμα μια βαθμολογία για τη λειτουργικότητά του (functional score), καθώς και μια βαθμολογία για την επαναχρησιμοποιησιμότητά του (reusability score). Επιπλέον, ο δείκτης επαναχρησιμοποιησιμότητας είναι παραμετροποιήσιμος (configurable) ώστε να είναι δυνατή η διαμόρφωσή του από ειδικούς ποιότητας.

## 9.2 Το Σύστημα Προτάσεων Επαναχρησιμοποιήσιμου Κώδικα QualBoa

### 9.2.1 Επισκόπηση του QualBoa

Η αρχιτεκτονική του QualBoa φαίνεται στο Σχήμα 9.1. Σημειώνουμε ότι η αρχιτεκτονική αυτή μπορεί να υποστηρίξει οποιαδήποτε γλώσσα προγραμματισμού και να συνδεθεί με διάφορες υπηρεσίες φιλοξενίας κώδικα (code hosting services). Στην περίπτωση αυτή, χρησιμοποιούμε τη Boa και το GitHub ως υπηρεσίες φιλοξενίας κώδικα, και υλοποιούμε τα διάφορα τμήματα για τη γλώσσα προγραμματισμού Java.



Σχήμα 9.1: Αρχιτεκτονική του QualBoa

Αρχικά, ο προγραμματιστής παρέχει ένα ερώτημα σε μορφή υπογραφής (*signature*). Όπως και στο σύστημα Mantissa (βλέπε Κεφάλαιο 6), μια υπογραφή είναι παρόμοια με μια διεπαφή (interface) και περιέχει όλες τις μεθόδους του επιθυμητού τμήματος (κλάση Java). Ένα παράδειγμα υπογραφής για μια δομή στοίβας (“Stack”) φαίνεται στο Σχήμα 9.2.

```
public class Stack{
    public void push(Object element);
    public Object pop();
}
```

Σχήμα 9.2: Παράδειγμα υπογραφής για μια κλάση “Stack” με μεθόδους “push” και “pop”

Η υπογραφή δίνεται ως είσοδος στον *Parser*. Ο *Parser* χρησιμοποιεί το Eclipse JDT<sup>1</sup> για να εξάγει το *Αφηρημένο Συντακτικό Δένδρο* (*Abstract Syntax Tree - AST*) από αρχεία Java. Μετά την εξαγωγή του AST της υπογραφής από τον *Parser*, ο *Boa Downloader* κατασκευάζει ένα ερώτημα *Boa* για σχετικά τμήματα κώδικα, το οποίο επιπλέον περιέχει υπολογισμούς για μετρικές ποιότητας. Το ερώτημα στέλνεται στη *Boa* [229], και η απάντηση που επιστρέφεται είναι μια λίστα από αρχεία Java καθώς και οι τιμές των μετρικών ποιότητας για αυτά τα αρχεία. Η λίστα των αποτελεσμάτων δίνεται στον *GitHub Downloader*, που κατεβάζει τα αρχεία από το GitHub. Στη συνέχεια, ο *Parser* λαμβάνει τα αρχεία και εξάγει το AST από κάθε αρχείο.

Τα ASTs των αρχείων τα επεξεργάζεται ο *Functional Scorer*, που παράγει μια *βαθμολογία λειτουργικότητας* (*functional score*) για κάθε αποτέλεσμα. Η λειτουργική βαθμολογία ενός αποτελέσματος δηλώνει κατά πόσο το αρχείο καλύπτει τα κριτήρια που τίθενται από το ερώτημα του προγραμματιστή. Ο *Reusability Scorer* λαμβάνει ως είσοδο τις μετρικές για τα αρχεία αποτελεσμάτων και παράγει μια *βαθμολογία επαναχρησιμοποιησιμότητας* (*reusability score*), που δηλώνει το κατά πόσο κάθε αρχείο είναι επαναχρησιμοποιήσιμο. Τελικά, στον προγραμματιστή (*Client*) παρέχονται οι βαθμολογίες για όλα τα αρχεία, ενώ τα αποτελέσματα κατατάσσονται σύμφωνα με τη βαθμολογία λειτουργικότητας κάθε αρχείου. Οι διαδικασίες της λήψης τμημάτων κώδικα και της κατασκευής των βαθμολογιών λειτουργικότητας και επαναχρησιμοποιησιμότητας περιγράφονται στις επόμενες ενότητες.

## 9.2.2 Λήψη Πηγαίου Κώδικα και Μετρικών

Το *QualBoa* αρχικά εξάγει τα στοιχεία του ερωτήματος, που περιλαμβάνουν το όνομα κλάσης, τα ονόματα και τους τύπους επιστροφής των μεθόδων, και στη συνέχεια βρίσκει χρήσιμα τμήματα και υπολογίζει τις τιμές μετρικών τους χρησιμοποιώντας τη *Boa* [229]. Μια απλοποιημένη εκδοχή του ερωτήματος φαίνεται στο Σχήμα 9.3.

Το ερώτημα ακολουθεί το πρότυπο επισκέπτη (*visitor pattern*) για να διασχίσει τα ASTs των αρχείων Java στη *Boa*. Στο πρώτο μέρος του ερωτήματος διασχίζονται οι δηλώσεις τύπων (*type declarations*) όλων των αρχείων και προσδιορίζεται αν αντιστοιχίζονται με το όνομα κλάσης του ερωτήματος (*class\_name*). Στη συνέχεια, οι μέθοδοι των αντιστοιχισμένων αρχείων ελέγχονται για το αν τα ονόματα και οι τύποι τους ταιριάζουν με αυτά του ερωτήματος (*method\_names* και *method\_types*). Ο αριθμός των αντιστοιχισμένων στοιχείων συναθροίζεται για την κατάταξη των αποτελεσμάτων και μια λίστα με τα πρώτα 150 αποτελέσματα της κατάταξης δίνεται στον *GitHub Downloader*, που κατεβάζει τα αρχεία Java από το *GitHub API*.

Στο δεύτερο μέρος του ερωτήματος υπολογίζονται μετρικές πηγαίου κώδικα (*source code metrics*) για τα ανακτηθέντα αρχεία. Στο Σχήμα 9.3 φαίνεται ο κώδικας για τον υπολογισμό του πλήθους των δημόσιων πεδίων (*number of public fields*). Το συνολικό ερώτημα υπολογίζει τις μετρικές που φαίνονται στον Πίνακα 9.1 (η επιλογή αυτών των μετρικών αναλύεται στην ενότητα 9.2.4).

---

<sup>1</sup><http://www.eclipse.org/jdt/>

---

```

visit(p, visitor {
  ...
  before node: Declaration -> {
    if (match(class_name, node.name)) {
      foreach (i: int; node.methods[i])
        visit(node.methods[i]);
    }
  }
  before node: Method -> {
    for (i := 0; i < len(method_names); i++) {
      if (match(method_names[i], node.name)) {
        match_names[i] = true;
        if (method_types[i] == node.return_type.name)
          match_types[i] = true;
      }
    }
  }
  ...
  after node: Declaration -> {
    foreach (i: int; def(node.fields[i])) {
      foreach (j: int; def(node.fields[i].modifiers[j])) {
        if (node.fields[i].modifiers[j].visibility == Visibility.PUBLIC)
          num_public_fields++;
      }
    }
    ...
  }
});

```

---

Σχήμα 9.3: Ερώτημα Boa για τμήματα κώδικα και μετρικές

### 9.2.3 Εξόρυξη Τμημάτων Κώδικα

Ο Functional Scorer υπολογίζει την ομοιότητα μεταξύ των ASTs των αποτελεσμάτων με το AST του ερωτήματος, έτσι ώστε να κατατάξει τα αποτελέσματα σύμφωνα με την επιθυμητή λειτουργικότητα (όπως ορίζεται από το ερώτημα του προγραμματιστή). Αρχικά, το ερώτημα καθώς και κάθε αρχείο αποτελέσματος αναπαρίστανται ως λίστες. Η λίστα για ένα αρχείο αποτελέσματος ορίζεται από την εξίσωση:

$$result = [name, method_1, method_2, \dots, method_n] \quad (9.1)$$

όπου  $name$  είναι το όνομα της κλάσης και  $method_i$  είναι μια υπολίστα για την  $i$ -η μέθοδο της κλάσης, από τις συνολικά  $n$  μεθόδους. Η υπολίστα μιας μεθόδου ορίζεται ως:

$$method = [name, type, param_1, param_2, \dots, param_m] \quad (9.2)$$

Πίνακας 9.1: Μετρικές Επαναχρησιμοποιησιμότητας του QualBoa

Μετρική	Περιγραφή
Average Lines of Code per Method	Μέσο πλήθος γραμμών κώδικα για κάθε μέθοδο
Average Cyclomatic Complexity	Μέση πολυπλοκότητα McCabe για όλες τις μεθόδους
Coupling Between Objects	Αριθμός κλάσεων από τις οποίες εξαρτάται η υπό εξέταση κλάση
Lack of Cohesion in Methods	Πλήθος των ζευγών μεθόδων με κοινές μεταβλητές μεταξύ τους αφαιρούμενο από το πλήθος των ζευγών μεθόδων χωρίς κοινές μεταβλητές μεταξύ τους
Average Block Depth	Μέσο βάθος ροής ελέγχου για όλες τις μεθόδους
Efferent Couplings	Αριθμός τύπων δεδομένων για την κλάση
Number of Public Fields	Πλήθος δημόσιων πεδίων της κλάσης
Number of Public Methods	Πλήθος δημόσιων μεθόδων της κλάσης

όπου *name* είναι το όνομα της μεθόδου, *type* είναι ο τύπος επιστροφής της, και *param<sub>j</sub>* είναι ο τύπος επιστροφής της *j*-ης παραμέτρου, από τις *m* παραμέτρους συνολικά. Χρησιμοποιώντας τις εξισώσεις (9.1) και (9.2), μπορούμε να αναπαραστήσουμε κάθε αποτέλεσμα, όπως επίσης και το ερώτημα, σε μια δομή εμφωλευμένων λιστών (nested list structure).

Για τη σύγκριση ενός ερωτήματος με ένα αποτέλεσμα χρειάζεται ο υπολογισμός μιας τιμής ομοιότητας (similarity score) μεταξύ των δύο αντίστοιχων λιστών, που με τη σειρά του απαιτεί τον υπολογισμό της βαθμολογίας μεταξύ κάθε ζεύγους μεθόδων. Για τον υπολογισμό της μέγιστης τιμής ομοιότητας μεταξύ δυο λιστών από μεθόδους χρειάζεται να βρούμε τη βαθμολογία για κάθε ζεύγος μεθόδων και να επιλέξουμε το συνδυασμό με τη μεγαλύτερη συνολική βαθμολογία. Ακολουθώντας τη λογική του προβλήματος *σταθερού γάμου (stable marriage)* (βλέπε Κεφάλαιο 6), κατατάσσουμε τα ζεύγη σύμφωνα με την τιμή ομοιότητάς τους σε φθίνουσα σειρά και στη συνέχεια επιλέγουμε κάθε φορά ένα ζεύγος από την κορυφή, ελέγχοντας αν κάποια μέθοδος έχει ήδη αντιστοιχιστεί. Η ίδια λογική εφαρμόζεται και για τη βέλτιστη αντιστοίχιση των παραμέτρων μεταξύ δύο λιστών μεθόδων.

Τα ονόματα κλάσεων, τα ονόματα και οι τύποι μεθόδων, καθώς και οι τύποι παραμέτρων αντιστοιχίζονται συγκρίνοντας τα σύνολα από του όρους τους (token sets). Καθώς στη Java τα ονόματα είναι σε camelCase, αρχικά διαχωρίζουμε κάθε συμβολοσειρά σε όρους (tokens). Στη συνέχεια, χρησιμοποιούμε το δείκτη Jaccard (Jaccard index) για να συγκρίνουμε δύο σύνολα από όρους. Ο δείκτης Jaccard για δύο σύνολα ορίζεται ως το πηλίκο του μεγέθους της τομής (intersection) τους με το μέγεθος της ένωσής (union) τους. Τέλος, η ομοιότητα μεταξύ δύο λιστών ή διανυσμάτων  $\vec{A}$  και  $\vec{B}$  (είτε σε επίπεδο μεθόδων είτε σε επίπεδο κλάσεων/αποτελεσμάτων) υπολογίζεται χρησιμοποιώντας το συντελεστή Tanimoto (Tanimoto coefficient) των διανυσμάτων  $\vec{A} \cdot \vec{B} / (|\vec{A}|^2 + |\vec{B}|^2 - \vec{A} \cdot \vec{B})$ , όπου τα  $|\vec{A}|$  και  $|\vec{B}|$  είναι τα μήκη των διανυσμάτων  $\vec{A}$  και  $\vec{B}$  αντίστοιχα, ενώ το  $\vec{A} \cdot \vec{B}$  είναι το εσωτερικό τους γινόμενο.

Ας θεωρήσουμε, π.χ., τη λίστα ερωτήματος  $[Stack, [push, void, Object], [pop, Object]]$  και τη λίστα αποτελέσματος  $[IntStack, [pushObject, void, int], [popObject, int]]$ . Η ομοιότητα μεταξύ των ζευγών μεθόδων για τη λειτουργία εισαγωγής στοιχείου (“push”) στη στοίβα

υπολογίζεται ως:

$$\begin{aligned}
 score_{PUSH} &= Tanimoto([1, 1, 1], & (9.3) \\
 & \quad [Jaccard(\{push\}, \{push, object\}), \\
 & \quad \quad Jaccard(\{void\}, \{void\}) \\
 & \quad \quad Jaccard(\{object\}, \{int\})]) \\
 &= Tanimoto([1, 1, 1], [0.5, 1, 0]) \simeq 0.545
 \end{aligned}$$

όπου παρατηρούμε πως η λίστα ερωτήματος είναι πάντοτε μια λίστα με όλα τα στοιχεία ίσα με τη μονάδα, καθώς αποτελεί μια τέλεια αντιστοίχιση. Όμοια, η βαθμολογία για τη λειτουργία εξαγωγής στοιχείου (“pop”) από τη στοίβα υπολογίζεται ως:

$$\begin{aligned}
 score_{POP} &= Tanimoto([1, 1], & (9.4) \\
 & \quad [Jaccard(\{pop\}, \{pop, object\}), \\
 & \quad \quad Jaccard(\{object\}, \{int\})]) \\
 &= Tanimoto([1, 1], [0.5, 0]) \simeq 0.286
 \end{aligned}$$

Τελικά, η συνολική βαθμολογία για την κλάση υπολογίζεται ως:

$$\begin{aligned}
 score_{STACK} &= Tanimoto([1, 1, 1], & (9.5) \\
 & \quad [Jaccard(\{stack\}, \{int, stack\}), \\
 & \quad \quad score_{PUSH}, \\
 & \quad \quad score_{POP}]) \\
 &= Tanimoto([1, 1, 1], [0.5, 0.545, 0.286]) \simeq 0.579
 \end{aligned}$$

## 9.2.4 Προτάσεις Ποιοτικού Κώδικα

Μετά την αξιολόγηση της λειτουργικότητας των τμημάτων κώδικα, το QualBoa επιπλέον ελέγχει αν τα αποτελέσματα είναι κατάλληλα για επαναχρησιμοποίηση χρησιμοποιώντας μετρικές πηγαίου κώδικα. Παρόλο που το πρόβλημα της μέτρησης της επαναχρησιμοποιησιμότητας (reusability) τμημάτων κώδικα έχει μελετηθεί αρκετά [230–233], η επιλογή των κατάλληλων μετρικών δεν είναι απλή διαδικασία· οι σχετικές προσεγγίσεις χρησιμοποιούν πλήθος διαφορετικών μετρικών, όπως π.χ. τις μετρικές C&K (C&K metrics) [231], μετρικές που αφορούν τη σύζευξη (coupling) και τη συνοχή (cohesion) [232], καθώς και μετρικές που είναι σχετικές με το μέγεθος (size) και την πολυπλοκότητα (complexity) [230]. Σε κάθε περίπτωση, όμως, οι βασικοί άξονες ποιότητας που προσδιορίζουν αν ένα τμήμα κώδικα είναι επαναχρησιμοποιήσιμο είναι κοινοί. Σύμφωνα με τους ορισμούς των διάφορων χαρακτηριστικών ποιότητας [56], και σύμφωνα με τη σχετική βιβλιογραφία [231, 233, 234], η επαναχρησιμοποιησιμότητα είναι σχετική με τις έννοιες της *μηματοποιησιμότητας* (modularity), της *χρησιμότητας* (usability), της *συντηρησιμότητας* (maintainability) και της *κατανοησιμότητας* (understandability).

Σχεδιάζουμε ένα μοντέλο επαναχρησιμοποιησιμότητας, που φαίνεται στον Πίνακα 9.2 και περιλαμβάνει 8 μετρικές που είναι σχετικές με ένα ή περισσότερα από τα προαναφερθέντα χαρακτηριστικά ποιότητας. Οι μετρικές αυτές καλύπτουν διάφορες ιδιότητες του κώδικα, όπως το μέγεθος, η πολυπλοκότητα, η σύζευξη και η συνοχή. Το μοντέλο είναι σχετικά

από, σημειώνοντας την τιμή κάθε μετρικής ως *κανονική (normal)* ή *ακραία (extreme)*, σύμφωνα με τα όρια (thresholds) που φαίνονται στη δεύτερη στήλη του Πίνακα 9.2. Όταν μια τιμή είναι κανονική, τότε συνεισφέρει ένα βαθμό ποιότητας στα σχετικά χαρακτηριστικά. Η επαναχρησιμοποίησιμότητα περιέχει και τις 8 τιμές. Έτσι, για παράδειγμα, για ένα τμήμα κώδικα που έχει ακραίες τιμές σε 2 από τις 8 μετρικές, η βαθμολογία της επαναχρησιμοποίησης του θα είναι 6 στα 8.

Πίνακας 9.2: Μοντέλο Επαναχρησιμοποίησης του QualBoa

Μετρικές Ποιότητας	Όρια	Σχετικά Χαρακτηριστικά Ποιότητας				
		Τμηματοποιησιμότητα	Συντηρησιμότητα	Χρηστικότητα	Κατανοησιμότητα	Επαναχρησιμοποίησης
Average Lines of Code per Method	> 30		×		×	×
Average Cyclomatic Complexity	> 8		×		×	×
Coupling Between Objects	> 20	×		×		×
Lack of Cohesion in Methods	> 20	×				×
Average Block Depth	> 3				×	×
Efferent Couplings	> 20	×		×		×
Number of Public Fields	> 10		×	×		×
Number of Public Methods	> 30		×	×		×
#Μετρικές ανά Χαρακτηριστικό Ποιότητας:		3	4	4	3	8

Ο καθορισμός των κατάλληλων ορίων (thresholds) για μετρικές ποιότητας δεν είναι απλή διαδικασία, καθώς διαφορετικοί τύποι λογισμικού μπορεί να πρέπει να καλύψουν διαφορετικούς στόχους ποιότητας. Ως εκ τούτου, το QualBoa επιτρέπει την παραμετροποίηση (configuration) αυτών των ορίων, ενώ οι προκαθορισμένες τιμές τους τίθενται στις τιμές που επιτάσσει το σύγχρονο state-of-the-practice, όπως αυτό ορίζεται από τη σχετική βιβλιογραφία και εφαρμόζεται από τα ευρέως χρησιμοποιούμενα εργαλεία στατικής ανάλυσης, όπως το PMD<sup>2</sup> ή το CodePro AnalytiX<sup>3</sup>.

### 9.3 Αξιολόγηση

Αξιολογούμε το QualBoa στο σύνολο ερωτημάτων του Πίνακα 9.3, που περιέχει 7 ερωτήματα για διαφορετικά τμήματα κώδικα (components). Αυτό το σύνολο ερωτημάτων είναι παρόμοιο με αυτό που χρησιμοποιήθηκε στο Κεφάλαιο 6 για την αξιολόγηση της Mantissa

<sup>2</sup><https://pmd.github.io/>

<sup>3</sup><https://developers.google.com/java-dev-tools/codepro/>

και του Code Conjuror [96]. Σημειώστε, ωστόσο, ότι αυτή τη φορά το σύστημά μας είναι συνδεδεμένο με την Boa, οπότε τα ερωτήματα πρέπει να προσαρμοστούν για την ανάκτηση χρήσιμων αποτελεσμάτων. Η Boa επέστρεψε αποτελέσματα για όλα τα ερωτήματα, εκτός από το MortgageCalculator, για το οποίο χρειάστηκε να αλλάξουμε το όνομα κλάσης σε Mortgage και επίσης να αντιστοιχίσουμε το ερώτημα Boa μόνο με τη μέθοδο setRate. Ωστόσο, ο Functional Scorer του QualBoa χρησιμοποιεί όλες τις μεθόδους για τη βαθμολόγηση.

