ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ (MSc)
στα ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Βελτίωση Συνεκτικότητας στο Αντικειμενοστραφές Λογισμικό»

ΚΙΟΥΚΗΣ ΙΩΑΝΝΗΣ
M3110024

ΑΘΗΝΑ, ΦΕΒΡΟΥΑΡΙΟΣ 2013

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ (MSc)
στα ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Βελτίωση Συνεκτικότητας
στο Αντικειμενοστραφές Λογισμικό»

ΚΙΟΥΚΗΣ ΙΩΑΝΝΗΣ
M3110024

Επιβλέπων Καθηγητής: Γιακουμάκης Εμμανουήλ
Εξωτερικός Κριτής: Μαλεύρης Νικόλαος

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΘΗΝΑ, ΦΕΒΡΟΥΑΡΙΟΣ 2013

Ευχαριστίες

Θα ήθελα να ευχαριστήσω όλους όσοι με στήριξαν κατά την εκπόνηση της διπλωματικής μου εργασίας.

Καταρχάς, θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον καθηγητή κ. Εμμανουήλ Γιακουμάκη, ο οποίος με εμπιστεύτηκε για την εκπόνηση της παρούσας διπλωματικής εργασίας. Η συμβολή του ήταν πολύτιμη καθώς το θέμα της εργασίας αντιπροσωπεύει απόλυτα το αντικείμενο του ενδιαφέροντός μου στον τομέα της πληροφορικής.

Επίσης, αμέριστη υπήρξε η υποστήριξη που έλαβα από τον διδάκτορα Πληροφορικής κ. Νικόλαο Διαμαντίδη, ο οποίος σε κάθε βήμα της προσπάθειάς μου, μου παρείχε τους κύριους άξονες στους οποίους στηρίχθηκα και με ενθάρρυνε για τη βελτίωση και την ολοκλήρωση του έργου μου.

Επιπλέον, πρωταρχικής σημασίας υπήρξε η προσφορά του διδάκτορα Πληροφορικής κ. Βασίλη Ζαφείρη, με τον οποίο είχα την τιμή να συνεργαστώ, να μοιραστώ τις σκέψεις μου και να τις επανεξετάσω υπό άλλες διαστάσεις. Οι συμβουλές του υπήρξαν καθοριστικές.

Τέλος, δεν πρέπει να παραλείψω την αρωγή του υποψήφιου διδάκτορα κ. Θεόδωρου Χαϊκάλη, ο οποίος μου εμπιστεύτηκε τον κώδικα της μεταπτυχιακής του διατριβής ώστε να τον μελετήσω και να τον επεκτείνω.

Περίληψη

Τα τελευταία έτη η ραγδαία εξέλιξη των τεχνολογιών έχει οδηγήσει στην ανάπτυξη πληθώρας λογισμικού. Ωστόσο, η ποιότητα συγγραφής του ποικίλει εξαιτίας διάφορων παραγόντων. Οι προγραμματιστές καλούνται να συντάξουν μεγάλα τμήματα λογισμικού, το οποίο επεκτείνεται συνεχώς από τον ίδιο ή και διαφορετικούς προγραμματιστές. Επίσης, η έλλειψη χρόνου λόγω της πίεσης για την έγκαιρη έκδοση και παράδοσή του λειτουργεί αποτρεπτικά για την πραγματοποίηση επαρκών ελέγχων για τη διασφάλιση της ποιότητας. Η έλλειψη εμπειρίας των προγραμματιστών είναι ακόμα ένας παράγοντας που ενδέχεται να οδηγήσει στη συγγραφή μη ποιοτικού κώδικα.

Αρωγός για την ανάπτυξη της παρούσας εργασίας είναι η ανάγκη για τη βελτίωση του κώδικα, προκειμένου να πληροί βασικά χαρακτηριστικά. Στόχος είναι η υλοποίηση ενός αυτοματοποιημένου εργαλείου που θα μπορεί να χρησιμοποιηθεί εύκολα και γρήγορα από τους προγραμματιστές αντικειμενοστραφών γλωσσών προγραμματισμού.