Πίνακας 9.3: Σύνολο Ερωτημάτων για την Αξιολόγηση του QualBoa

Κλάση	Μέθοδοι
Calculator	add, sub, div, mult
ComplexNumber	ComplexNumber, add, getRealPart, getImaginaryPart
Matrix	Matrix, set, get, multiply
MortgageCalculator	setRate, setPrincipal, setYears, getMonthlyPayment
ShoppingCart	getItemCount, getBalance, addItem, empty, removeItem
Spreadsheet	put, get
Stack	push, pop

Για κάθε ερώτημα, εξετάζουμε τα πρώτα 30 αποτελέσματα του QualBoa, και σημειώνουμε κάθε αποτέλεσμα ως χρήσιμο (useful) ή μη χρήσιμο (not useful). Ένα αποτέλεσμα θεωρείται χρήσιμο αν η ενσωμάτωσή του στον κώδικα του προγραμματιστή απαιτεί ελάχιστη ή καμία προσπάθεια. Με άλλα λόγια, τα χρήσιμα τμήματα κώδικα πρέπει να είναι κατανοητά και ταυτόχρονα να καλύπτουν την απαιτούμενη λειτουργικότητα. Η παραπάνω διαδικασία έγινε χωρίς γνώση της κατάταξης των αποτελεσμάτων για να αποφύγουμε ζητήματα σχετικά με την εγκυρότητα της μεθοδολογίας αξιολόγησης (threads to validity).

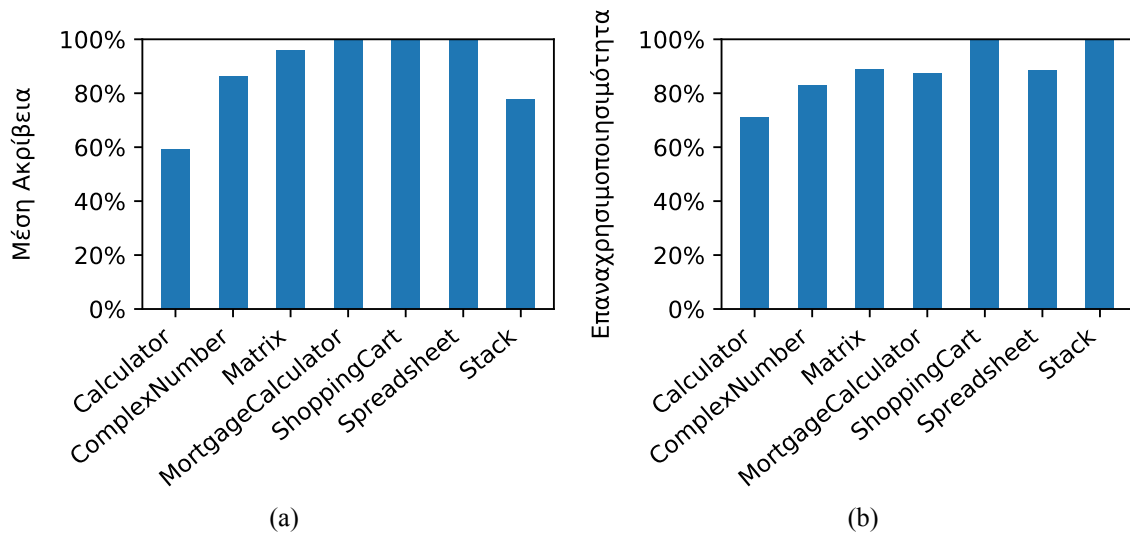
Έχοντας λοιπόν τα σημειωμένα αποτελέσματα, μετράμε τον αριθμό των σχετικών (χρήσιμων) αποτελεσμάτων για κάθε ερώτημα και υπολογίζουμε επιπλέον τη μέση ακρίβεια (average precision) για την κατάταξη των αποτελεσμάτων. Ο λόγος που επιλέγουμε τη μέση ακρίβεια ως μετρική είναι ότι καλύπτει όχι μόνο τη σχετικότητα των αποτελεσμάτων αλλά και την κατάταξή τους. Αξιολογούμε περαιτέρω τα αποτελέσματα χρησιμοποιώντας το μοντέλο επαναχρησιμοποιησιμότητας του QualBoa και, μετά από την κανονικοποίηση σε ποσοστά, αναφέρουμε και τη μέση βαθμολογία επαναχρησιμοποιησιμότητας (reusability score) για τα χρήσιμα/σχετικά αποτελέσματα κάθε ερωτήματος. Τα αποτελέσματα της αξιολόγησης συνοψίζονται στον Πίνακα 9.4, ενώ στο Σχήμα 9.4 απεικονίζονται γραφικά η τιμή της μέσης ακρίβειας για κάθε ερώτημα και η μέση βαθμολογία επαναχρησιμοποιησιμότητας για κάθε ερώτημα.

Όσον αφορά τον αριθμό των σχετικών αποτελεσμάτων, το QualBoa είναι αρκετά αποτελεσματικό. Συγκεκριμένα, το σύστημά μας ανακτά επιτυχώς τουλάχιστον 10 χρήσιμα αποτελέσματα για 5 από τα 7 ερωτήματα. Επιπλέον, η μέση ακρίβεια για σχεδόν όλα τα ερωτήματα είναι υψηλότερη από 75%, που σημαίνει ότι τα αποτελέσματα κατατάσσονται αποτελεσματικά. Αυτό είναι ιδιαίτερα εμφανές σε περιπτώσεις που ο αριθμός των αποτελεσμάτων είναι περιορισμένος, όπως στο ερώτημα MortgageCalculator ή στο ερώτημα Spreadsheet. Οι μέγιστες τιμές ακρίβειας (100%) δείχνουν ότι τα σχετικά αποτελέσματα για αυτά τα ερωτήματα τοποθετούνται στην κορυφή της κατάταξης.



Πίνακας 9.4: Αποτελέσματα Αξιολόγησης του QualBoa

Ερώτημα	#Σχετικά Αποτελέσματα	Μέση Ακρίβεια	Επαναχρησιμοποιησιμότητα
Calculator	18	59.27%	70.83%
ComplexNumber	15	86.18%	82.76%
Matrix	10	95.88%	88.68%
MortgageCalculator	7	100.00%	87.17%
ShoppingCart	13	100.00%	100.00%
Spreadsheet	2	100.00%	88.54%
Stack	22	77.59%	100.00%
Μέσες τιμές	12.43	88.42%	88.28%



Σχήμα 9.4: Διαγράμματα αξιολόγησης του QualBoa που απεικονίζουν (a) τη μέση ακρίβεια, και (b) τη μέση βαθμολογία επαναχρησιμοποιησιμότητας, για κάθε ερώτημα

Τα αποτελέσματα είναι επίσης σε μεγάλο βαθμό επαναχρησιμοποιήσιμα, όπως φαίνεται από τη βαθμολογία επαναχρησιμοποιησιμότητας για κάθε ερώτημα. Σε 5 από τα 7 ερωτήματα, η μέση τιμή της επαναχρησιμοποιησιμότητας των αποτελεσμάτων είναι αρκετά κοντά ή υψηλότερη από 87.5%, που σημαίνει ότι κατά μέσο όρο τα τμήματα κώδικα δεν υπερβαίνουν περισσότερα από 1 όρια μετρικών, από αυτά που ορίστηκαν στον Πίνακα 9.2. Επομένως έχουν βαθμολογία επαναχρησιμοποιησιμότητας τουλάχιστον 7 στα 8. Η επαναχρησιμοποιησιμότητα των τμημάτων σχετίζεται επίσης με την πολυπλοκότητα του ερωτήματος. Συγκεκριμένα, τα τμήματα κώδικα για τα ερωτήματα που αφορούν δομές δεδομένων, όπως το Stack ή το ShoppingCart, δεν είναι επιρρεπή σε προβλήματα ποιότητας και συνεπώς έχουν τέλειες βαθμολογίες (100%). Αντίθετα, τα τμήματα κώδικα για πιο πολύπλοκα ερωτήματα, όπως το Calculator ή το ComplexNumber, μπορεί να περιέχουν μεθόδους που αλληλεπιδρούν, έτσι οι βαθμολογίες επαναχρησιμοποιησιμότητας τους είναι χαμηλότερες.

## 9.4 Συμπεράσματα

Σε αυτό το κεφάλαιο, παρουσιάσαμε το QualBoa, ένα σύστημα που προτείνει τμήματα κώδικα τα οποία καλύπτουν την επιθυμητή λειτουργικότητα όπως αυτή ορίζεται από το ερώτημα του προγραμματιστή, ενώ ταυτόχρονα είναι επαναχρησιμοποιήσιμα. Η αξιολόγηση του συστήματος δείχνει ότι το QualBoa είναι αποτελεσματικό για την ανάκτηση επαναχρησιμοποιήσιμων αποτελεσμάτων. Μελλοντικές επεκτάσεις μπορούν να υπάρξουν προς διάφορες κατευθύνσεις. Ο βασικός άξονας, ωστόσο, είναι το μοντέλο επαναχρησιμοποιησιμότητας του QualBoa. Σε αυτό τον άξονα, στο επόμενο κεφάλαιο προτείνουμε μια μεθοδολογία για την αυτοματοποιημένη κατασκευή ενός μοντέλο επαναχρησιμοποιησιμότητας. Τέλος, μια άλλη ενδιαφέρουσα ιδέα θα ήταν η κατασκευή ενός μοντέλου το οποίο θα προσαρμόζεται αυτοματοποιημένα, ούτως ώστε τα προτεινόμενα τμήματα κώδικα να έχουν τα ίδια χαρακτηριστικά ποιότητας με τον κώδικα του προγραμματιστή.

# 10

## Αξιολόγηση της Επαναχρησιμοποιησιμότητας Κώδικα

### 10.1 Επισκόπηση

Όπως ήδη αναφέρθηκε, η αξιολόγηση της ποιότητας τμημάτων λογισμικού είναι πολύ σημαντική στο πλαίσιο της επαναχρησιμοποίησης κώδικα. Η διαδικασία ενσωμάτωσης τμημάτων κώδικα στον πηγαίο κώδικα του προγραμματιστή μπορεί να οδηγήσει σε αποτυχίες, καθώς τα τμήματα αυτά μπορεί να μην ικανοποιούν βασικές λειτουργικές ή μη λειτουργικές απαιτήσεις (functional/non-functional requirements). Επομένως, η αξιολόγηση της ποιότητας των προς επαναχρησιμοποίηση τμημάτων αποτελεί μια σημαντική πρόκληση για την ερευνητική κοινότητα.

Μια σημαντική πτυχή αυτής της πρόκλησης είναι το γεγονός ότι η ποιότητα είναι μια σχετική έννοια και μπορεί να ερμηνεύεται διαφορετικά από το κάθε άτομο [16]. Επομένως, προκύπτει η ανάγκη για προτυποποίηση (standardization), ώστε να δημιουργηθεί μια κοινή αναφορά. Προς αυτήν την κατεύθυνση, τα διεθνή πρότυπα ISO/IEC 25010:2011 [17] και ISO/IEC 9126 [235] ορίζουν ένα μοντέλο ποιότητας που αποτελείται από οκτώ βασικά χαρακτηριστικά ποιότητας (quality attributes) και σχετικές ιδιότητες ποιότητας (quality properties). Σύμφωνα με αυτά τα πρότυπα, η επαναχρησιμοποιησιμότητα (reusability), δηλαδή το κατά πόσο ένα τμήμα κώδικα είναι επαναχρησιμοποιήσιμο (reusable), σχετίζεται με τέσσερις βασικές ιδιότητες ποιότητας: την *Κατανοησιμότητα* (Understandability), τη *Δυνατότητα Εκμάθησης* (Learnability), τη *Λειτουργικότητα* (Operability) και την *Ελκυστικότητα* (Attractiveness). Αυτές οι ιδιότητες είναι άμεσα συσχετισμένες με τη *Χρησιμότητα* (Usability), ενώ επιπλέον επηρεάζουν τη *Λειτουργική Καταλληλότητα* (Functional Suitability), τη *Συντηρησιμότητα* (Maintainability) και τη *Φορητότητα* (Portability), καλύπτοντας έτσι τα τέσσερα χαρακτηριστικά ποιότητας που αφορούν την επαναχρησιμοποιησιμότητα [148, 236] (τα υπόλοιπα χαρακτηριστικά είναι η *Αξιοπιστία* (Reliability), η *Αποδοτικότητα Επίδοσης* (Performance Efficiency), η *Ασφάλεια* (Security) και η *Συμβατότητα* (Compatibility)).

Οι σύγχρονες ερευνητικές προσπάθειες εστιάζουν κυρίως σε χαρακτηριστικά ποιότητας όπως η συντηρησιμότητα και η ασφάλεια, ενώ η αξιολόγηση της επαναχρησιμοποίησης δεν έχει αναλυθεί επαρκώς. Υπάρχουν διάφορες προσεγγίσεις που έχουν στόχο την αξιολόγηση της ποιότητας [111, 112, 148, 236] και της επαναχρησιμοποίησης [237, 238] τμημάτων κώδικα χρησιμοποιώντας μετρικές στατικής ανάλυσης (static analysis metrics), όπως οι μετρικές C&K (C&K metrics) [41]. Αυτές οι προσεγγίσεις, ωστόσο, βασίζονται σε όρια μετρικών (metric thresholds) τα οποία, είτε καθορίζονται από τη βιβλιογραφία [111, 112, 148], είτε εξάγονται αυτοματοποιημένα με χρήση κάποιου μοντέλου [108, 239–241], περιορίζονται συνήθως από την έλλειψη αντικειμενικών τιμών ποιότητας (ground truth quality values). Ως εκ τούτου, καταφεύγουν συνήθως σε βοήθεια εμπειρογνομόνων (quality experts), που μπορεί να είναι υποκειμενική, να διαφέρει από περίπτωση σε περίπτωση ή ακόμη πιο συχνά να είναι μη διαθέσιμη [113, 242].

Σε αυτό το κεφάλαιο, αίρουμε την ανάγκη ύπαρξης εξακριβωμένων τιμών ποιότητας, χρησιμοποιώντας την ποιότητα όπως την αντιλαμβάνεται ο χρήστης (user-perceived quality) ως μέτρο της επαναχρησιμοποίησης (reusability) ενός τμήματος λογισμικού. Από τη σκοπιά του προγραμματιστή, προτείνουμε να συσχετίσουμε την ποιότητα με το κατά πόσο ένα τμήμα λογισμικού προτιμάται από την κοινότητα, υποστηρίζοντας έτσι ότι η δημοτικότητα (popularity) ενός λογισμικού μπορεί να αποτελεί ένδειξη της επαναχρησιμοποίησής του. Αφού διερευνήσουμε την παραπάνω υπόθεση, στη συνέχεια εξετάζουμε περαιτέρω τη σχέση μεταξύ δημοτικότητας και ποιότητας όπως την αντιλαμβάνεται ο χρήστης, χρησιμοποιώντας ένα σύνολο δεδομένων από τμήματα κώδικα, των οποίων η δημοτικότητα καθορίζεται από την προτίμησή τους στο GitHub. Βασιζόμενοι στα αποτελέσματα αυτής τη διερεύνησης, σχεδιάζουμε μια μεθοδολογία που καλύπτει τις τέσσερις βασικές ιδιότητες ποιότητας που σχετίζονται με την επαναχρησιμοποίησης, χρησιμοποιώντας ένα σύνολο από μετρικές στατικής ανάλυσης. Χρησιμοποιώντας αυτές τις μετρικές, καθώς και τη δημοτικότητα των έργων από το GitHub ως βασική γνώση (ground truth), είμαστε σε θέση να αξιολογήσουμε την επαναχρησιμοποίησης οποιοδήποτε τμήματος κώδικα. Αρχικά, μοντελοποιούμε τη συμπεριφορά των μετρικών για να μεταφράσουμε τις τιμές τους σε μια βαθμολογία επαναχρησιμοποίησης [243]. Στη συνέχεια, οι συμπεριφορές των μετρικών χρησιμοποιούνται για την εκπαίδευση ενός συνόλου από μοντέλα αξιολόγησης επαναχρησιμοποίησης. Τα μοντέλα αυτά είναι ικανά να αξιολογήσουν το βαθμό στον οποίο ένα τμήμα κώδικα (είτε σε επίπεδο κλάσης είτε σε επίπεδο πακέτου) είναι επαναχρησιμοποίησιμο, από την οπτική του προγραμματιστή.

## 10.2 Βιβλιογραφία για την Αξιολόγηση Επαναχρησιμοποίησης

Οι περισσότερες προσεγγίσεις αξιολόγησης ποιότητας και επαναχρησιμοποίησης χρησιμοποιούν μετρικές στατικής ανάλυσης για να εκπαιδεύσουν κάποιο μοντέλο αξιολόγησης [111, 237, 238, 244]. Γενικώς, η αξιολόγηση της ποιότητας με μετρικές δεν είναι απλή διαδικασία, καθώς απαιτεί τον καθορισμό κάποιων ορίων για τις μετρικές (metric thresholds) [148], που συνήθως παρέχονται από ειδικούς ποιότητας (quality experts) οι οποίοι χρειάζεται να εξετάσουν τον πηγαίο κώδικα [245]. Ωστόσο, η εξέταση κώδικα δεν είναι πάντα εφικτή, ιδιαίτερα για μεγάλα και πολύπλοκα έργα που μεταβάλλονται συχνά. Επιπλέον, η βοήθεια

από κάποιον ειδικό μπορεί να είναι υποκειμενική ή να εξαρτάται σε μεγάλο βαθμό από την περίπτωση του έργου.

Άλλες προσεγγίσεις μπορεί να απαιτούν πολλές παραμέτρους για την κατασκευή των μοντέλων αξιολόγησης ποιότητας [113], που, όπως και προηγουμένως, μπορεί να εξαρτώνται σε μεγάλο βαθμό από τον τομέα εφαρμογής του λογισμικού και/ή να επηρεάζονται εύκολα από υποκειμενικές κρίσεις. Επομένως, μια κοινή πρακτική περιλαμβάνει την εξαγωγή ορίων μετρικών εφαρμόζοντας τεχνικές μηχανικής μάθησης (machine learning) σε κάποιο κατάλληλα διαμορφωμένο σύνολο δεδομένων (benchmark dataset). Ο Ferreira και οι συνεργάτες του [107] προτείνουν μια μεθοδολογία για την εκτίμηση ορίων προσαρμόζοντας (fitting) τις τιμές των μετρικών σε κατανομές πιθανότητας (probability distributions), ενώ ο Alves και οι συνεργάτες του [106] εφαρμόζουν ένα μοντέλο μετρικών με βάρη για να εξάγουν όρια μέσω στατιστικής ανάλυσης. Άλλες προσεγγίσεις περιλαμβάνουν την εξαγωγή ορίων με χρήση bootstrapping [246] ή ανάλυσης καμπυλών ROC (ROC curve analysis) [247]. Ωστόσο, η αποτελεσματικότητα αυτών των προσεγγίσεων υπόκειται στην επιλογή των έργων για τη διαμόρφωση του συνόλου δεδομένων. Τέλος, υπάρχουν προσεγγίσεις που εστιάζουν συγκεκριμένα στην επαναχρησιμοποιησιμότητα [239–241, 248]. Όπως και στην αξιολόγηση ποιότητας, αυτές οι προσπάθειες αναλαμβάνουν αρχικά την ποσοτικοποίηση της επαναχρησιμοποιησιμότητας, π.χ. μέσω της συχνότητας επαναχρησιμοποίησης [248]. Στη συνέχεια, εφαρμόζονται τεχνικές μηχανικής μάθησης για την εκπαίδευση μοντέλων αξιολόγησης επαναχρησιμοποιησιμότητας χρησιμοποιώντας ως είσοδο μετρικές στατικής ανάλυσης [239–241].

Παρόλο που οι παραπάνω προσεγγίσεις είναι αποτελεσματικές σε κάποιες περιπτώσεις, η εφαρμογή τους σε πραγματικά προβλήματα είναι περιορισμένη. Κατ' αρχάς, οι προσεγγίσεις που χρησιμοποιούν προκαθορισμένα όρια μετρικών [111, 112, 148, 245], ακόμα και με την καθοδήγηση κάποιου ειδικού, δε μπορούν να επεκτείνουν το πεδίο εφαρμογής τους πέρα από την κατηγορία και τα χαρακτηριστικά του λογισμικού σύμφωνα με τα οποία σχεδιάστηκαν. Τα συστήματα αυτοματοποιημένης αξιολόγησης ποιότητας φαίνονται να μην αντιμετωπίζουν αυτά τα προβλήματα [106–108, 239–241, 246, 247]. Ωστόσο, περιορίζονται και αυτά από τη γνώση κάποιου ειδικού για την κατασκευή ενός ground truth και την εκπαίδευση του μοντέλου, κι έτσι δεν αξιολογούν την επαναχρησιμοποιησιμότητα κώδικα σύμφωνα με τις διαφορετικές ανάγκες των προγραμματιστών. Ως εκ τούτου, αρκετά ερευνητικά αλλά και εμπορικά συστήματα [113, 249] περιορίζονται στην παροχή αναφορών για μετρικές ποιότητας, αφήνοντας την προσαρμογή στους προγραμματιστές ή τους ειδικούς ποιότητας.

Για όλα τα παραπάνω ζητήματα, κατασκευάζουμε ένα σύστημα αξιολόγησης της επαναχρησιμοποιησιμότητας που μπορεί να παρέχει μια συγκεκριμένη βαθμολογία για κάθε κλάση και κάθε πακέτο ενός έργου λογισμικού. Το σύστημά μας δε χρησιμοποιεί προκαθορισμένα όρια μετρικών, ενώ επιπλέον αποφεύγει τον προσδιορισμό τιμών ποιότητας από κάποιον ειδικό· αντί αυτού, χρησιμοποιούμε τη δημοτικότητα (δηλαδή την ποιότητα όπως την αντιλαμβάνεται ο χρήστης) ως ένα ground truth για την επαναχρησιμοποιησιμότητα [149]. Καθώς τα δημοφιλή τμήματα λογισμικού έχουν υψηλά ποσοστά επαναχρησιμοποίησης [250], αντιλαμβανόμαστε ότι η προτίμηση των προγραμματιστών μπορεί να αποτελέσει ένα μέτρο της υψηλής επαναχρησιμοποιησιμότητας. Εφαρμόζουμε αυτή την ιδέα για να αξιολογήσουμε μεμονωμένα τον αντίκτυπο κάθε μετρικής στο βαθμό επαναχρησιμοποιησιμότητας των τμημάτων λογισμικού, και στη συνέχεια συγκεντρώνουμε τα αποτελέσματα της ανά-

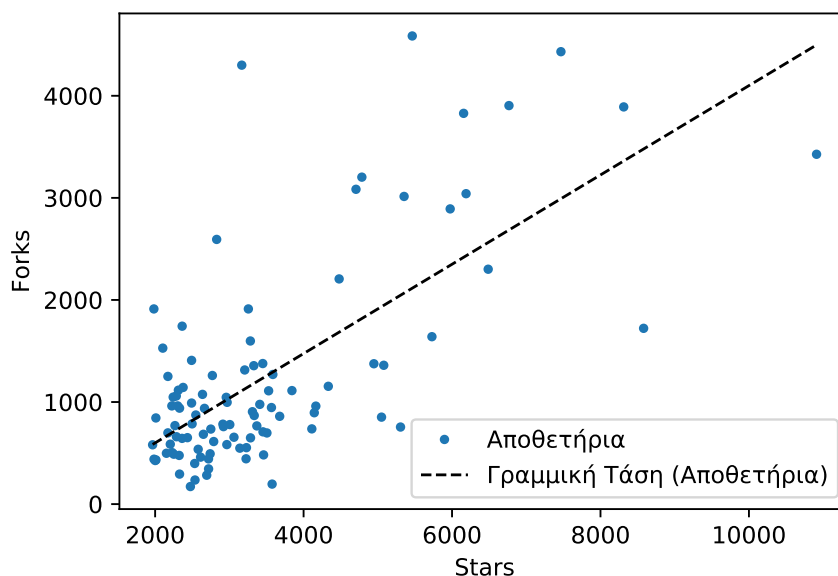
λυσής μας για να κατασκευάσουμε μια τελική βαθμολογία επαναχρησιμοποιησιμότητας. Τέλος, ποσοτικοποιούμε το βαθμό επαναχρησιμοποιησιμότητας τμημάτων λογισμικού σε επίπεδο κλάσης και σε επίπεδο πακέτου, εκπαιδεύοντας μοντέλα αξιολόγησης επαναχρησιμοποιησιμότητας που εκτιμούν το κατά πόσο ένα τμήμα θεωρείται κατάλληλο για επαναχρησιμοποίηση από τους προγραμματιστές.

## 10.3 Συσχέτιση Επαναχρησιμοποιησιμότητας με Δημοτικότητα

Σε αυτό το υποκεφάλαιο, αναλύουμε την εν δυνάμει σύνδεση μεταξύ της επαναχρησιμοποιησιμότητας (reusability) και της δημοτικότητας (popularity) και δείχνουμε πώς η μεθοδολογία μας βασίζεται στα πρότυπα ISO.

### 10.3.1 Δημοτικότητα στο GitHub ως Ένδειξη Επαναχρησιμοποιησιμότητας

Για να υποστηρίξουμε τους ισχυρισμούς μας ότι η επαναχρησιμοποιησιμότητα σχετίζεται με τη δημοτικότητα, υπολογίσαμε τη συσχέτιση (correlation) μεταξύ του αριθμού των stars και του αριθμού των forks για τα 100 πιο δημοφιλή αποθετήρια Java του GitHub (δηλαδή τα 100 αποθετήρια με τα περισσότερα stars) [149]. Όπως φαίνεται στο Σχήμα 10.1, υπάρχει ισχυρή θετική συσχέτιση μεταξύ των δύο μετρικών. Συγκεκριμένα, η τιμή του συντελεστή συσχέτισης είναι 0.68.



Σχήμα 10.1: Διάγραμμα που απεικονίζει τον αριθμό των stars με τον αριθμό των forks για τα 100 πιο δημοφιλή αποθετήρια Java του GitHub

### 10.3.2 Μοντελοποίηση Δημοτικότητας στο GitHub

Συσχετίζουμε την επαναχρησιμοποιησιμότητα με τις παρακάτω βασικές ιδιότητες ποιότητας που περιγράφονται στα πρότυπα ISO/IEC 25010 [17] και ISO/IEC 9126 [235]): την *Κατανοησιμότητα (Understandability)*, τη *Δυνατότητα Εκμάθησης (Learnability)*, τη *Λειτουργικότητα (Operability)* και την *Ελκυστικότητα (Attractiveness)*. Σύμφωνα με έρευνα που πραγματοποιήθηκε από το ARiSA [251], διάφορες μετρικές στατικής ανάλυσης είναι σχετικές με αυτές τις ιδιότητες. Στον Πίνακα 10.1 συνοψίζονται οι σχέσεις μεταξύ των κατηγοριών των μετρικών στατικής ανάλυσης με τις προαναφερθείσες ιδιότητες. Οι ανάλογες σχέσεις συμβολίζονται με “P” και οι αντιστρόφως ανάλογες με “IP”.

Πίνακας 10.1: Κατηγορίες Μετρικών Στατικής Ανάλυσης που σχετίζονται με την Επαναχρησιμοποιησιμότητα

Κατηγορίες	Σχετικά με Forks		Σχετικά με Stars	
	Κατανοησιμότητα	Δυν. Εκμάθησης	Λειτουργικότητα	Ελκυστικότητα
Πολυπλοκότητα	IP	IP	IP	P
Σύζευξη	IP	IP	IP	P
Συνοχή	P	P	P	P
Τεκμηρίωση	P	P	–	–
Κληρονομικότητα	IP	IP	–	P
Μέγεθος	IP	IP	IP	P

P: Ανάλογη Σχέση  
 IP: Αντιστρόφως Ανάλογη Σχέση

Όπως ήδη αναφέρθηκε, χρησιμοποιούμε τα stars και τα forks του GitHub ώστε να ποσοτικοποιήσουμε την επαναχρησιμοποιησιμότητα και στη συνέχεια να τη συσχετίσουμε με τις ιδιότητες που αναφέρθηκαν. Καθώς τα forks μετρούν πόσες φορές έχει αντιγραφεί (clone) ένα αποθετήριο κώδικα, μπορούμε να τα συσχετίσουμε με την Κατανοησιμότητα, τη Δυνατότητα Εκμάθησης και τη Λειτουργικότητα, καθώς αυτές οι ιδιότητες αναφέρονται στο κατά πόσο ένα τμήμα κώδικα είναι επαναχρησιμοποιήσιμο. Τα stars, από την άλλη πλευρά, αντικατοπτρίζουν το πλήθος των προγραμματιστών που θεώρησαν ότι το αποθετήριο είναι ενδιαφέρον, οπότε μπορούμε να τα χρησιμοποιήσουμε ως ένα μέτρο Ελκυστικότητας.

## 10.4 Μοντελοποίηση Επαναχρησιμοποιησιμότητας

### 10.4.1 Σύνολο Δεδομένων

Χρησιμοποιήσαμε το SourceMeter [252] για να κατασκευάσουμε ένα σύνολο δεδομένων που περιέχει τιμές για τις μετρικές στατικής ανάλυσης που φαίνονται στο Πίνακα 10.2. Οι τιμές υπολογίστηκαν για τα 100 αποθετήρια Java του GitHub με τα περισσότερα stars

και τα 100 αποθετήρια Java του GitHub με τα περισσότερα forks (συνολικά 137 αποθετήρια, καθώς υπάρχουν επικαλύψεις). Αυτά τα έργα έχουν περισσότερες από 12 εκατομμύρια γραμμές κώδικα που εκτείνονται σε σχεδόν 15 χιλιάδες πακέτα και 150 χιλιάδες κλάσεις.

Πίνακας 10.2: Μετρικές Στατικής Ανάλυσης και Εφαρμογή τους σε Διαφορετικά Επίπεδα

ID	Όνομα	Κλάση	Πακέτο
<i>Πολυπλοκότητα</i>			
NL{·,E}	Nesting Level {Else-If}	×	
WMC	Weighted Methods per Class	×	
<i>Σύζευξη</i>			
CBO{·,I}	Coupling Between Object classes {Inverse}	×	
N{I,O}I	Number of {Incoming, Outgoing} Invocations	×	
RFC	Response set For Class	×	
<i>Συνοχή</i>			
LCOM5	Lack of Cohesion in Methods 5	×	
<i>Τεκμηρίωση</i>			
AD	API Documentation	×	
{·,T}CD	{Total} Comment Density	×	×
{·,T}CLOC	{Total} Comment Lines of Code	×	×
DLOC	Documentation Lines of Code	×	
P{D,U}A	Public {Documented, Undocumented} API	×	×
TAD	Total API Documentation		×
TP{D,U}A	Total Public {Documented, Undocumented} API		×
<i>Κληρονομικότητα</i>			
DIT	Depth of Inheritance Tree	×	
NO{A,C}	Number of {Ancestors, Children}	×	
NO{D,P}	Number of {Descendants, Parents}	×	
<i>Μέγεθος</i>			
{·,T}{·,L}LOC	{Total} {Logical} Lines of Code	×	×
N{G,S}	Number of {Getters, Setters}	×	×
N{A,M}	Number of {Attributes, Methods}	×	×
N{CL,EN,IN,P}	Number of {Classes, Enums, Interfaces, Packages}		×
NL{G,S}	Number of Local {Getters, Setters}	×	
NL{A,M}	Number of Local {Attributes, Methods}	×	
NLP{A,M}	Number of Local Public {Attributes, Methods}	×	
NP{A,M}	Number of Public {Attributes, Methods}	×	×
NOS	Number of Statements	×	
TNP{CL,EN,IN}	Total Number of Public {Classes, Enums, Interfaces}		×
TN{CL,DI,EN,FI}	Total Number of {Classes, Directories, Enums, Files}		×

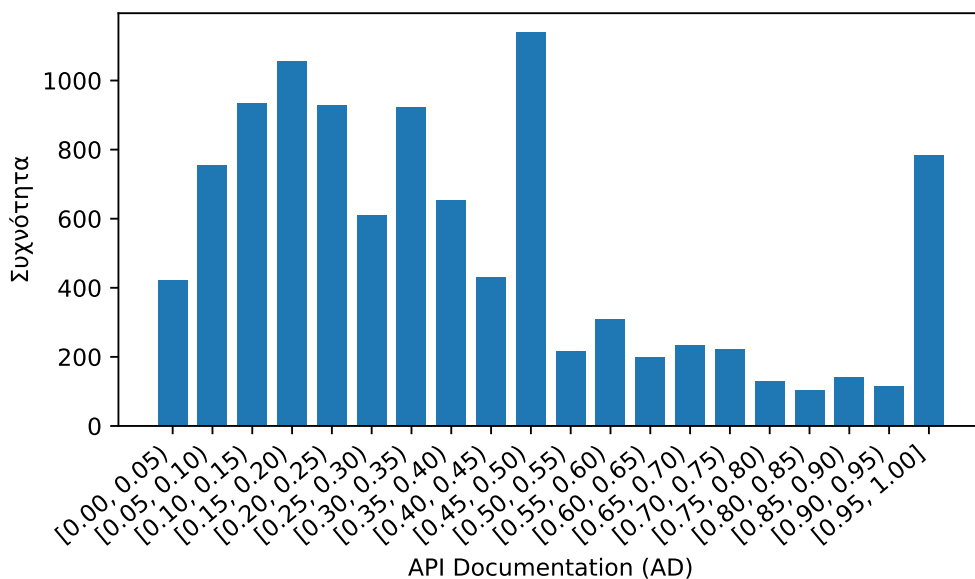


## 10.4.2 Εκτίμηση της Επιρροής των Μετρικών στην Επαναχρησιμοποιησιμότητα

Καθώς τα stars και τα forks αναφέρονται σε επίπεδο αποθετηρίου, δεν είναι από μόνα τους επαρκή για την εκτίμηση της επαναχρησιμοποιησιμότητας τμημάτων σε επίπεδο κλάσης και επίπεδο πακέτου. Συνεπώς, χρειάζεται να εκτιμήσουμε την επαναχρησιμοποιησιμότητα με χρήση μετρικών στατικής ανάλυσης. Για κάθε μετρική, αρχικά χωρίζουμε τις τιμές της σε περιοχές με βάση την κατανομή (distribution-based binning), και στη συνέχεια συσχετίζουμε τις τιμές αυτές με τις τιμές των stars/forks για να ενσωματώσουμε την πληροφορία επαναχρησιμοποιησιμότητας. Η τελική εκτίμηση επαναχρησιμοποιησιμότητας υπολογίζεται συναθροίζοντας (aggregation) τις εκτιμήσεις που εξάγονται από όλες τις μετρικές.

### 10.4.2.1 Διαχωρισμός με βάση την κατανομή

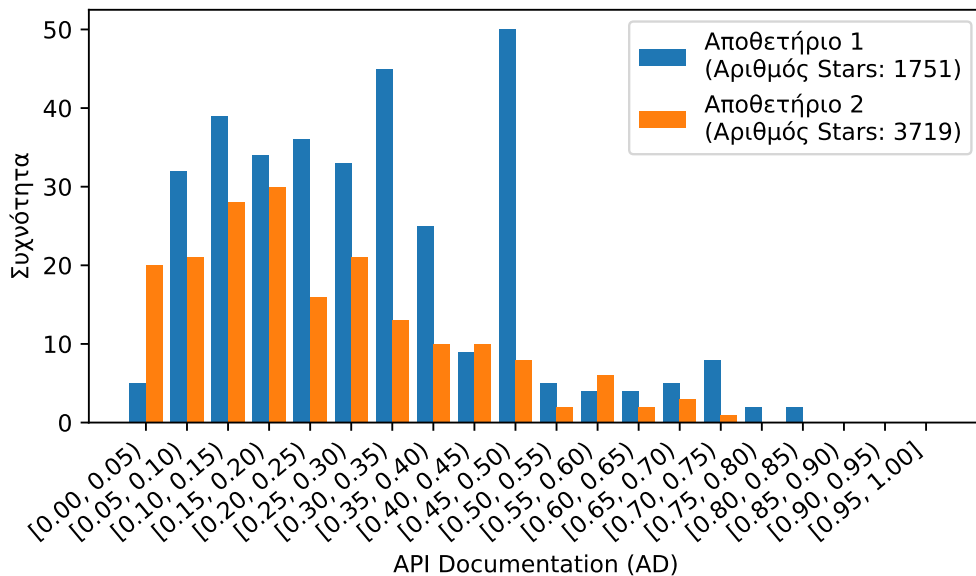
Καθώς οι τιμές των μετρικών στατικής ανάλυσης κατανέμονται διαφορετικά εντός των αποθετηρίων, αρχικά ορίζουμε ένα σύνολο από αντιπροσωπευτικά διαστήματα (bins), που είναι τα ίδια για όλα τα αποθετήρια. Τα διαστήματα θα πρέπει να προσεγγίζουν με ακρίβεια την πραγματική κατανομή των τιμών. Για το σκοπό αυτό χρησιμοποιούμε τις τιμές κάθε μετρικής από όλα τα πακέτα (ή τις κλάσεις) τους συνόλου δεδομένων για να δημιουργήσουμε μια γενικευμένη κατανομή, και στη συνέχεια να καθορίσουμε το βέλτιστο μέγεθος των διαστημάτων (bin size), το οποίο ελαχιστοποιεί την απώλεια πληροφορίας (information loss). Χρειάζεται να εφαρμόσουμε μια γενικευμένη και ακριβή στρατηγική διαχωρισμού (binning strategy), ενώ είναι σημαντικό να λάβουμε υπόψη ότι οι τιμές των μετρικών δεν ακολουθούν απαραίτητα κανονική κατανομή (normal distribution). Επομένως, χρησιμοποιούμε τη μέθοδο *Doane* (*Doane formula*) [253] για να επιλέξουμε το μέγεθος των διαστημάτων-περιοχών, η οποία λαμβάνει υπόψη πιθανές ασυμμετρίες (skewness) των δεδομένων. Στο Σχήμα 10.2 απεικονίζεται το ιστόγραμμα (histogram) για τη μετρική τεκμηρίωσης API (API Documentation - AD), που θα χρησιμοποιηθεί ως παράδειγμα σε αυτήν την ενότητα.



Σχήμα 10.2: Κατανομή σε Επίπεδο Πακέτου της Μετρικής AD για όλα τα Αποθετήρια

Ακολουθώντας τη στρατηγική μας, δημιουργούνται 20 διαστήματα (bins). Οι τιμές της AD παρουσιάζουν θετική ασυμμετρία (positive skewness), ενώ η υψηλότερη συχνότητά τους εμφανίζεται στο διάστημα  $[0.1, 0.5]$ . Αφού επιλέξουμε τα κατάλληλα διαστήματα για κάθε μετρική, στη συνέχεια κατασκευάζουμε τα ιστογράμματα για κάθε ένα από τα 137 αποθετήρια.

Στο Σχήμα 10.3 απεικονίζονται τα ιστογράμματα για δύο διαφορετικά αποθετήρια, όπου είναι σαφές ότι κάθε αποθετήριο ακολουθεί διαφορετική κατανομή για τις τιμές της μετρικής AD. Αυτό είναι αναμενόμενο, καθώς κάθε αποθετήριο αποτελεί μια ξεχωριστή οντότητα λογισμικού που μπορεί να έχει διαφορετικό πεδίο εφαρμογής και/ή να αναπτύσσεται από διαφορετικούς προγραμματιστές· επομένως, κάθε αποθετήριο έχει διαφορετικά χαρακτηριστικά.



Σχήμα 10.3: Κατανομή σε Επίπεδο Πακέτου της Μετρικής AD για δύο Αποθετήρια

#### 10.4.2.2 Συσχέτιση Τιμών Διαστημάτων με GitHub Stars και Forks

Χρησιμοποιούμε τα ιστογράμματα που παράγονται (ένα για κάθε αποθετήριο με βάση τα διαστήματα που καθορίστηκαν στο προηγούμενο βήμα) για να κατασκευάσουμε ένα σύνολο από τιμές δεδομένων, με σκοπό να συνδέσουμε κάθε τιμή διαστήματος για κάθε μετρική (εδώ για την AD) με μια τιμή από GitHub stars (ή forks). Έτσι, συναθροίζουμε (aggregation) τις τιμές κάθε μετρικής για όλα τα διαστήματα, δηλαδή για τη μετρική  $X$  και το διάστημα (bin) 1, συγκεντρώνουμε όλες τις τιμές stars (ή forks) που αντιστοιχούν σε πακέτα (ή κλάσεις) για τα οποία η τιμή της μετρικής ανήκει στο διάστημα 1, και τα συναθροίζουμε παίρνοντας το μέσο όρο τους. Αυτή η διαδικασία επαναλαμβάνεται για κάθε μετρική και για κάθε διάστημα.

Πρακτικά, για κάθε τιμή μιας μετρικής σε κάποιο διάστημα, συγκεντρώνουμε όλες τις σχετικά εγγραφές δεδομένων και υπολογίζουμε το σταθμισμένο μέσο όρο (weighted average) του πλήθους των stars (ή των forks) τους. Η τιμή που προκύπτει αναπαριστά την επαναχρησιμοποιησιμότητα με βάση τα stars (ή τα forks) για το συγκεκριμένο διάστημα. Οι βαθμο-

λογίες επαναχρησιμοποιησιμότητας ορίζονται από τις παρακάτω εξισώσεις:

$$RS_{Metric}(i) = \sum_{repo=1}^N freq_{p.u.}(i) \cdot \log(S(repo)) \quad (10.1)$$

$$RF_{Metric}(i) = \sum_{repo=1}^N freq_{p.u.}(i) \cdot \log(F(repo)) \quad (10.2)$$

όπου το  $RS_{Metric}(i)$  αναφέρεται στη βαθμολογία επαναχρησιμοποιησιμότητας με βάση τα stars του  $i$ -ου διαστήματος για την υπό εξέταση μετρική, ενώ το  $RF_{Metric}(i)$  στην αντίστοιχη βαθμολογία επαναχρησιμοποιησιμότητας με βάση τα forks. Τα  $S(repo)$  και  $F(repo)$  αναφέρονται στον αριθμό των stars και στον αριθμό των forks του αποθετηρίου, αντίστοιχα, ενώ ο λογάριθμος χρησιμοποιείται για την εξομάλυνση (smoothing) των μεγάλων διαφορών στους αριθμούς των stars και forks μεταξύ αποθετηρίων. Τέλος, ο όρος  $freq_{p.u.}(i)$  είναι η κανονικοποιημένη/σχετική συχνότητα (relative frequency) της τιμής της μετρικής για το  $i$ -ο διάστημα, που ορίζεται ως:

$$freq_{p.u.}(i) = \frac{F_i}{\sum_{i=1}^N F_i} \quad (10.3)$$

όπου  $F_i$  είναι η απόλυτη συχνότητα (absolute frequency, δηλαδή το πλήθος) των τιμών της μετρικής που ανήκουν στο  $i$ -ο διάστημα. Για παράδειγμα, αν ένα αποθετήριο είχε 3 διαστήματα με 5 τιμές AD στο διάστημα 1, 8 τιμές στο διάστημα 2 και 7 τιμές στο διάστημα 3, τότε η κανονικοποιημένη συχνότητα για το διάστημα 1 θα ήταν  $5/(5 + 8 + 7) = 0.25$ , για το διάστημα 2 θα ήταν  $8/(5 + 8 + 7) = 0.4$ , και για το διάστημα 3 θα ήταν  $7/(5 + 8 + 7) = 0.35$ . Επιλέξαμε τη χρήση της κανονικοποιημένης συχνότητας για να εξαλείψουμε την πόλωση (bias) που μπορεί να προκύψει μεταξύ αποθετηρίων που έχουν διαφορετικό αριθμό πακέτων (ή κλάσεων). Καθώς τα αποθετήρια είναι μεταξύ αυτών με τα περισσότερα stars και forks, οι διαφορές στο πλήθος των πακέτων (ή των κλάσεων) δεν αντικατοπτρίζουν απαραίτητα διαφορές το βαθμό επαναχρησιμοποιησιμότητας των τμημάτων κώδικα.

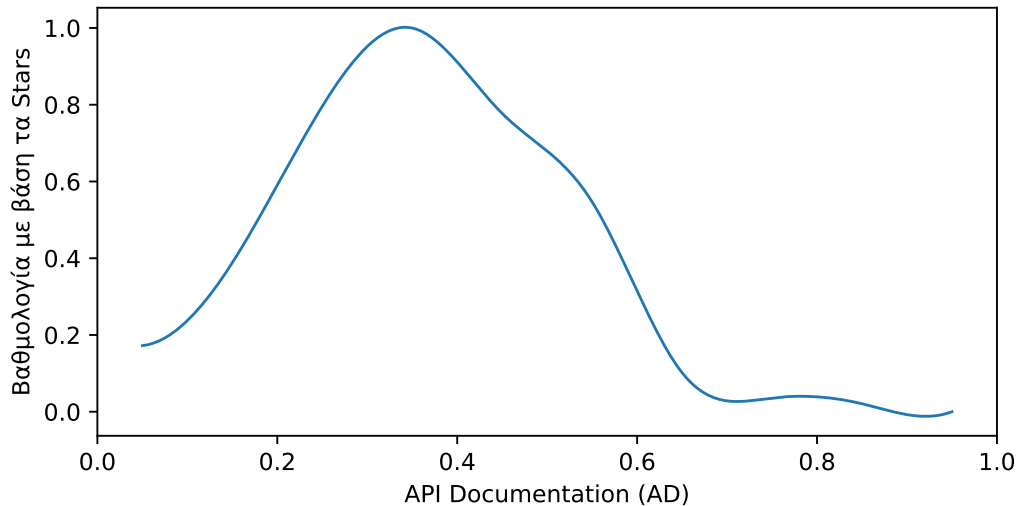
Στο Σχήμα 10.4 φαίνονται τα αποτελέσματα της εφαρμογής της εξίσωσης (10.1), δηλαδή της βαθμολογίας επαναχρησιμοποιησιμότητας με βάση τα stars, για τις τιμές της μετρικής AD σε επίπεδο πακέτου. Σημειώστε επίσης ότι το άνω 5% και το κάτω 5% των τιμών αφαιρούνται σε αυτή τη φάση καθώς αποτελούν ακραίες τιμές (outliers). Όπως φαίνεται σε αυτό το Σχήμα, η βαθμολογία επαναχρησιμοποιησιμότητας με βάση τη μετρική AD είναι μέγιστη για τιμές AD στο διάστημα [0.25, 0.5].

Σε αυτή τη φάση, έχουμε υπολογίσει μια βαθμολογία επαναχρησιμοποιησιμότητας με βάση τα stars και μια βαθμολογία επαναχρησιμοποιησιμότητας με βάση τα forks για κάθε μετρική. Η τελική βαθμολογία επαναχρησιμοποιησιμότητας για κάθε τμήμα κώδικα (κλάση ή πακέτο) δίνεται από τις παρακάτω εξισώσεις:

$$RS_{Final} = \frac{\sum_{j=1}^k RS_{metric}(j) \cdot corr(metric_j, stars)}{\sum_{j=1}^k corr(metric_j, stars)} \quad (10.4)$$

$$RF_{Final} = \frac{\sum_{j=1}^k RF_{metric}(j) \cdot corr(metric_j, forks)}{\sum_{j=1}^k corr(metric_j, forks)} \quad (10.5)$$

$$Final_{Score} = \frac{3 \cdot RF_{Final} + RS_{Final}}{4} \quad (10.6)$$



Σχήμα 10.4: Βαθμολογία Επαναχρησιμοποιησιμότητας με βάση τα Stars συναρτήσει των Τιμών της Μετρικής AD

όπου το  $k$  είναι ο αριθμός των μετρικών σε κάθε επίπεδο (κλάσης ή πακέτου). Τα  $RS_{Final}$  και  $RF_{Final}$  είναι τα τελικές βαθμολογίες με βάση τα stars και με βάση τα forks, αντίστοιχα. Οι τιμές  $RS_{metric}(j)$  and  $RF_{metric}(j)$  αναφέρονται στις βαθμολογίες για τη  $j$ -η μετρική ( $metric$ ), οι οποίες δίνονται από τις εξισώσεις (10.1) and (10.2), ενώ τα  $corr(metric_j, stars)$  και  $corr(metric_j, forks)$  είναι οι τιμές συσχέτισης Pearson (Pearson correlation coefficient) που προκύπτουν μεταξύ των τιμών της  $j$ -ης μετρική και των stars ή των forks, αντίστοιχα. Το  $FinalScore$  αναφέρεται στην τελική τιμή της επαναχρησιμοποιησιμότητας και υπολογίζεται ως ο σταθμισμένος μέσος όρος (weighted average) της τιμής της επαναχρησιμοποιησιμότητας με βάση τα stars και της τιμής επαναχρησιμοποιησιμότητας με βάση τα forks. Μεγαλύτερο βάρος (3 αντί 1) δίνεται στην τιμή με βάση τα forks καθώς αφορά περισσότερα χαρακτηριστικά ποιότητας που σχετίζονται με την επαναχρησιμοποιησιμότητα (βλέπε Πίνακα 10.1).

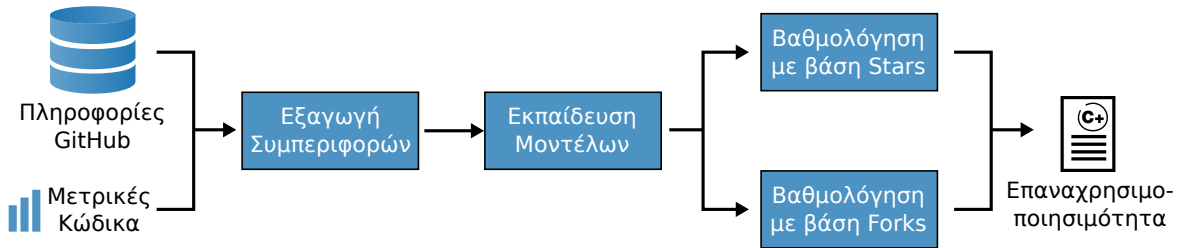
## 10.5 Σύστημα Αξιολόγησης Επαναχρησιμοποιησιμότητας

Σε αυτό το υποκεφάλαιο προτείνουμε μια μεθοδολογία για την εκτίμηση της επαναχρησιμοποιησιμότητας λογισμικού και δείχνουμε πώς μπορούμε να κατασκευάσουμε κατάλληλα μοντέλα.

### 10.5.1 Επισκόπηση Συστήματος

Η ροή δεδομένων του συστήματός μας απεικονίζεται στο Σχήμα 10.5. Η είσοδος είναι ένα σύνολο από μετρικές στατικής ανάλυσης, μαζί με τις σχετικές πληροφορίες (stars και forks) από αποθετήρια του GitHub. Καθώς το σύστημα περιλαμβάνει την αξιολόγηση της επαναχρησιμοποιησιμότητας σε επίπεδο κλάσης και σε επίπεδο πακέτου, εκπαιδεύουμε ένα μοντέλο για κάθε μετρική και για κάθε επίπεδο, με βάση την ανάλυση που παρουσιάστηκε στο προηγούμενο υποκεφάλαιο. Η έξοδος κάθε μοντέλου παρέχει μια βαθμολογία για την επαναχρησιμοποιησιμότητα που προκύπτει από την τιμή της αντίστοιχης μετρικής για την

κλάση (ή το πακέτο) που εξετάζεται. Στη συνέχεια, όλες οι επιμέρους βαθμολογίες συναθροίζονται για τον υπολογισμό της τελικής τιμής της επαναχρησιμοποιησιμότητας που είναι σχετική με το βαθμό στον οποίο μια κλάση (ή ένα πακέτο) επιλέγεται από τους προγραμματιστές.



Σχήμα 10.5: Επισκόπηση του Συστήματος Αξιολόγησης της Επαναχρησιμοποιησιμότητας

## 10.5.2 Κατασκευή Μοντέλων

Για την κατασκευή αποτελεσματικών μοντέλων για τη συμπεριφορά της κάθε μετρικής, επιλέγουμε να συγκρίνουμε μεταξύ διάφορων αλγορίθμων μηχανικής μάθησης. Συγκεκριμένα συγκρίνουμε τρεις τεχνικές: την *Παλινδρόμηση με Support Vector Machines (Support Vector Regression)*, την *Πολυωνυμική Παλινδρόμηση (Polynomial Regression)* και την *Παλινδρόμηση με Random Forest (Random Forest Regression)*. Τα μοντέλα με Random Forest κατασκευάστηκαν χρησιμοποιώντας τη μέθοδο bagging, ενώ για τα μοντέλα με Support Vector Machines επιλέχθηκε ο πυρήνας RBF (Radial Basis Function kernel). Όσον αφορά τα πολυωνυμικά μοντέλα, για τη διαδικασία προσαρμογής στις καμπύλες (fitting), ο βέλτιστος βαθμός του πολυωνύμου επιλέχθηκε χρησιμοποιώντας τη μέθοδο του γόνατου (elbow method) στο άθροισμα των τετραγώνων των σφαλμάτων<sup>1</sup>. Σε περιπτώσεις που ο αριθμός των διαστημάτων (bins), και συνεπώς ο αριθμός των σημείων δεδομένων εκπαίδευσης, είναι μικρός, χρησιμοποιήσαμε γραμμική παρεμβολή (linear interpolation) μέχρις ότου το σύνολο δεδομένων για κάθε μοντέλο να περιέχει 60 σημεία.

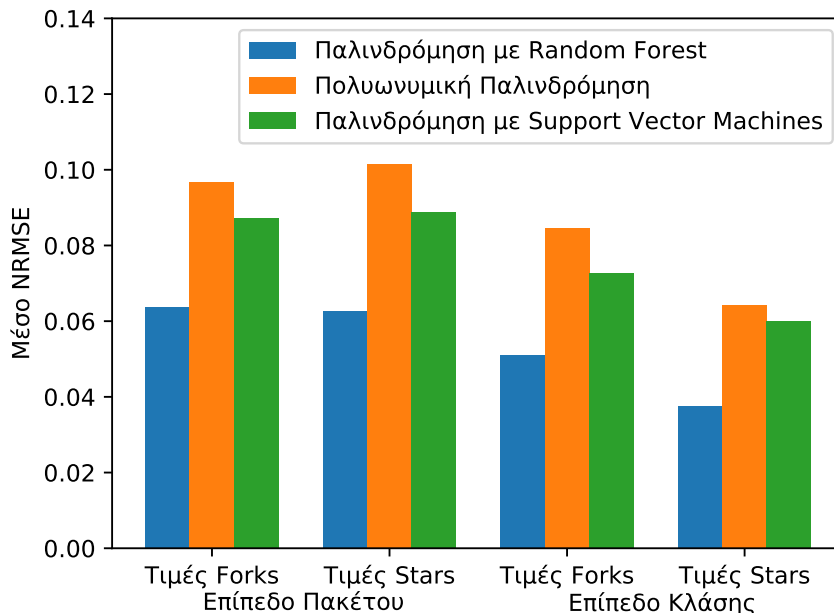
Για την αξιολόγηση των μοντέλων και για να επιλέξουμε αυτά που μοντελοποιούν καλύτερα τις συμπεριφορές των μετρικών, χρησιμοποιούμε τη κανονικοποιημένη εκδοχή της μετρικής της τετραγωνικής ρίζας του μέσου τετραγωνικού σφάλματος (Normalized Root Mean Squared Error - NRMSE). Με βάση τις πραγματικές τιμές (actual scores)  $y_1, y_2, \dots, y_n$ , και τις τιμές που εκτιμήθηκαν (predicted scores)  $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$ , η μετρική NRMSE υπολογίζεται ως:

$$NRMSE = \sqrt{\frac{1}{N} \cdot \frac{\sum_{i=1}^N (\hat{y}_i - y_i)^2}{\sum_{i=1}^N (\bar{y} - y_i)^2}} \quad (10.7)$$

όπου  $\bar{y}$  είναι ο μέσος όρος των πραγματικών τιμών και  $N$  είναι ο αριθμός των δειγμάτων στο σύνολο δεδομένων. Η μετρική NRMSE επιλέχθηκε καθώς λαμβάνει υπόψη τη μέση διαφορά μεταξύ των πραγματικών τιμών και των τιμών που εκτιμήθηκαν, ενώ ταυτόχρονα είναι στην ίδια τάξη μεγέθους με αυτές τις τιμές.

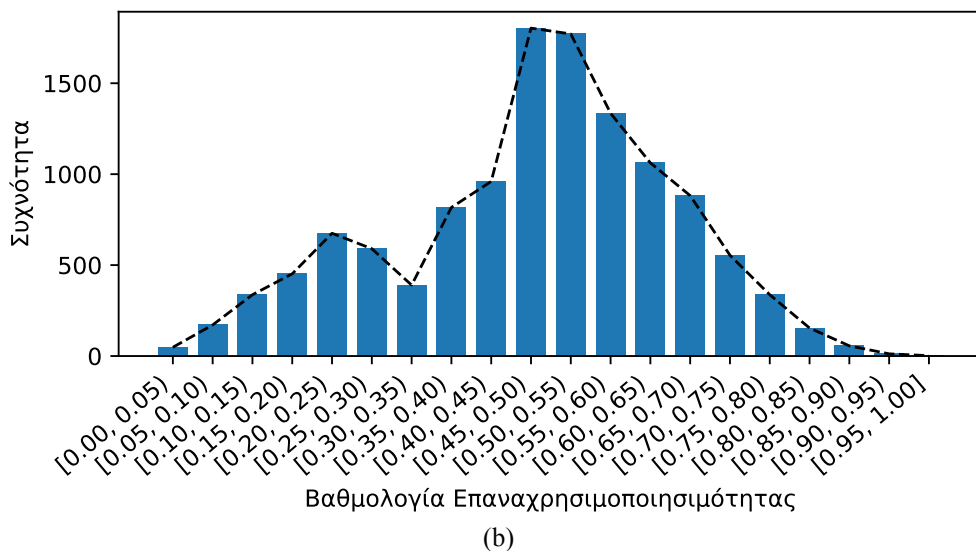
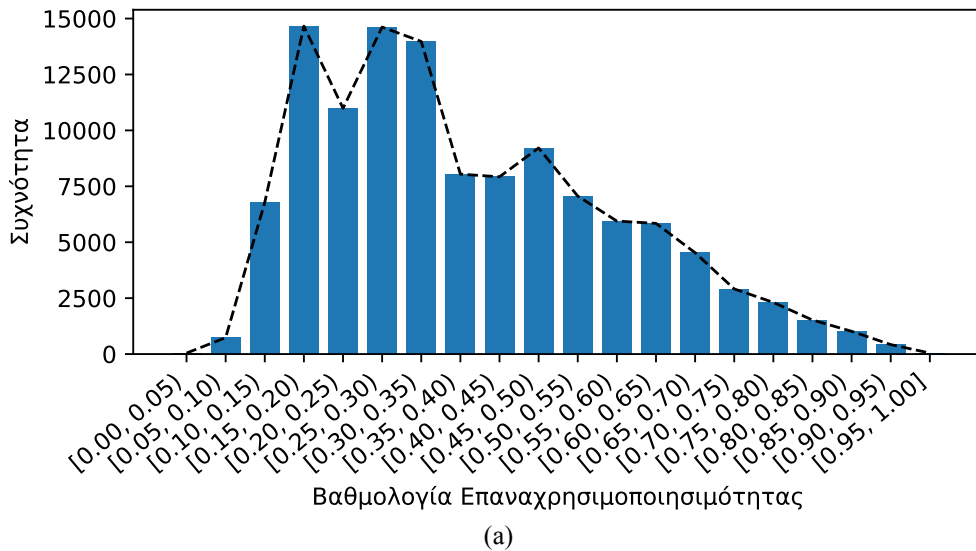
<sup>1</sup>Σύμφωνα με αυτή τη μέθοδο, ο βέλτιστος βαθμός είναι αυτός για τον οποίο δεν υπάρχει σημαντική μείωση στο άθροισμα των τετραγώνων των σφαλμάτων όταν υπάρχει αύξηση κατά ένα βαθμό.

Στο Σχήμα 10.6 απεικονίζεται το μέσο NRMSE για τους τρεις αλγορίθμους, όσον αφορά την εκτίμηση των τιμών της επαναχρησιμοποιησιμότητας (με βάση τα stars και τα forks) σε επίπεδο κλάσης και σε επίπεδο πακέτου. Παρόλο που και οι τρεις προσεγγίσεις είναι αποτελεσματικές (καθώς έχουν χαμηλές τιμές NRMSE), το Random Forest υπερέχει σαφώς των δύο άλλων αλγορίθμων και στις τέσσερις κατηγορίες, ενώ η Πολυωνυμική Παλινδρόμηση έχει το μεγαλύτερο σφάλμα (error). Ένα άλλο ενδιαφέρον συμπέρασμα είναι ότι η εκτίμηση της επαναχρησιμοποιησιμότητας σε επίπεδο πακέτου καταλήγει σε μεγαλύτερο σφάλμα και για τους τρεις αλγορίθμους. Αυτό βασικά είναι ένα αποτέλεσμα της μεθοδολογίας εξαγωγής των συμπεριφορών των μετρικών, για την οποία το ground truth παρέχεται σε επίπεδο αποθετηρίου (repository). Καθώς τα πακέτα κάθε αποθετηρίου είναι περισσότερα από τις κλάσεις του, οι συμπεριφορές που εξάγονται σε επίπεδο πακέτου είναι λιγότερο ανθεκτικές στον θόρυβο και συνεπώς οδηγούν σε υψηλότερα σφάλματα. Τέλος, εφόσον το Random Forest είναι πιο αποτελεσματικό από τους άλλους δυο αλγορίθμους, το επιλέγουμε ως το βασικό αλγόριθμο για τη συνέχεια.



Σχήμα 10.6: Μέσο NRMSE για τους τρεις αλγορίθμους μηχανικής μάθησης

Τα Σχήματα 10.7a και 10.7b απεικονίζουν τις κατανομές της βαθμολογίας επαναχρησιμοποιησιμότητας σε επίπεδο κλάσης και σε επίπεδο πακέτου, αντίστοιχα. Όπως είναι αναμενόμενο, η βαθμολογία και στις δύο περιπτώσεις ακολουθεί μια κατανομή που είναι παρόμοια με την κανονική (normal distribution), ενώ οι τιμές για τις περισσότερες εγγραφές συγκεντρώνονται ομοιόμορφα γύρω από το 0.5. Για την επαναχρησιμοποιησιμότητα σε επίπεδο κλάσης, παρατηρούμε ότι έχουμε θετική ασυμμετρία (positive/left-sided skewness). Εξετάζοντας περαιτέρω τις κλάσεις που έλαβαν βαθμολογίες στο διάστημα  $[0.2, 0.25]$ , καταλήγουμε ότι περιέχουν λιγοστή πληροφορία (π.χ. οι περισσότερες από αυτές έχουν λιγότερες από 10 γραμμές κώδικα), επομένως έχουν χαμηλές τιμές επαναχρησιμοποιησιμότητας.



Σχήμα 10.7: Κατανομή της βαθμολογίας επαναχρησιμοποιησιμότητας (a) σε επίπεδο κλάσης, και (b) σε επίπεδο πακέτου

## 10.6 Αξιολόγηση

Χρησιμοποιούμε δύο τύπους έργων για να αξιολογήσουμε τη μεθοδολογία μας: έργα που κατασκευάστηκαν από προγραμματιστές (human-generated) και έργα που έχουν παραχθεί αυτοματοποιημένα από κάποιο εργαλείο (auto-generated). Συγκεκριμένα, χρησιμοποιούμε τα 5 έργα που φαίνονται στον Πίνακα 10.3, 3 εκ των οποίων ανακτήθηκαν από το GitHub και 2 που κατασκευάστηκαν αυτοματοποιημένα χρησιμοποιώντας τα εργαλεία του S-CASE<sup>2</sup>. Από τη σύγκριση των αποτελεσμάτων του συστήματος σε αυτούς τους δύο τύπους έργων, αναμένουμε να εξάγουμε ενδιαφέροντα συμπεράσματα. Τα human-generated έργα είναι αρκετά πιθανό να παρουσιάζουν υψηλές αποκλίσεις (deviations) στη βαθμολογία επαναχρησιμοποιησιμότητας, καθώς περιέχουν διαφορετικά σύνολα τμημάτων κώδικα. Αντίθετα, τα auto-generated είναι υπηρεσίες RESTful, για τις οποίες η δομή του κώδικα

<sup>2</sup><http://s-case.github.io/>



παράγεται αυτοματοποιημένα, κι έτσι αναμένεται να έχουν υψηλές τιμές επαναχρησιμοποιησιμότητας και χαμηλές αποκλίσεις. Επιπρόσθετα, όπως φαίνεται και στον Πίνακα 10.3, τα έργα διαφέρουν αρκετά ως προς το μέγεθος του (αριθμός κλάσεων και πακέτων), συνεπώς μπορούν να χρησιμοποιηθούν για την αξιολόγηση του κατά πόσο το μοντέλο μας έχει δυνατότητα γενίκευσης σε τμήματα κώδικα από διαφορετικά έργα.

Πίνακας 10.3: Στατιστικά του Συνόλου Δεδομένων Αξιολόγησης

Όνομα Έργου/Αποθετηρίου	Τύπος	#Πακέτα	#Κλάσεις
realm/realm-java	Human-generated	137	3859
liferay/liferay-portal	Human-generated	1670	3941
spring-projects/spring-security	Human-generated	543	7099
Webmarks	Auto-generated	9	27
WSAT	Auto-generated	20	127

Στις ακόλουθες ενότητες, παρέχεται μια ανάλυση για την εκτίμηση της επαναχρησιμοποιησιμότητας αυτών των έργων καθώς και ένα παράδειγμα εκτίμησης για συγκεκριμένες κλάσεις και πακέτα.

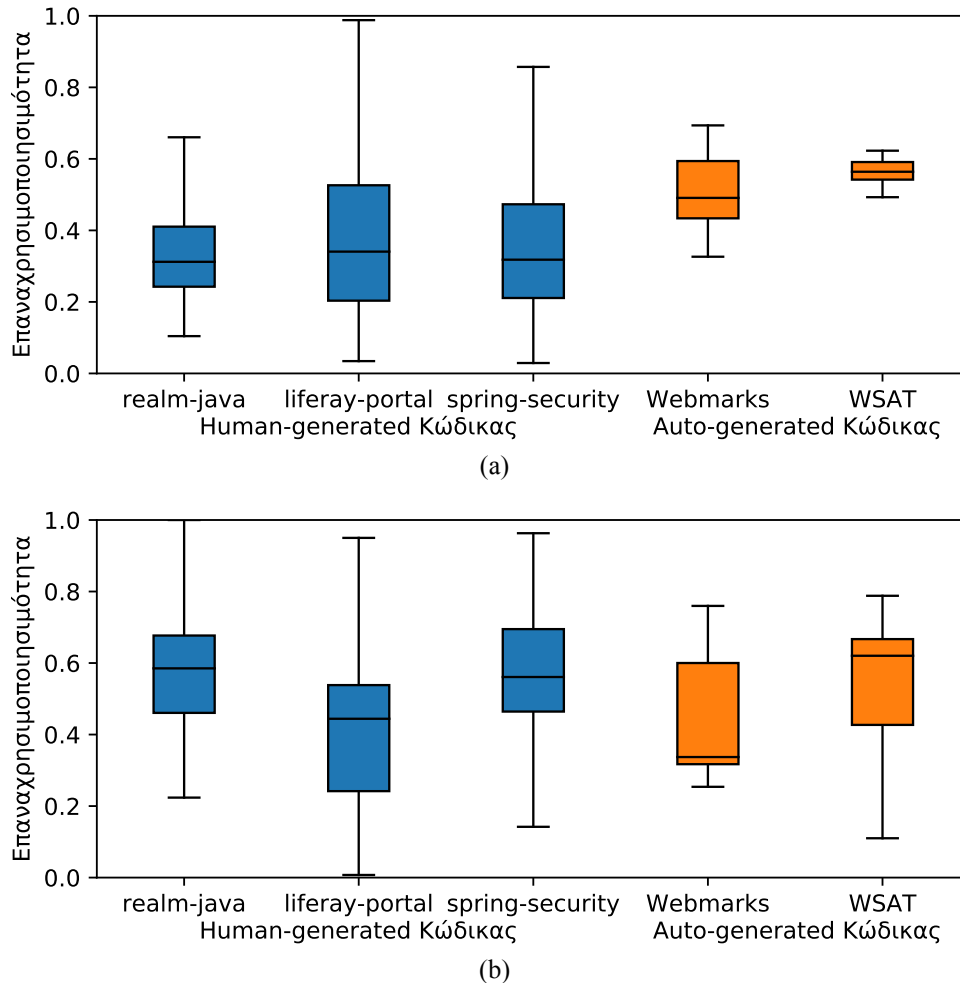
### 10.6.1 Αξιολόγηση Μοντέλου Επαναχρησιμοποιησιμότητας

Τα Σχήματα 10.8a και 10.8b απεικονίζουν τις κατανομές των βαθμολογιών επαναχρησιμοποιησιμότητας για όλα τα έργα σε επίπεδο κλάσης και σε επίπεδο πακέτου, αντίστοιχα. Τα boxplots μπλε χρώματος αναφέρονται σε έργα που αναπτύχθηκαν από προγραμματιστές, ενώ αυτά που έχουν πορτοκαλί χρώμα αναφέρονται σε έργα που αναπτύχθηκαν αυτοματοποιημένα.

Κατ' αρχάς, είναι σαφές ότι η διακύμανση (variance) των τιμών επαναχρησιμοποιησιμότητας είναι υψηλότερη στα έργα προγραμματιστών από ότι στα έργα που αναπτύχθηκαν αυτοματοποιημένα. Αυτό είναι αναμενόμενο καθώς τα auto-generated έργα έχουν σωστή αρχιτεκτονική και υψηλά επίπεδα αφάιρησης (abstraction). Τα δύο συγκεκριμένα έργα που αναλύουμε (Webmarks και WSAT) έχουν επίσης παρόμοιες τιμές επαναχρησιμοποιησιμότητας, που οφείλεται στο ότι έχουν παραχθεί από το ίδιο εργαλείο. Συνεπώς, έχουν παρόμοια χαρακτηριστικά κάτι το οποίο αντικατοπτρίζεται και στις τιμές των μετρικών στατικής ανάλυσης.

Η υψηλή διακύμανση για τα human-generated έργα υποδηλώνει ότι η μεθοδολογία μας είναι ικανή να προσδιορίσει τμήματα κώδικα υψηλής και χαμηλής επαναχρησιμοποιησιμότητας. Έργα όπως αυτά που αναλύουμε συνήθως περιλαμβάνουν αρκετά τμήματα κώδικα, κάποια εκ των οποίων έχουν γραφτεί έχοντας κατά νου την επαναχρησιμοποίηση (π.χ. APIs) και κάποια χωρίς αντίστοιχη απαίτηση. Κάτι άλλο που φαίνεται να ενισχύει τη διακύμανση είναι ο αριθμός των προγραμματιστών που εργάζονται σε κάθε έργο, που φυσικά δεν επηρεάζει τις τιμές των έργων που δημιουργούνται αυτόματα. Τέλος, με βάση τις κατανομές επαναχρησιμοποιησιμότητας για όλα τα έργα, καταλήγουμε ότι το σύστημα δεν επηρεάζεται απροσδόκητα από το μέγεθος των έργων. Παρά το γεγονός ότι ο αριθμός των κλάσεων και ο αριθμός των πακέτων αλλάζει από έργο σε έργο (υπάρχει έργο με 9 πακέτα και 27 κλάσεις, και έργο με 1670 πακέτα και 7099 κλάσεις), η τιμή της επαναχρησιμοποιησιμότητας δεν επηρεάζεται σε επίπεδο τμήματος κώδικα.





Σχήμα 10.8: Boxplots που απεικονίζουν τις κατανομές επαναχρησιμοποιησιμότητας για 3 έργα που αναπτύχθηκαν από προγραμματιστές, δηλαδή έχουν human-generated κώδικα (■), και 2 έργα που αναπτύχθηκαν αυτοματοποιημένα, δηλαδή έχουν auto-generated κώδικα (■), (a) σε επίπεδο κλάσης και (b) σε επίπεδο πακέτου

### 10.6.2 Παράδειγμα Εκτίμησης Επαναχρησιμοποιησιμότητας

Για να αξιολογήσουμε περαιτέρω την εγκυρότητα των βαθμολογιών του συστήματός μας, εξετάσαμε τις μετρικές στατικής ανάλυσης κάποιων ενδεικτικών κλάσεων και πακέτων για να ελέγξουμε αν συμβαδίζουν με τις βαθμολογίες επαναχρησιμοποιησιμότητας. Στον Πίνακα 10.4 παρέχεται ένα υποσύνολο των μετρικών στατικής ανάλυσης για κάποια αντιπροσωπευτικά παραδείγματα κλάσεων και πακέτων με διαφορετικές τιμές επαναχρησιμοποιησιμότητας. Συγκεκριμένα, ο Πίνακας περιέχει μετρικές για δύο κλάσεις και δύο πακέτα που έλαβαν υψηλές και χαμηλές βαθμολογίες επαναχρησιμοποιησιμότητας.

Εξετάζοντας τις τιμές των μετρικών, παρατηρούμε ότι η εκτίμηση της επαναχρησιμοποιησιμότητας είναι και στις τέσσερις περιπτώσεις λογική, ενώ συμφωνεί με τη σχετική βιβλιογραφία όσον αφορά την επίδραση των μετρικών στις ιδιότητες ποιότητας που σχετίζονται με την επαναχρησιμοποιησιμότητα (βλέπε Πίνακα 10.1). Όσον αφορά την κλάση που έλαβε υψηλή βαθμολογία επαναχρησιμοποιησιμότητας, φαίνεται να είναι καλά τεκμηριωμένη (η τιμή της μετρικής Comments Density - CD είναι 20.31%), που υποδεικνύει ότι είναι κατάλληλη για επαναχρησιμοποίηση. Επίσης, το μέγεθός της (σύμφωνα με τη μετρική

Πίνακας 10.4: Τιμές Μετρικών για Κλάσεις και Πακέτα με διαφορετική Επαναχρησιμοποιησιμότητα

Μετρικές	Κλάσεις		Πακέτα	
	Κλάση 1	Κλάση 2	Πακέτο 1	Πακέτο 2
WMC	14	12	–	–
CBO	12	3	–	–
LCOM5	2	11	–	–
CD (%)	20.31%	10.2%	41.5%	0.0%
RFC	12	30	–	–
LOC	84	199	2435	38
TNCL	–	–	8	2
Επαναχρησιμοποιησιμότητα	95.78%	10.8%	90.59%	16.75%

Lines of Code - LOC) είναι αναμενόμενο και σχετικά μικρό, κάτι που συνεισφέρει στην κατανοησιμότητα και επομένως την επαναχρησιμοποιησιμότητά της. Συνεπώς, η βαθμολογία για αυτήν την κλάση είναι αρκετά σωστή. Από την άλλη πλευρά, η κλάση που έλαβε χαμηλή βαθμολογία φαίνεται να έχει χαμηλή συνοχή (η τιμή της μετρικής LCOM5 είναι σχετικά υψηλή) και υψηλή σύζευξη (η τιμή της μετρικής RFC είναι σχετικά υψηλή). Αυτές οι ιδιότητες είναι κρίσιμες για τα χαρακτηριστικά ποιότητας που σχετίζονται με την επαναχρησιμοποιησιμότητα, οπότε η χαμηλή βαθμολογία της κλάσης αυτής είναι αναμενόμενη.

Εξετάζοντας τις τιμές των μετρικών για τα πακέτα, τα συμπεράσματα είναι παρόμοια. Το πακέτο που έλαβε υψηλή βαθμολογία επαναχρησιμοποιησιμότητας φαίνεται να έχει καλή τεκμηρίωση (με βάση την τιμή της μετρικής CD) και τυπικό μέγεθος, και άρα είναι κατάλληλο για επαναχρησιμοποίηση. Από την άλλη πλευρά, το πακέτο που έλαβε χαμηλή βαθμολογία φαίνεται να μην έχει αξιοποιήσιμη πληροφορία καθώς το μέγεθός του είναι πολύ μικρό (συνολικά 38 γραμμές κώδικα).

## 10.7 Συμπεράσματα

Σε αυτό το κεφάλαιο, προτείναμε μια μεθοδολογία εκτίμησης ποιότητας κατά την οποία η ποιότητα όπως την αντιλαμβάνεται ο χρήστης (user-perceived quality) χρησιμοποιείται ως μέτρο της επαναχρησιμοποιησιμότητας ενός τμήματος λογισμικού. Χρησιμοποιώντας τα stars και forks από αποθετήρια GitHub για την κατασκευή ενός συνόλου δεδομένων ground truth, κατασκευάσαμε ένα σύστημα που παρέχει μια βαθμολογία επαναχρησιμοποιησιμότητας για κλάσεις και πακέτα. Η αξιολόγησή μας υποδεικνύει ότι αυτή η προσέγγιση είναι αποτελεσματική για την εκτίμηση της επαναχρησιμοποιησιμότητας.

Ως ιδέα για μελλοντική εργασία, θα μπορούσαμε να διερευνήσουμε τη βαθμολογία επαναχρησιμοποιησιμότητας στο πλαίσιο κάποιου συγκεκριμένου τομέα εφαρμογής. Επιπρόσθετα, σκοπεύουμε να εστιάσουμε στη δημιουργία μοντέλων για διαφορετικούς άξονες ποιότητας [150, 254]. Τέλος, θα ήταν χρήσιμο να αξιολογήσουμε το σύστημά μας σε ρεαλιστικά σενάρια επαναχρησιμοποίησης, πιθανώς συμπεριλαμβάνοντας προγραμματιστές, για να επιβεβαιώσουμε περαιτέρω την αποτελεσματικότητά του.

**Μέρος**



**ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΗ**

**ΕΡΓΑΣΙΑ**



# 11

## Συμπεράσματα

Το κύριο αντικείμενο και ο σκοπός αυτής της διατριβής ήταν η εφαρμογή μιας σειράς μεθοδολογιών για τη διευκόλυνση των φάσεων ανάπτυξης του λογισμικού μέσω της επαναχρησιμοποίησης. Σε αυτό το πλαίσιο, εφαρμόσαμε διάφορες τεχνικές για να επιτρέψουμε την επαναχρησιμοποίηση των απαιτήσεων και των προδιαγραφών, να υποστηρίξουμε την διαδικασία ανάπτυξης λογισμικού και να επικεντρωθούμε επιπλέον στην αξιολόγηση της ποιότητας υπό το πρίσμα της επαναχρησιμοποίησης.

Στο Μέρος II εστίασαμε στις προκλήσεις της μοντελοποίησης και της εξόρυξης απαιτήσεων λογισμικού (requirements modeling and mining). Η μεθοδολογία μας για τη μοντελοποίηση απαιτήσεων παρέχει ένα επεκτάσιμο μοντέλο που επιτρέπει την αποθήκευση και την ευρετηριοποίηση απαιτήσεων λογισμικού. Το μοντέλο μας υποστηρίζει την εξαγωγή αντικειμένων από απαιτήσεις διαφορετικών τύπων, όπως π.χ. από λειτουργικές απαιτήσεις και διαγράμματα UML. Οι μεθοδολογίες που προτείναμε για την εξόρυξη απαιτήσεων λογισμικού αναδεικνύουν περαιτέρω τις δυνατότητες του μοντέλου των οντολογιών μας. Επίσης, οι προσεγγίσεις εξόρυξης που εφαρμόσαμε αποδείχθηκαν αποτελεσματικές για την επαναχρησιμοποίηση σε επίπεδο απαιτήσεων. Μια άλλη σημαντική συνεισφορά σε αυτό το επίπεδο είναι ο μηχανισμός εξαγωγής προδιαγραφών από απαιτήσεις που μπορούν στη συνέχεια να χρησιμεύσουν ως βάση για την ανάπτυξη της εφαρμογής. Επιπρόσθετα, αυτή η σύνδεση απαιτήσεων με προδιαγραφές, την οποία διερευνήσαμε σε ένα σενάριο υπηρεσιών RESTful, είναι αμφίδρομη και ανιχνεύσιμη.

Στο Μέρος III εστίασαμε στην περιοχή της εξόρυξης πηγαίου κώδικα (source code mining) προκειμένου να υποστηρίξουμε την επαναχρησιμοποίηση σε διάφορα επίπεδα. Αρχικά παρουσιάσαμε μια μεθοδολογία για την ανάκτηση πηγαίου κώδικα, την αποθήκευση και την ευρετηριοποίησή του, ώστε να είναι δυνατή η εφαρμογή τεχνικών εξόρυξης. Στη συνέχεια, αντιμετωπίσαμε τις διάφορες προκλήσεις που προκύπτουν μέσα στο πλαίσιο της επαναχρησιμοποίησης κώδικα. Αρχικά, παρουσιάσαμε ένα σύστημα επαναχρησιμοποίησης τμημάτων λογισμικού, το οποίο λαμβάνει ένα ερώτημα σε μορφή διεπαφής (επιτρέποντας έτσι την εύκολη δημιουργία ερωτημάτων) και ανακτά και κατατάσσει τα τμήματα λογισμικού χρησιμοποιώντας ένα μοντέλο με βάση τη σύνταξη (syntax-aware). Τα αποτελέσματα αναλύονται επιπλέον για να αξιολογηθεί κατά πόσο συμφωνούν με τις απαιτήσεις του προγραμματιστή

(χρησιμοποιώντας ελέγχους) καθώς και για να εξαχθούν χρήσιμες πληροφορίες, όπως η ροή ελέγχου του κώδικα. Η δεύτερη συνεισφορά μας είναι στην περιοχή της εξόρυξης μικρών τμημάτων κώδικα (snippet mining) που αναφέρονται σε διαφορετικά APIs. Η μεθοδολογία μας σε αυτήν την περίπτωση αντιμετωπίζει την πρόκληση της εύρεσης χρήσιμων τμημάτων κώδικα για συχνά προγραμματιστικά ερωτήματα. Συγκεκριμένα, κατεβάζουμε snippets από διαφορετικές διαδικτυακές πηγές και τα ομαδοποιούμε για να μπορεί στη συνέχεια ο προγραμματιστής να διακρίνει εύκολα τις διαφορετικές υλοποιήσεις (διαφορετικά APIs). Μια σημαντική ιδέα που παρουσιάστηκε σε αυτό τον άξονα ήταν η κατάταξη των υλοποιήσεων και των τμημάτων κώδικα σύμφωνα με την προτίμηση των προγραμματιστών, που διευκολύνει περαιτέρω την εύρεση των πιο συχνά προτιμώμενων τμημάτων κώδικα. Τέλος, εστίασαμε στην περαιτέρω βελτίωση των επιλεγμένων υλοποιήσεων (ή γενικά του πηγαίου κώδικα του προγραμματιστή), βρίσκοντας παρόμοιες λύσεις σε διαδικτυακές κοινότητες. Η συνεισφορά μας σε αυτό το πλαίσιο είναι μια μεθοδολογία για την εύρεση λύσεων που ανταποκρίνονται σε παρόμοια προβλήματα με αυτά που παρουσιάζονται στον προγραμματιστή. Μια σημαντική πτυχή της μεθοδολογίας μας ήταν ο σχεδιασμός μιας τεχνικής ομοιότητας που αξιοποιεί όλα τα στοιχεία των αναρτήσεων-ερωτήσεων (question posts), συμπεριλαμβανομένων των τίτλων, των ετικετών, του κειμένου και του πηγαίου κώδικα, τα οποία δείξαμε ότι περιέχουν χρήσιμες πληροφορίες.

Στο μέρος IV εστίασαμε την επαναχρησιμοποιησιμότητα κώδικα (code reusability), στο κατά πόσο δηλαδή τα τμήματα κώδικα είναι κατάλληλα για επαναχρησιμοποίηση. Αρχικά, κατασκευάσαμε ένα σύστημα για να αναδείξουμε τις εφαρμογές της αξιολόγησης της επαναχρησιμοποιησιμότητας. Συγκεκριμένα, το σύστημά μας επικεντρώνεται στην επαναχρησιμοποίηση τμημάτων κώδικα και δείχνει πως οι προγραμματιστές χρειάζεται να αξιολογούν τον κώδικα όχι μόνο από λειτουργική άποψη αλλά και σε σχέση με την ποιότητά του. Ως εκ τούτου, προτείναμε μια μεθοδολογία για την αξιολόγηση της επαναχρησιμοποιησιμότητας τμημάτων λογισμικού. Η μεθοδολογία μας βασίζεται στην ιδέα ότι τα τμήματα κώδικα που συνήθως προτιμούνται από τους προγραμματιστές είναι πιο πιθανό να έχουν υψηλή ποιότητα. Ως μέρος της συνεισφοράς μας σε αυτήν την περιοχή, δείξαμε αρχικά πώς η δημοτικότητα του έργου (προτίμηση από τους προγραμματιστές) συσχετίζεται με την επαναχρησιμοποιησιμότητα, καθώς και τον τρόπο με τον οποίο μπορούν να μοντελοποιηθούν αυτές οι έννοιες χρησιμοποιώντας μετρικές στατικής ανάλυσης. Όπως αποδείξαμε, τα μοντέλα μας είναι σε θέση να αξιολογούν τμήματα του λογισμικού χωρίς να απαιτείται εξωτερική βοήθεια από κάποιον ειδικό.

Ως τελικό σχόλιο, η παρούσα διατριβή ανέδειξε τρόπους με τους οποίους η επαναχρησιμοποίηση μπορεί να διευκολύνει όλες τις φάσεις του κύκλου ζωής της ανάπτυξης λογισμικού. Δείξαμε πώς μπορεί κανείς να εφαρμόσει τεχνικές εξόρυξης σε δεδομένα τεχνολογίας λογισμικού από πολλαπλές πηγές και σε διαφορετικές μορφές, με σκοπό την παροχή επαναχρησιμοποιούμενων λύσεων στο πλαίσιο του καθορισμού απαιτήσεων και εξαγωγής προδιαγραφών, της ανάπτυξης λογισμικού και της αξιολόγησης ποιότητας. Οι μεθοδολογίες μας καλύπτουν πολλά διαφορετικά σενάρια, το σημαντικότερο όμως είναι ότι προσφέρουν μια ολοκληρωμένη λύση. Η ερευνητική μας συνεισφορά μπορεί να θεωρηθεί ως το μέσο για τη δημιουργία καλύτερου λογισμικού με την επαναχρησιμοποίηση όλων των διαθέσιμων πληροφοριών. Και όπως ήδη αναφέρθηκε στην εισαγωγή αυτής της διατριβής, η δημιουργία καλύτερου λογισμικού με ελάχιστο κόστος και προσπάθεια έχει άμεσο αντίκτυπο στις οικονομικές και κοινωνικές πτυχές της καθημερινής ζωής.

# 12

## Μελλοντική Εργασία

Όπως ήδη αναφέρθηκε στο προηγούμενο κεφάλαιο, οι συνεισφορές μας αφορούν διαφορετικές φάσεις του κύκλου ζωής της ανάπτυξης λογισμικού. Σε αυτό το κεφάλαιο συζητάμε πιθανές μελλοντικές επεκτάσεις σχετικά με τις συνεισφορές μας, καθώς και σχετικά με τις σχετικές ερευνητικές περιοχές<sup>1</sup>. Αναλύουμε αρχικά κάθε περιοχή από μόνη της, συνδέοντάς την με το αντίστοιχο μέρος της διατριβής, και στη συνέχεια εστιάζουμε στο μελλοντική εργασία για το σύνολο της διατριβής.

Για τη μοντελοποίηση και την εξόρυξη απαιτήσεων (μέρος II) υπάρχουν ιδέες για μελλοντικές επεκτάσεις σε διάφορες κατευθύνσεις. Όσον αφορά το τμήμα της μοντελοποίησης, η κύρια πρόκληση είναι να δημιουργηθεί ένα μοντέλο που θα περιλαμβάνει τη διαδικασία καθορισμού απαιτήσεων στο σύνολό της. Ένα τέτοιο μοντέλο θα μπορούσε να περιλαμβάνει απαιτήσεις, προτιμήσεις από τα ενδιαφερόμενα μέρη, μη λειτουργικά χαρακτηριστικά των έργων κ.λπ. Η προσέγγισή μας σε αυτό το πλαίσιο μπορεί να θεωρηθεί ως ένα πρώτο βήμα, καθώς αποθηκεύει και ευρετηριοποιεί κάποια από αυτά τα δεδομένα, ενώ είναι επίσης κατάλληλη για την εφαρμογή τεχνικών εξόρυξης με σκοπό την επαναχρησιμοποίηση. Ομοίως, και όσον αφορά επίσης και την εξόρυξη απαιτήσεων, μια ενδιαφέρουσα μελλοντική κατεύθυνση είναι η υποστήριξη για διαφορετικούς τύπους απαιτήσεων (π.χ. ιστορίες χρηστών, διαφορετικοί τύποι διαγραμμάτων κ.α.) και η εφαρμογή τεχνικών εξόρυξης με σκοπό την επαναχρησιμοποίηση καθώς και την εξαγωγή προδιαγραφών. Στο πλαίσιο της εξαγωγής προδιαγραφών, ένα άλλο σημαντικό σημείο που πρέπει να εξεταστεί ως μελλοντική εργασία είναι η σύνδεση των απαιτήσεων με την υλοποίηση. Η εργασία μας σε αυτόν τον άξονα, που επικεντρώνεται σε υπηρεσίες RESTful, χρησιμεύει ως ένα πρακτικό παράδειγμα και μπορεί να αποτελέσει τη βάση για την περαιτέρω διερεύνηση αυτής της σύνδεσης.

Όσον αφορά την εξόρυξη πηγαίου κώδικα (μέρος III), οι μελλοντικές ερευνητικές κατευθύνσεις είναι σχετικές με τις προκλήσεις που αναλύθηκαν. Μια από αυτές τις προκλήσεις είναι η επαναχρησιμοποίηση τμημάτων κώδικα, για την οποία προτείναμε μια μεθοδολογία που αξιολογεί κατά πόσο τα τμήματα κώδικα που ανακτώνται καλύπτουν τη λειτουργικότητα που απαιτείται από το ερώτημα του προγραμματιστή. Ορισμένες ενδιαφέρουσες ιδέες

<sup>1</sup>Μια σημαντική σημείωση για τον αναγνώστη είναι ότι συγκεκριμένες μελλοντικές προτάσεις για τις τεχνικές μας παρέχονται στο τέλος κάθε κεφαλαίου. Ως εκ τούτου, εστιάζουμε εδώ στη μελλοντική εργασία για τις σχετικές ερευνητικές περιοχές σε ένα πιο γενικό επίπεδο.

προς αυτήν την κατεύθυνση είναι η υποστήριξη για μεγαλύτερα τμήματα κώδικα (π.χ. με πολλές κλάσεις) καθώς και η αυτοματοποιημένη εξαγωγή εξαρτήσεων για κάθε τμήμα κώδικα. Στην ιδανική περίπτωση, θα θέλαμε ένα σύστημα που θα μπορούσε να ανακτήσει πηγαίο κώδικα από το διαδίκτυο, να ελέγξει εάν είναι κατάλληλος για τις αντίστοιχες απαιτήσεις και να τον ενσωματώσει στον κώδικα του προγραμματιστή λαμβάνοντας υπόψη και τις εξαρτήσεις από εξωτερικές βιβλιοθήκες. Ομοίως, για το σενάριο επαναχρησιμοποίησης snippet/API, μια μελλοντική κατεύθυνση θα ήταν να ενσωματωθεί το snippet στον πηγαίο κώδικα του προγραμματιστή, ενδεχομένως μετά τη συνόψισή του (summarization) για να εξασφαλιστεί ότι είναι βέλτιστο. Τέλος, μια ενδιαφέρουσα επέκταση σχετικά με την εύρεσης λύσεων σε διαδικτυακές κοινότητες θα ήταν η ενσωμάτωση σημασιολογίας (semantics). Με αυτόν τον τρόπο, θα μπορούσαμε να σχηματίσουμε μια σημασιολογική σύνδεση μεταξύ του ερωτήματος (που εκφράζει κάποια λειτουργικά κριτήρια) και των σχετικών τμημάτων κώδικα. Τέτοιες συνδέσεις θα μπορούσαν να αξιοποιηθούν για τη βελτίωση των τεχνικών εξόρυξης τμημάτων κώδικα ή ακόμα και για την αυτοματοποιημένη τεκμηρίωση ή την εξαγωγή παραδειγμάτων χρήσης για API βιβλιοθηκών.

Οι ερευνητικές κατευθύνσεις για την αξιολόγηση της ποιότητας (μέρος IV) είναι επίσης αρκετές. Καταρχάς, θα ήταν ενδιαφέρον να διερευνηθεί περαιτέρω η ενσωμάτωση μετρικών ποιότητας σε συστήματα επαναχρησιμοποίησης πηγαίου κώδικα. Με αυτόν τον τρόπο, θα μπορεί να παρέχεται στον προγραμματιστή η καλύτερη δυνατή λύση, μία λύση δηλαδή που καλύπτει τη λειτουργικότητα που απαιτείται από το ερώτημα, ενώ ταυτόχρονα έχει υψηλή ποιότητα και προτιμάται σε μεγάλο βαθμό από την κοινότητα των προγραμματιστών. Σε αυτό το πλαίσιο, θα ήταν χρήσιμο να διερευνήσουμε πώς μπορούν να χρησιμοποιηθούν διαφορετικές μετρικές ποιότητας και μοντέλα για την αξιολόγηση τμημάτων πηγαίου κώδικα. Θεωρώντας τη μεθοδολογία αξιολόγησης της επαναχρησιμοποιησιμότητας που παρουσιάστηκε ως σημείο εκκίνησης, θα μπορούσαμε να διερευνήσουμε περαιτέρω τη σχέση μεταξύ της προτίμησης των προγραμματιστών και της επαναχρησιμοποιησιμότητας με πολλούς τρόπους. Θα μπορούσαμε, για παράδειγμα να μετρήσουμε το πραγματικό ποσοστό επαναχρησιμοποίησης κάθε βιβλιοθήκης (ή τμήματος) λογισμικού, το οποίο θα μπορούσε να εξαχθεί εφαρμόζοντας τεχνικές εξόρυξης στις εισαγωγές βιβλιοθηκών (imports) των έργων από υπηρεσίες φιλοξενίας κώδικα. Επιπλέον, η σχέση αυτήν θα μπορούσε να διερευνηθεί σε διάφορους τομείς: για παράδειγμα, θα περίμενε κανείς ότι οι βιβλιοθήκες γενικής χρήσης επαναχρησιμοποιούνται συχνότερα από τις βιβλιοθήκες μηχανικής μάθησης, που υποδεικνύει ότι η προτίμηση τους πρέπει να μετράται χρησιμοποιώντας διαφορετική βάση.

Τέλος, σε αυτή τη διατριβή διερευνήσαμε την έννοια της επαναχρησιμοποίησης λογισμικού ως βασικό άξονα για την ανάπτυξη λογισμικού. Μια ενδιαφέρουσα επέκταση που καλύπτει τη διαδικασία ανάπτυξης θα ήταν η μοντελοποίηση και ευρετηριοποίηση τμημάτων λογισμικού διαφορετικών μεγεθών, μαζί με τις απαιτήσεις τους, τον πηγαίο κώδικα, καθώς και όλες τις σχετικές πληροφορίες. Στη συνέχεια, θα ήταν δυνατό, πιθανώς με χρήση σημασιολογικών τεχνικών, να παρέχονται επαναχρησιμοποιήσιμες λύσεις σε διαφορετικά επίπεδα (π.χ. απαιτήσεις, πηγαίος κώδικας, τεκμηρίωση, κ.λπ.). Αν επιπλέον οι συνδέσεις μεταξύ των επιπέδων ήταν ανιχνεύσιμες, θα μπορούσαμε να επαναχρησιμοποιούμε τμήματα λογισμικού, ενώ ταυτόχρονα η ενσωμάτωσή τους θα ανανεώνει όλα τα σχετικά σημεία του συστήματος (π.χ. σχετικές απαιτήσεις). Έτσι, θα ήταν δυνατή η παραγωγή αποδοτικών και συντηρήσιμων προϊόντων λογισμικού. Επομένως, ως τελική παρατήρηση, μπορούμε να πούμε ότι αυτή η διατριβή αποτελεί ένα πρώτο βήμα προς αυτή την κατεύθυνση.



## **ΒΙΒΛΙΟΓΡΑΦΙΑ ΚΑΙ ΛΙΣΤΑ ΔΗΜΟΣΙΕΥΣΕΩΝ**



# Βιβλιογραφία

- [1] Robert Curley. *Architects of the Information Age*. Britannica Educational Publishing, 2011.
- [2] Ian Sommerville. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010.
- [3] Standish Group. *The CHAOS Report (1994)*. Technical Report. Standish Group, 1994.
- [4] Standish Group. *The CHAOS Report (2015)*. Technical Report. Standish Group, 2015.
- [5] Vicente Rodríguez Montequín, Sonia Cousillas Fernández, Francisco Ortega Fernández, and Joaquín Villanueva Balsera. “Analysis of the Success Factors and Failure Causes in Projects: Comparison of the Spanish Information and Communication Technology ICT Sector”. In: *Int. J. Inf. Technol. Proj. Manag.* 7.1 (2016), pp. 18–31.
- [6] Daniel J. Power. *A Brief History of Decision Support Systems*. Available online: <http://dssresources.com/history/dsshhistory.html> [Retrieved: November, 2017]. 2007.
- [7] Richard Miller Devens. *Cyclopaedia of Commercial and Business Anecdotes: Comprising Interesting Reminiscences and Facts, Remarkable Traits and Humors ... of Merchants, Traders, Bankers ... Etc. in All Ages and Countries*. D. Appleton & Company, 1868.
- [8] Ahmed E. Hassan and Tao Xie. “Software Intelligence: The Future of Mining Software Engineering Data”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 161–166.
- [9] Malcolm D. McIlroy. “Mass-produced Software Components”. In: *Software Engineering; Report of a Conference sponsored by the NATO Science Committee*. Ed. by P. Naur and B. Randell. Available online: <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF> [Retrieved: , ]. NATO Scientific Affairs Division, Brussels, Belgium. Garmisch, Germany, 1968, pp. 138–155.
- [10] Lombard Hill Group. *What is Software Reuse?* 2017.
- [11] William B. Frakes and Kyo Kang. “Software Reuse Research: Status and Future”. In: *IEEE Trans. Softw. Eng.* 31.7 (2005), pp. 529–536.
- [12] Charles W. Krueger. “Software Reuse”. In: *ACM Comput. Surv.* 24.2 (1992), pp. 131–183.

- [13] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. 3rd. Boca Raton, FL, USA: CRC Press, Inc., 2014.
- [14] Martin Robillard, Robert Walker, and Thomas Zimmermann. “Recommendation Systems for Software Engineering”. In: *IEEE Softw.* 27.4 (2010), pp. 80–86.
- [15] Roger Pressman. *Software Engineering: A Practitioner’s Approach*. 7th ed. New York, NY, USA: McGraw-Hill, Inc., 2010.
- [16] SL Pfleeger and B Kitchenham. “Software quality: The elusive target”. In: *IEEE Software* (1996), pp. 12–21.
- [17] *ISO/IEC 25010:2011*. 2011. Available online: <https://www.iso.org/standard/35733.html> [Retrieved: November, 2017].
- [18] IEEE. *IEEE Standards Collection: Software Engineering, IEEE Standard 610.12-1990*. Technical Report. IEEE, 1993.
- [19] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering: Theory and Practice*. 4th ed. Pearson, 2009.
- [20] Stefan Franck. *Going Agile - Does it work for all?* Available online: <https://www.netcentric.biz/blog/Does-Agile-Work-For-The-Enterprise.html> [Retrieved: August, 2017]. 2017.
- [21] Liesbeth Dusink and Jan van Katwijk. “Reuse Dimensions”. In: *SIGSOFT Softw. Eng. Notes* 20.SI (1995), pp. 137–149.
- [22] Johannes Sametinger. *Software Engineering with Reusable Components*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.
- [23] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.
- [24] Barthélemy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. “Moving into a New Software Project Landscape”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 275–284.
- [25] Susan Elliott Sim and Rosalva E. Gallardo-Valencia. *Finding Source Code on the Web for Remix and Reuse*. Springer Publishing Company, Incorporated, 2015.
- [26] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’09. Boston, MA, USA: ACM, 2009, pp. 1589–1598.
- [27] H. Li, Z. Xing, X. Peng, and W. Zhao. “What help do developers seek, when and how?” In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 142–151.
- [28] Jeff Atwood. *Introducing Stackoverflow.com*. Available online: <https://blog.codinghorror.com/introducing-stackoverflow-com/> [Retrieved: April, 2008]. 2008.

- [29] Stefan Franck. *None of Us is as Dumb as All of Us*. Available online: <https://blog.codinghorror.com/stack-overflow-none-of-us-is-as-dumb-as-all-of-us/> [Retrieved: September, 2008]. 2008.
- [30] David A. Garvin. "What Does 'Product Quality' Really Mean?" In: *MIT Sloan Management Review* 26.1 (1984).
- [31] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values*. HarperTorch, 1974.
- [32] Witold Suryn. *Software Quality Engineering: A Practitioner's Approach*. Wiley-IEEE Press, 2014.
- [33] Jim A. McCall, Paul K. Richards, and Gene F. Walters. *Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality*. Technical Report ADA0490-14. General Electric Co, Sunnyvale, CA, 1977.
- [34] Jim A. McCall, Paul K. Richards, and Gene F. Walters. *Factors in Software Quality. Volume II. Metric Data Collection and Validation*. Technical Report ADA049014. General Electric Co, Sunnyvale, CA, 1977.
- [35] Jim A. McCall, Paul K. Richards, and Gene F. Walters. *Factors in Software Quality. Volume III. Preliminary Handbook on Software Quality for an Acquisition Manager*. Technical Report ADA049014. General Electric Co, Sunnyvale, CA, 1977.
- [36] ISO/IEC. *ISO/IEC 9126:1991*. Technical Report. ISO/IEC, 1991.
- [37] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [38] Pierre Bourque and Richard E. Fairley, eds. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014.
- [39] Thomas J. McCabe. "A Complexity Measure". In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 407–.
- [40] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.
- [41] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.
- [42] Brian Henderson-Sellers. *Object-oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
- [43] Fernando Brito e Abreu and Rogério Carapuça. "Candidate Metrics for Object-oriented Software Within a Taxonomy Framework". In: *J. Syst. Softw.* 26.1 (1994), pp. 87–96.
- [44] Mark Lorenz and Jeff Kidd. *Object-oriented Software Metrics: A Practical Guide*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.
- [45] KP Srinivasan and T Devi. "A comprehensive review and analysis on object-oriented software metrics in software measurement". In: *International Journal on Computer Science and Engineering* 6.7 (2014), p. 247.

- [46] Jitender Kumar Chhabra and Varun Gupta. “A Survey of Dynamic Software Metrics”. In: *J. Comput. Sci. Technol.* 25.5 (2010), pp. 1016–1029.
- [47] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson. “Dynamic Metrics for Object Oriented Designs”. In: *Proceedings of the 6th International Symposium on Software Metrics. METRICS '99*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 50–.
- [48] Erik Arisholm, Lionel C. Briand, and Audun Foyen. “Dynamic Coupling Measurement for Object-Oriented Software”. In: *IEEE Trans. Softw. Eng.* 30.8 (2004), pp. 491–506.
- [49] Norman Schneidewind. *Systems and software engineering with applications*. New York, NY: Institute of Electrical and Electronics Engineers, 2009.
- [50] Allan J. Albrecht. “Measuring Application Development Productivity”. In: *Proc. of IBM Application Development Symp.* Ed. by I. B. M. Press. 1979, pp. 83–92.
- [51] T. Capers Jones. *Estimating Software Costs*. Hightstown, NJ, USA: McGraw-Hill, Inc., 1998.
- [52] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. “An Analysis of Developer Metrics for Fault Prediction”. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering. PROMISE '10*. Timisoara, Romania: ACM, 2010, 18:1–18:9.
- [53] Robert D. Austin. *Measuring and Managing Performance in Organizations*. New York: Dorset House Publishing, 1996.
- [54] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction”. In: *Proceedings of the 30th International Conference on Software Engineering. ICSE '08*. Leipzig, Germany: ACM, 2008, pp. 181–190.
- [55] Ahmed E. Hassan. “Predicting Faults Using the Complexity of Code Changes”. In: *Proceedings of the 31st International Conference on Software Engineering. ICSE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.
- [56] Diomidis Spinellis. *Code Quality: The Open Source Perspective (Effective Software Development Series)*. Addison-Wesley Professional, 2006.
- [57] Barry W. Boehm. *Software Engineering Economics*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.
- [58] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011.
- [59] Boris Beizer. *Software Testing Techniques*. 2nd. New York, NY, USA: Van Nostrand Reinhold Co., 1990.
- [60] Thomas P. Hughes. *American Genesis: A Century of Invention and Technological Enthusiasm, 1870-1970, 2nd edition*. Chicago, IL, USA: University Of Chicago Press, 2004.
- [61] James S. Huggins. *First Computer Bug*. 2000.

- [62] Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach*. 1st. Springer Publishing Company, Incorporated, 2010.
- [63] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Norwood, MA, USA: Artech House, Inc., 2003.
- [64] Beck. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [65] Kent Beck. *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
- [66] Steve McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.
- [67] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [68] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.
- [69] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of Data Mining*. Cambridge, MA, USA: MIT Press, 2001.
- [70] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.
- [71] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- [72] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.
- [73] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. 2nd. Wiley Publishing, 2007.
- [74] Jiawei Han, Jian Pei, and Micheline Kamber. *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [75] Dursun Delen and Haluk Demirkan. "Data, Information and Analytics As Services". In: *Decis. Support Syst.* 55.1 (2013), pp. 359–363.
- [76] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.
- [77] A. R. Webb. *Statistical Pattern Recognition*. 2nd ed. Wiley, 2002.
- [78] Ryan J. Urbanowicz and Will N. Browne. *Introduction to Learning Classifier Systems*. SpringerBriefs in Intelligent Systems. New York, NY, USA: Springer New York Inc., 2017.
- [79] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. "Data Mining for Software Engineering". In: *Computer* 42.8 (2009), pp. 55–62.

- [80] Martin Monperrus. *Data-mining for Software Engineering*. Available online: <https://www.monperrus.net/martin/data-mining-software-engineering> [Retrieved: December, 2013]. 2013.
- [81] M. Halkidi, D. Spinellis, G. Tsatsaronis, and M. Vazirgiannis. “Data Mining in Software Engineering”. In: *Intell. Data Anal.* 15.3 (2011), pp. 413–441.
- [82] Manish Kumar, Nirav Ajmeri, and Smita Ghaisas. “Towards Knowledge Assisted Agile Requirements Evolution”. In: *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*. RSSE ’10. Cape Town, South Africa: ACM, 2010, pp. 16–20.
- [83] S. Ghaisas and N. Ajmeri. “Knowledge-Assisted Ontology-Based Requirements Evolution”. In: *Managing Requirements Knowledge*. Ed. by Walid Maalej and Anil Kumar Thurimella. Springer Berlin Heidelberg, 2013, pp. 143–167.
- [84] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. “An Approach to Constructing Feature Models Based on Requirements Clustering”. In: *Proceedings of the 13th IEEE International Conference on Requirements Engineering*. RE ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 31–40.
- [85] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. “An Exploratory Study of Information Retrieval Techniques in Domain Analysis”. In: *Proceedings of the 2008 12th International Software Product Line Conference*. SPLC ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 67–76.
- [86] William Frakes, Ruben Prieto-Diaz, and Christopher Fox. “DARE: Domain Analysis and Reuse Environment”. In: *Ann. Softw. Eng.* 5 (1998), pp. 125–141.
- [87] Alexander Felfernig, Monika Schubert, Monika Mandl, Francesco Ricci, and Walid Maalej. “Recommendation and Decision Technologies for Requirements Engineering”. In: *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*. RSSE ’10. Cape Town, South Africa: ACM, 2010, pp. 11–15.
- [88] Walid Maalej and Anil Kumar Thurimella. “Towards a Research Agenda for Recommendation Systems in Requirements Engineering”. In: *Proceedings of the 2009 Second International Workshop on Managing Requirements Knowledge*. MARK ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 32–39.
- [89] David Binkley. “Source Code Analysis: A Road Map”. In: *2007 Future of Software Engineering*. FOSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 104–119.
- [90] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. “An Unabridged Source Code Dataset for Research in Software Reuse”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 339–342.



- [91] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. “Sourcerer: A Search Engine for Open Source Code Supporting Structure-based Search”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 681–682.
- [92] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. “Sourcerer: Mining and Searching Internet-scale Software Repositories”. In: *Data Min. Knowl. Discov.* 18.2 (2009), pp. 300–336.
- [93] Reid Holmes and Gail C. Murphy. “Using Structural Context to Recommend Source Code Examples”. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: ACM, 2005, pp. 117–125.
- [94] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. “Jungloid Mining: Helping to Navigate the API Jungle”. In: *SIGPLAN Not.* 40.6 (2005), pp. 48–61.
- [95] Suresh Thummalapenta and Tao Xie. “Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web”. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: ACM, 2007, pp. 204–213.
- [96] Oliver Hummel, Werner Janjic, and Colin Atkinson. “Code Conjurer: Pulling Reusable Software out of Thin Air”. In: *IEEE Softw.* 25.5 (2008), pp. 45–52.
- [97] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. “CodeGenie: A Tool for Test-driven Source Code Search”. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 917–918.
- [98] Steven P. Reiss. “Semantics-based Code Search”. In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 243–253.
- [99] Mehrdad Nurolohzade, Robert J. Walker, and Frank Maurer. “An Assessment of Test-Driven Reuse: Promises and Pitfalls”. In: *Safe and Secure Software Reuse*. Ed. by John Favaro and Maurizio Morisio. Vol. 7925. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 65–80.
- [100] Tao Xie and Jian Pei. “MAPO: Mining API Usages from Open Source Repositories”. In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR ’06. Shanghai, China: ACM, 2006, pp. 54–57.
- [101] Naiyana Sahavechaphan and Kajal Claypool. “XSnippet: Mining for Sample Code”. In: *SIGPLAN Not.* 41.10 (2006), pp. 413–430.
- [102] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. “Example Overflow: Using Social Media for Code Recommendation”. In: *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. RSSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 38–42.

- [103] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. “SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization”. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST '12. Cambridge, Massachusetts, USA: ACM, 2012, pp. 219–228.
- [104] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. *Building Bing Developer Assistant*. Technical Report MSR-TR-2015-36. Microsoft Research, 2015.
- [105] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. “A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution”. In: *J. Softw. Maint. Evol.* 19.2 (2007), pp. 77–131.
- [106] Tiago L Alves, Christiaan Ypma, and Joost Visser. “Deriving metric thresholds from benchmark data”. In: *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM. IEEE. 2010, pp. 1–10.
- [107] Kecia A. M. Ferreira, Mariza A. S. Bigonha, Roberto S. Bigonha, Luiz F. O. Mendes, and Heitor C. Almeida. “Identifying Thresholds for Object-oriented Software Metrics”. In: *Journal of Systems and Software* 85.2 (2012), pp. 244–257.
- [108] Shi Zhong, Taghi M Khoshgoftaar, and Naeem Seliya. “Unsupervised Learning for Expert-Based Software Quality Estimation.” In: *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering*. HASE'04. Tampa, Florida, 2004, pp. 149–155.
- [109] David Hovemeyer, Jaime Spacco, and William Pugh. “Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs”. In: *SIGSOFT Softw. Eng. Notes* 31.1 (2005), pp. 13–19.
- [110] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. “Using Static Analysis to Find Bugs”. In: *IEEE Softw.* 25.5 (2008), pp. 22–29.
- [111] Claire Le Goues and Westley Weimer. “Measuring code quality to improve specification mining”. In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 175–190.
- [112] Hironori Washizaki, Rieko Namiki, Tomoyuki Fukuoka, Yoko Harada, and Hiroyuki Watanabe. “A framework for measuring and evaluating program source code quality”. In: *Proceedings of the 8th International Conference on Product-Focused Software Process Improvement*. PROFES. Springer. Riga, Latvia, 2007, pp. 284–299.
- [113] Teresa Cai, Michael R Lyu, Kam-Fai Wong, and Mabel Wong. “ComPARE: A generic quality assessment environment for component-based software systems”. In: *Proceedings of the 2001 International Symposium on Information Systems and Engineering*. ISE'2001. Las Vegas, USA, 2001.
- [114] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
- [115] Luisa Mich, Franch Mariangela, and Novi Inverardi Pierluigi. “Market research for requirements analysis using linguistic tools”. In: *Requirements Engineering* 9.1 (2004), pp. 40–56.

- [116] Barry Boehm and Victor R. Basili. “Software defect reduction top 10 list”. In: *Computer* 34 (1 2001), pp. 135–137.
- [117] H. Kaindl, M. Smialek, D. Svetinovic, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, H. Schwarz, D. Bildhauer, J. P. Brogan, K. S. Mukasa, K. Wolter, and T. Krebs. *Requirements specification language definition: Defining the ReD-SeeDS languages, Deliverable D2.4.1*. Public Deliverable. ReDSeeDS (Requirements Driven Software Development System) Project, 2007.
- [118] Michal Smialek. “Facilitating Transition from Requirements to Code with the ReD-SeeDS Tool”. In: *Proceedings of the 2012 IEEE 20th International Requirements Engineering Conference (RE)*. RE ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 321–322.
- [119] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.
- [120] Dan North. *JBehave: A framework for behaviour driven development*. 2003. Available online: <http://jbehave.org/> [Retrieved: March, 2015].
- [121] John Mylopoulos, Jaelson Castro, and Manuel Kolp. “Tropos: A Framework for Requirements-Driven Software Development”. In: *Information Systems Engineering: State of the Art and Research Themes*. Springer-Verlag, 2000, pp. 261–273.
- [122] Eric Siu-Kwong Yu. “Modelling Strategic Relationships for Process Reengineering”. PhD thesis. Toronto, Ont., Canada: University of Toronto, 1995.
- [123] Russell J. Abbott. “Program Design by Informal English Descriptions”. In: *Commun. ACM* 26.11 (1983), pp. 882–894.
- [124] Grady Booch. “Object-oriented Development”. In: *IEEE Trans. Softw. Eng.* 12.1 (1986), pp. 211–221.
- [125] Motoshi Saeki, Hisayuki Horai, and Hajime Enomoto. “Software Development Process from Natural Language Specification”. In: *Proceedings of the 11th International Conference on Software Engineering*. ICSE ’89. Pittsburgh, Pennsylvania, USA: ACM, 1989, pp. 64–73.
- [126] L. Mich. “NL-OOPS: From Natural Language to Object Oriented Requirements Using the Natural Language Processing System LOLITA”. In: *Nat. Lang. Eng.* 2.2 (1996), pp. 161–187.
- [127] H. M. Harmain and R. Gaizauskas. “CM-Builder: A Natural Language-Based CASE Tool for Object-Oriented Analysis”. In: *Automated Software Engg.* 10.2 (2003), pp. 157–181.
- [128] Verónica Castañeda, Luciana Ballejos, Ma. Laura Caliusco, and Ma. Rosa Galli. “The Use of Ontologies in Requirements Engineering”. In: *Global Journal of Researches In Engineering* 10.6 (2010).
- [129] Katja Siegemund, Edward J Thomas, Yuting Zhao, Jeff Pan, and Uwe Assmann. “Towards ontology-driven requirements engineering”. In: *Workshop semantic web enabled software engineering at 10th international semantic web conference (ISWC), Bonn*. 2011.

- [130] Hans-Jörg Happel and Stefan Seedorf. “Applications of Ontologies in Software Engineering”. In: *Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006)*. 2006, pp. 5–9.
- [131] Diego Dermeval, Jessyka Vilela, Iglbert Bittencourt, Jaelson Castro, Seiji Isotani, Patrick Brito, and Alan Silva. “Applications of ontologies in requirements engineering: a systematic review of the literature”. In: *Requirements Engineering* (2015), pp. 1–33.
- [132] Nanda Kambhatla. “Combining Lexical, Syntactic, and Semantic Features with Maximum Entropy Models for Extracting Relations”. In: *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions*. ACLdemo ’04. Barcelona, Spain: ACL, 2004, pp. 178–181.
- [133] Shubin Zhao and Ralph Grishman. “Extracting Relations with Integrated Information Using Kernel Methods”. In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. ACL ’05. Ann Arbor, Michigan: Association for Computational Linguistics, 2005, pp. 419–426.
- [134] Zhou GuoDong, Su Jian, Zhang Jie, and Zhang Min. “Exploring Various Knowledge in Relation Extraction”. In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. ACL ’05. Ann Arbor, Michigan: Association for Computational Linguistics, 2005, pp. 427–434.
- [135] Razvan Bunescu and Raymond J. Mooney. “Subsequence Kernels for Relation Extraction”. In: *Advances in Neural Information Processing Systems, Vol. 18: Proceedings of the 2005 Conference (NIPS)*. 2005, pp. 171–178.
- [136] Dmitry Zelenko, Chinatsu Aone, and Anthony Richardella. “Kernel Methods for Relation Extraction”. In: *J. Mach. Learn. Res.* 3 (2003), pp. 1083–1106.
- [137] Aron Culotta and Jeffrey Sorensen. “Dependency Tree Kernels for Relation Extraction”. In: *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. ACL ’04. Barcelona, Spain: Association for Computational Linguistics, 2004, pp. 423–429.
- [138] Razvan C. Bunescu and Raymond J. Mooney. “A Shortest Path Dependency Kernel for Relation Extraction”. In: *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Vancouver, British Columbia, Canada: Association for Computational Linguistics, 2005, pp. 724–731.
- [139] Nguyen Bach and Sameer Badaskar. “A Review of Relation Extraction”. Available online: <http://www.cs.cmu.edu/~textasciitilde{}nbach/papers/A-survey-on-Relation-Extraction.pdf> [Retrieved: , ]. 2007.
- [140] Christoforos Zolotas, Themistoklis Diamantopoulos, Kyriakos Chatzidimitriou, and Andreas Symeonidis. “From requirements to source code: a Model-Driven Engineering approach for RESTful web services”. In: *Automated Software Engineering* (2016), pp. 1–48.

- [141] Michael Roth, Themistoklis Diamantopoulos, Ewan Klein, and Andreas L. Symeonidis. “Software Requirements: A new Domain for Semantic Parsers”. In: *Proceedings of the ACL 2014 Workshop on Semantic Parsing*. SP14. Baltimore, Maryland, USA, 2014, pp. 50–54.
- [142] Themistoklis Diamantopoulos, Michael Roth, Andreas Symeonidis, and Ewan Klein. “Software Requirements as an Application Domain for Natural Language Processing”. In: *Language Resources and Evaluation 51.2* (2017), pp. 495–524.
- [143] Anders Björkelund, Love Hafdell, and Pierre Nugues. “Multilingual Semantic Role Labeling”. In: *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*. CoNLL ’09. Boulder, Colorado: Association for Computational Linguistics, 2009, pp. 43–48.
- [144] Bernd Bohnet. “Top Accuracy and Fast Dependency Parsing is not a Contradiction”. In: *Proceedings of the 23rd International Conference on Computational Linguistics*. Beijing, China, 2010, pp. 89–97.
- [145] R. E. Fan, K. W. Chang, C. J. Hsieh, X. R. Wang, and C. J. Lin. “LIBLINEAR: A library for large linear classification”. In: *Journal of Machine Learning Research 9* (2008), pp. 1871–1874.
- [146] V. R. Montequin, S. Cousillas, F. Ortega, and J. Villanueva. “Analysis of the Success Factors and Failure Causes in Information & Communication Technology (ICT) Projects in Spain”. In: *Procedia Technology 16* (2014), pp. 992–999.
- [147] Dean Leffingwell. “Calculating your return on investment from more effective requirements management”. In: *American Programmer 10.4* (1997), pp. 13–16.
- [148] Themistoklis Diamantopoulos, Klearchos Thomopoulos, and Andreas L. Symeonidis. “QualBoa: Reusability-aware Recommendations of Source Code Components”. In: *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*. MSR ’16. 2016, pp. 488–491.
- [149] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas L. Symeonidis. “User-Perceived Source Code Quality Estimation based on Static Analysis Metrics”. In: *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security*. QRS. Vienna, Austria, 2016, pp. 100–107.
- [150] Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. “Towards Modeling the User-perceived Quality of Source Code using Static Analysis Metrics”. In: *Proceedings of the 12th International Conference on Software Technologies - Volume 1*. ICSOFT. INSTICC. Setubal, Portugal: SciTePress, 2017, pp. 73–84.
- [151] Paulo Gomes, Pedro Gandola, and Joel Cordeiro. “Helping Software Engineers Re-using UML Class Diagrams”. In: *Proceedings of the 7th International Conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*. ICCBR ’07. Belfast, Northern Ireland, UK: Springer-Verlag, 2007, pp. 449–462.
- [152] Karina Robles, Anabel Fraga, Jorge Morato, and Juan Llorens. “Towards an Ontology-based Retrieval of UML Class Diagrams”. In: *Inf. Softw. Technol.* 54.1 (2012), pp. 72–86.

- [153] Han G. Woo and William N. Robinson. “Reuse of Scenario Specifications Using an Automated Relational Learner: A Lightweight Approach”. In: *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*. RE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 173–180.
- [154] William N. Robinson and Han G. Woo. “Finding Reusable UML Sequence Diagrams Automatically”. In: *IEEE Softw.* 21.5 (2004), pp. 60–67.
- [155] Daniel Bildhauer, Tassilo Horn, and Jurgen Ebert. “Similarity-driven Software Reuse”. In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. CVSM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 31–36.
- [156] Hamza Onoruoiza Salami and Moataz Ahmed. “Class Diagram Retrieval Using Genetic Algorithm”. In: *Proceedings of the 2013 12th International Conference on Machine Learning and Applications - Volume 02*. ICMLA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 96–101.
- [157] Udo Kelter, Jürgen Wehren, and Jorg Niere. “A Generic Difference Algorithm for UML Models.” In: *Software Engineering*. Ed. by Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke. Vol. 64. LNI. GI, 2005, pp. 105–116.
- [158] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. “Change Detection in Hierarchically Structured Information”. In: *SIGMOD Rec.* 25.2 (1996), pp. 493–504.
- [159] Y. Wang, D. J. DeWitt, and J. Y. Cai. “X-Diff: an effective change detection algorithm for XML documents”. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 519–530.
- [160] Themistoklis Diamantopoulos and Andreas Symeonidis. “Enhancing Requirements Reusability through Semantic Modeling and Data Mining Techniques”. In: *Enterprise Information Systems* (2017), pp. 1–22.
- [161] Horatiu Dumitru, Marek Gibiec, Negar Hariri, Jane Cleland-Huang, Bamshad Mobasher, Carlos Castro-Herrera, and Mehdi Mirakhorli. “On-demand Feature Recommendations Derived from Mining Public Product Descriptions”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Honolulu, HI, USA: ACM, 2011, pp. 181–190.
- [162] Soo Ling Lim and Anthony Finkelstein. “StakeRare: Using Social Networks and Collaborative Filtering for Large-Scale Requirements Elicitation”. In: *IEEE Trans. Softw. Eng.* 38.3 (2012), pp. 707–735.
- [163] Carlos Castro-Herrera, Chuan Duan, Jane Cleland-Huang, and Bamshad Mobasher. “Using Data Mining and Recommender Systems to Facilitate Large-Scale, Open, and Inclusive Requirements Elicitation Processes”. In: *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*. RE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 165–168.
- [164] Bamshad Mobasher and Jane Cleland-Huang. “Recommender Systems in Requirements Engineering”. In: *The AI magazine* 32.3 (2011), pp. 81–89.

- [165] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. “GroupLens: Applying Collaborative Filtering to Usenet News”. In: *Commun. ACM* 40.3 (1997), pp. 77–87.
- [166] Jose Romero-Mariona, Hadar Ziv, and Debra J. Richardson. “SRRS: A Recommendation System for Security Requirements”. In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. RSSE '08. Atlanta, Georgia: ACM, 2008, pp. 50–52.
- [167] M. C. Blok and J. L. Cybulski. “Reusing UML Specifications in a Constrained Application Domain”. In: *Proceedings of the Fifth Asia Pacific Software Engineering Conference*. APSEC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 196–.
- [168] Thomas A. Alspaugh, Annie I. Antón, Tiffany Barnes, and Bradford W. Mott. “An Integrated Scenario Management Strategy”. In: *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*. RE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 142–149.
- [169] Wei-Jin Park and Doo-Hwan Bae. “A Two-stage Framework for UML Specification Matching”. In: *Inf. Softw. Technol.* 53.3 (2011), pp. 230–244.
- [170] Belén Bonilla-Morales, Sérgio Crespo, and Clifton Clunie. “Reuse of Use Cases Diagrams: An Approach based on Ontologies and Semantic Web Technologies”. In: *Int. J. Comput. Sci.* 9.1 (2012), pp. 24–29.
- [171] George A. Miller. “WordNet: A Lexical Database for English”. In: *Commun. ACM* 38.11 (1995), pp. 39–41.
- [172] Mark Finlayson. “Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation”. In: *Proceedings of the Seventh Global Wordnet Conference*. Ed. by Heili Orav, Christiane Fellbaum, and Piek Vossen. Tartu, Estonia, 2014, pp. 78–85.
- [173] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. “WordNet::Similarity: Measuring the Relatedness of Concepts”. In: *Demonstration Papers at HLT-NAACL 2004*. HLT-NAACL–Demonstrations '04. Boston, Massachusetts: Association for Computational Linguistics, 2004, pp. 38–41.
- [174] Dekang Lin. “An Information-Theoretic Definition of Similarity”. In: *Proceedings of the Fifteenth International Conference on Machine Learning*. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 296–304.
- [175] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. “Mining Association Rules Between Sets of Items in Large Databases”. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93. Washington, D.C., USA: ACM, 1993, pp. 207–216.
- [176] Rakesh Agrawal and Ramakrishnan Srikant. “Fast Algorithms for Mining Association Rules in Large Databases”. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.

- [177] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. “Introduction to Algorithms, Third Edition”. In: 3rd. The MIT Press, 2009, pp. 390–396.
- [178] *Elasticsearch: RESTful, Distributed Search & Analytics*. 2016. Available online: <https://www.elastic.co/products/elasticsearch> [Retrieved: April, 2016].
- [179] *Java Compiler Tree API*. 2016. Available online: <http://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html> [Retrieved: April, 2016].
- [180] *Unicode Standard Annex #29, “Unicode Text Segmentation”, edited by Mark Davis. An integral part of The Unicode Standard*. 2016. Available online: <http://www.unicode.org/reports/tr29/> [Retrieved: April, 2016].
- [181] *CamelCase tokenizer, pattern analyzer, analyzers, Elasticsearch analysis*. 2016. Available online: [http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-pattern-analyzer.html#\\\_camelcase\\\_tokenizer](http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-pattern-analyzer.html#\_camelcase\_tokenizer) [Retrieved: April, 2016].
- [182] *Lucene’s Practical Scoring Function, Controlling Relevance, Search in Depth, Elasticsearch: The Definitive Guide*. 2016. Available online: <http://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html> [Retrieved: April, 2016].
- [183] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.
- [184] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. “Co-evolution of Project Documentation and Popularity Within Github”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 360–363.
- [185] S. Weber and J. Luo. “What Makes an Open Source Code Popular on GitHub?” In: *2014 IEEE International Conference on Data Mining Workshop*. ICDMW. 2014, pp. 851–855.
- [186] H. Borges, A. Hora, and M. T. Valente. “Understanding the Factors That Impact the Popularity of GitHub Repositories”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. ICSME. 2016, pp. 334–344.
- [187] Robert J. Walker. “Recent Advances in Recommendation Systems for Software Engineering”. In: *Recent Trends in Applied Artificial Intelligence*. Ed. by Moonis Ali, Tibor Bosse, Koen V. Hindriks, Mark Hoogendoorn, Catholijn M. Jonker, and Jan Treur. Vol. 7906. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 372–381.
- [188] Kim Mens and Angela Lozano. “Source Code-Based Recommendation Systems”. In: *Recommendation Systems in Software Engineering*. Ed. by Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 93–130.



- [189] Werner Janjic, Oliver Hummel, and Colin Atkinson. “Reuse-Oriented Code Recommendation Systems”. In: *Recommendation Systems in Software Engineering*. Ed. by Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 359–386.
- [190] Marko Gasparic and Andrea Janes. “What Recommendation Systems for Software Engineering Recommend”. In: *J. Syst. Softw.* 113.C (2016), pp. 101–113.
- [191] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. “CodeHint: Dynamic and Interactive Synthesis of Code Snippets”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 653–663.
- [192] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. “CodeGenie: Using Test-cases to Search and Reuse Source Code”. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 525–526.
- [193] Scott Henninger. “Retrieving Software Objects in an Example-based Programming Environment”. In: *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '91. Chicago, Illinois, USA: ACM, 1991, pp. 251–260.
- [194] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab, 1999.
- [195] Amir Michail. “Data Mining Library Reuse Patterns using Generalized Association Rules”. In: *Proceedings of the 22nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 167–176.
- [196] Yunwen Ye and Gerhard Fischer. “Supporting Reuse by Delivering Task-relevant and Personalized Information”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: ACM, 2002, pp. 513–523.
- [197] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. “Portfolio: Finding Relevant Functions and Their Usage”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Honolulu, HI, USA: ACM, 2011, pp. 111–120.
- [198] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. “Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications”. In: *IEEE Trans. Softw. Eng.* 38.5 (2012), pp. 1069–1087.
- [199] Oliver Hummel and Colin Atkinson. “Extreme Harvesting: test driven discovery and reuse of software components”. In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration*. IRI 2004. 2004, pp. 66–72.

- [200] Werner Janjic, Dietmar Stoll, Philipp Bostan, and Colin Atkinson. “Lowering the Barrier to Reuse Through Test-driven Search”. In: *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. SUITE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 21–24.
- [201] Oliver Hummel and Werner Janjic. “Test-Driven Reuse: Key to Improving Precision of Search Engines for Software Reuse”. English. In: *Finding Source Code on the Web for Remix and Reuse*. Ed. by SusanElliott Sim and RosalvaE. Gallardo-Valencia. Springer New York, 2013, pp. 227–250.
- [202] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masi-ero, and Cristina Lopes. “Applying Test-driven Code Search to the Reuse of Auxiliary Functionality”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 476–482.
- [203] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masi-ero, and Cristina Lopes. “A Test-driven Approach to Code Search and Its Application to the Reuse of Auxiliary Functionality”. In: *Inf. Softw. Technol.* 53.4 (2011), pp. 294–306.
- [204] Monika Krug. “FAST: An Eclipse Plug-in for Test-Driven Reuse”. MA thesis. University of Mannheim, 2007.
- [205] Steven P. Reiss. “Specifying What to Search for”. In: *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. SUITE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 41–44.
- [206] Themistoklis Diamantopoulos, Nikolaos Katirtzis, and Andreas Symeonidis. “Mantissa: A Recommendation System for Test-Driven Code Reuse”. 2018.
- [207] Ronald Rivest. *The MD5 Message-Digest Algorithm*. 1992.
- [208] D. Gale and L. S. Shapley. “College Admissions and the Stability of Marriage”. In: *American Mathematical monthly* 69.1 (1962), pp. 9–15.
- [209] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. 1st. O'Reilly Media, Inc., 2009.
- [210] V. I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10 (1966), p. 707.
- [211] Paul Jaccard. “Étude comparative de la distribution florale dans une portion des Alpes et des Jura”. In: *Bulletin del la Société Vaudoise des Sciences Naturelles* 37 (1901), pp. 547–579.
- [212] T. T. Tanimoto. *IBM Internal Report*. 1957.
- [213] Themistoklis Diamantopoulos and Andreas L. Symeonidis. “Employing Source Code Information to Improve Question-answering in Stack Overflow”. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 2015, pp. 454–457.

- [214] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. “Mining succinct and high-coverage API usage patterns from source code”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 319–328.
- [215] Jaroslav Fowkes and Charles Sutton. “Parameter-free probabilistic API mining across GitHub”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016, pp. 254–265.
- [216] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. “Documenting APIs with examples: Lessons learned with the APIMiner platform”. In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. Piscataway, NJ, USA: IEEE Computer Society, 2013, pp. 401–408.
- [217] Raymond P. L. Buse and Westley Weimer. “Synthesizing API Usage Examples”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 782–792.
- [218] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. “Towards an Intelligent Code Search Engine”. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI’10. Atlanta, Georgia: AAAI Press, 2010, pp. 1358–1363.
- [219] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. “Example-centric Programming: Integrating Web Search into the Development Environment”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’10. Atlanta, Georgia, USA: ACM, 2010, pp. 513–522.
- [220] Jianyong Wang and Jiawei Han. “BIDE: Efficient Mining of Frequent Closed Sequences”. In: *Proceedings of the 20th International Conference on Data Engineering*. ICDE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 79–90.
- [221] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones”. In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105.
- [222] Raymond P. L. Buse and Westley R. Weimer. “Learning a Metric for Code Readability”. In: *IEEE Trans. Softw. Eng.* 36.4 (2010), pp. 546–558.
- [223] Charu C. Aggarwal and ChengXiang Zhai. “A Survey of Text Clustering Algorithms”. In: *Mining Text Data*. Boston, MA: Springer US, 2012, pp. 77–128.
- [224] Annie T. T. Ying. “Mining Challenge 2015: Comparing and combining different information sources on the Stack Overflow data set”. In: *The 12th Working Conference on Mining Software Repositories*. 2015, to appear.
- [225] S. Subramanian and R. Holmes. “Making sense of online code snippets”. In: *Proceedings of the 2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*. 2013, pp. 85–88.

- [226] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. “Live API documentation”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 643–652.
- [227] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. “Mining StackOverflow to turn the IDE into a self-confident programming prompter”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 102–111.
- [228] Sheng-Kuei Hsu and Shi-Jen Lin. “MACs: Mining API code snippets for code reuse”. In: *Expert Syst. Appl.* 38.6 (2011), pp. 7291–7301.
- [229] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. “Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories”. In: *35th International Conference on Software Engineering*. ICSE 2013. San Francisco, CA, 2013, pp. 422–431.
- [230] Gianluigi Caldiera and Victor R. Basili. “Identifying and Qualifying Reusable Software Components”. In: *Computer* 24.2 (1991), pp. 61–70.
- [231] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. “Does Refactoring Improve Reusability?” In: *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components*. ICSR’06. Turin, Italy: Springer-Verlag, 2006, pp. 287–297.
- [232] Gui Gui and Paul D. Scott. “Coupling and Cohesion Measures for Evaluation of Component Reusability”. In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR ’06. Shanghai, China: ACM, 2006, pp. 18–21.
- [233] Jeffrey S. Poulin. “Measuring software reusability”. In: *Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability*. 1994, pp. 126–138.
- [234] Fatma Dandashi. “A Method for Assessing the Reusability of Object-oriented Code Using a Validated Set of Automated Measurements”. In: *Proceedings of the 2002 ACM Symposium on Applied Computing*. SAC ’02. Madrid, Spain: ACM, 2002, pp. 997–1003.
- [235] *ISO/IEC 9126-1:2001*. 2001. Available online: <https://www.iso.org/standard/22749.html> [Retrieved: October, 2017].
- [236] Fathi Taibi. “Empirical Analysis of the Reusability of Object-Oriented Program Code in Open-Source Software”. In: *International Journal of Computer, Information, System and Control Engineering* 8.1 (2014), pp. 114–120.
- [237] Aditya Pratap Singh and Pradeep Tomar. “Estimation of Component Reusability through Reusability Metrics”. In: *International Journal of Computer, Electrical, Automation, Control and Information Engineering* 8.11 (2014), pp. 1965–1972.
- [238] Parvinder Singh Sandhu and Hardeep Singh. “A reusability evaluation model for OO-based software components”. In: *International Journal of Computer Science* 1.4 (2006), pp. 259–264.

- [239] Anupama Kaur, Himanshu Monga, Mnupreet Kaur, and Parvinder S Sandhu. “Identification and performance evaluation of reusable software components based neural network”. In: *International Journal of Research in Engineering and Technology* 1.2 (2012), pp. 100–104.
- [240] Sonia Manhas, Rajeev Vashisht, Parvinder S Sandhu, and Nirvair Neeru. “Reusability Evaluation Model for Procedure-Based Software Systems”. In: *International Journal of Computer and Electrical Engineering* 2.6 (2010).
- [241] Ajay Kumar. “Measuring Software reusability using SVM based classifier approach”. In: *International Journal of Information Technology and Knowledge Management* 5.1 (2012), pp. 205–209.
- [242] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. “A probabilistic software quality model”. In: *27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 243–252.
- [243] Michail Papamichail, Themistoklis Diamantopoulos, Ilias Chrysovergis, Philippos Samlidis, and Andreas Symeonidis. “User-Perceived Reusability Estimation based on Analysis of Software Repositories”. In: *Proceedings of the 2018 IEEE International Workshop on Machine Learning Techniques for Software Quality Evaluation*. MaLTeSQuE. Campobasso, Italy, 2018, pp. 49–54.
- [244] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. “The SQO-OSS quality model: measurement based open source software evaluation”. In: *Open source development, communities and quality* (2008), pp. 237–248.
- [245] Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. “A drill-down approach for measuring maintainability at source code element level”. In: *Electronic Communications of the EASST* 60 (2013).
- [246] Matthieu Foucault, Marc Palyart, Jean-Rémy Falleri, and Xavier Blanc. “Computing contextual metric thresholds”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM. 2014, pp. 1120–1125.
- [247] Raed Shatnawi, Wei Li, James Swain, and Tim Newman. “Finding software metrics threshold values using ROC curves”. In: *Journal of Software: Evolution and Process* 22.1 (2010), pp. 1–16.
- [248] Till G. Bay and Karl Pauls. *Reuse Frequency as Metric for Component Assessment*. Technical Report. Zurich: ETH, Department of Computer Science, 2004.
- [249] *SonarQube platform*. 2016. Available online: <http://www.sonarqube.org/> [Retrieved: June, 2016].
- [250] Hudson Borges, Andre Hora, and Marco Tulio Valente. “Predicting the Popularity of GitHub Repositories”. In: *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2016. Ciudad Real, Spain: ACM, 2016, 9:1–9:10.
- [251] *ARISA - Reusability related metrics*. 2008. Available online: <http://www.arisa.se/compendium/node38.html> [Retrieved: September, 2017].
- [252] *SourceMeter static analysis tool*. 2017. Available online: <https://www.sourceter.com/> [Retrieved: November, 2017].

- [253] David P Doane and Lori E Seward. “Measuring skewness: a forgotten statistic”. In: *Journal of Statistics Education* 19.2 (2011), pp. 1–18.
- [254] Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. “Assessing the User-Perceived Quality of Source Code Components using Static Analysis Metrics”. In: *Communications in Computer and Information Science*. Springer, 2018, in press.

# Λίστα Δημοσιεύσεων

## Άρθρα σε Περιοδικά

1. **Themistoklis Diamantopoulos** and Andreas Symeonidis. “Enhancing Requirements Reusability through Semantic Modeling and Data Mining Techniques”. In: *Enterprise Information Systems* (2017), pp. 1–22.
2. **Themistoklis Diamantopoulos**, Michael Roth, Andreas Symeonidis, and Ewan Klein. “Software Requirements as an Application Domain for Natural Language Processing”. In: *Language Resources and Evaluation* 51.2 (2017), pp. 495–524.
3. Christoforos Zolotas, **Themistoklis Diamantopoulos**, Kyriakos Chatzidimitriou, and Andreas Symeonidis. “From requirements to source code: a Model-Driven Engineering approach for RESTful web services”. In: *Automated Software Engineering* (2016), pp. 1–48.
4. **Themistoklis Diamantopoulos** and Andreas Symeonidis. “Localizing Software Bugs using the Edit Distance of Call Traces”. In: *International Journal On Advances in Software* 7.1 (2014), pp. 277–288.
5. **Themistoklis Diamantopoulos** and Andreas L. Symeonidis. “AGORA: A Search Engine for Source Code Reuse”. In: *SoftwareX* (under review).

## Κεφάλαια σε Βιβλία

1. Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, **Themistoklis Diamantopoulos**, and Andreas Symeonidis. “Assessing the User-Perceived Quality of Source Code Components using Static Analysis Metrics”. In: *Communications in Computer and Information Science*. Springer, 2018, in press.

## Δημοσιεύσεις σε Συνέδρια

1. **Themistoklis Diamantopoulos**, Georgios Karagiannopoulos, and Andreas L. Symeonidis. “CodeCatch: Extracting Source Code Snippets from Online Sources”. In: *Proceedings of the IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. RAISE ’18. Gothenburg, Sweden, 2018, pp. 21–27.

2. Michail Papamichail, **Themistoklis Diamantopoulos**, Ilias Chrysovergis, Philippos Samlidis, and Andreas Symeonidis. “User-Perceived Reusability Estimation based on Analysis of Software Repositories”. In: *Proceedings of the 2018 IEEE International Workshop on Machine Learning Techniques for Software Quality Evaluation*. MaLTeSQuE. Campobasso, Italy, 2018, pp. 49–54.
3. Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, **Themistoklis Diamantopoulos**, and Andreas Symeonidis. “Towards Modeling the User-perceived Quality of Source Code using Static Analysis Metrics”. In: *Proceedings of the 12th International Conference on Software Technologies - Volume 1*. ICSoft. INSTICC. Setubal, Portugal: SciTePress, 2017, pp. 73–84.
4. Michail Papamichail, **Themistoklis Diamantopoulos**, and Andreas L. Symeonidis. “User-Perceived Source Code Quality Estimation based on Static Analysis Metrics”. In: *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security*. QRS. Vienna, Austria, 2016, pp. 100–107.
5. **Themistoklis Diamantopoulos**, Antonis Noutsos, and Andreas L. Symeonidis. “DP-CORE: A Design Pattern Detection Tool for Code Reuse”. In: *Proceedings of the 6th International Symposium on Business Modeling and Software Design*. BMSD. Rhodes, Greece, June 2016, pp. 160–169.
6. **Themistoklis Diamantopoulos**, Klearchos Thomopoulos, and Andreas L. Symeonidis. “QualBoa: Reusability-aware Recommendations of Source Code Components”. In: *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*. MSR ’16. 2016, pp. 488–491.
7. **Themistoklis Diamantopoulos** and Andreas Symeonidis. “Towards Interpretable Defect-Prone Component Analysis using Genetic Fuzzy Systems”. In: *Proceedings of the IEEE/ACM 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. RAISE ’15. Florence, Italy, 2015, pp. 32–38.
8. **Themistoklis Diamantopoulos** and Andreas L. Symeonidis. “Employing Source Code Information to Improve Question-answering in Stack Overflow”. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR ’15. Florence, Italy: IEEE Press, 2015, pp. 454–457.
9. Michael Roth, **Themistoklis Diamantopoulos**, Ewan Klein, and Andreas L. Symeonidis. “Software Requirements: A new Domain for Semantic Parsers”. In: *Proceedings of the ACL 2014 Workshop on Semantic Parsing*. SP14. Baltimore, Maryland, USA, 2014, pp. 50–54.
10. **Themistoklis Diamantopoulos** and Andreas L. Symeonidis. “Towards Scalable Bug Localization using the Edit Distance of Call Traces”. In: *Proceedings of the Eighth International Conference on Software Engineering Advances*. ICSEA 2013. Venice, Italy, 2013, pp. 45–50.