Η καθιέρωση προτύπων είναι μία προσπάθεια για τη συγγραφή ποιοτικού κώδικα. Κάθε πρότυπο περιγράφει ένα πρόβλημα το οποίο συναντάται επανειλημμένα σε ένα περιβάλλον. Εν συνεχεία, περιγράφει τον πυρήνα της λύσης του προβλήματος με τέτοιο τρόπο, ώστε η λύση να μπορεί να επαναχρησιμοποιηθεί χιλιάδες φορές, χωρίς απαραίτητα να εφαρμόζεται με τον ίδιο τρόπο. Στόχος των προτύπων είναι η ικανοποίηση των αξιών και των αρχών προγραμματισμού.

Για την εφαρμογή ενός προτύπου σε ήδη υπάρχοντα κώδικα απαιτείται η χρήση αναδομήσεων. Αναδόμηση κώδικα ορίζεται η διαδικασία της μετατροπής ενός συστήματος λογισμικού με τέτοιο τρόπο ώστε να μην τροποποιείται η συμπεριφορά του, ενώ ταυτόχρονα βελτιώνεται η δομή του. Για συγκεκριμένες περιπτώσεις προβληματικού σχεδιασμού ή συγγραφής, υπάρχουν τυποποιημένες προτάσεις

αναδόμησης. Οι δομές κώδικα για τις οποίες απαιτείται αναδόμηση αποκαλούνται ως κακές οσμές (bad smells).

Στην παρούσα διπλωματική επιδιώκεται η επέκταση του εργαλείου JDeodorant, ενός προσθέτου για το περιβάλλον ανάπτυξης λογισμικού του Eclipse. Το εργαλείο παρέχει τη δυνατότητα ανάδειξης κακών οσμών που οδηγούν σε μείωση της ποιότητας του λογισμικού και προτείνει λύσεις για αυτά. Η λειτουργικότητα που προστίθεται είναι η ανίχνευση και η αντιμετώπιση σύνθετων μεθόδων.

Σύνθετη μέθοδος καλείται κάθε μακροσκελής μέθοδος η οποία έχει αυξημένη πολυπλοκότητα. Η αναδόμηση που προτείνεται για την αντιμετώπιση της παραπάνω κακής οσμής είναι η Replace Method with Method Object. Αυτή ορίζει τη μετατροπή μιας μεθόδου σε τάξη με πεδία τις τοπικές μεταβλητές και τις παραμέτρους της αρχικής μεθόδου.

Με την αναδόμηση Replace Method with Method Object επιδιώκεται η βελτίωση της συνεκτικότητας του κώδικα. Η συνεκτικότητα είναι ένας σημαντικός παράγοντας για την αξιολόγηση της ποιότητας του λογισμικού. Ορίζεται ως η ποιοτική ιδιότητα που εκτιμά τη μοναδικότητα του σκοπού μιας μονάδας λογισμικού.

Οι περιπτώσεις εφαρμογής της συγκεκριμένης αναδόμησης είναι σπάνιες. Η χρήση της ωστόσο, οδηγεί στη βελτίωση πολλών ποιοτικών χαρακτηριστικών του λογισμικού, όπως η συντηρησιμότητα, η επεκτασιμότητα, η ευχρηστία και η επαναχρησιμότητα. Λόγω της σπανιότητάς της, η ανίχνευση των αντίστοιχων κακών οσμών με αυτοματοποιημένο τρόπο αποτελεί μια πρόκληση.

Η προσέγγιση που ακολουθείται για την ανίχνευση των κακών οσμών περιλαμβάνει την ανάλυση του κώδικα και την υπόδειξη εκείνων των τμημάτων που παρουσιάζουν συγκεκριμένα χαρακτηριστικά. Τίθενται κάποια αρχικά κριτήρια για να αποκλεισθούν από την περαιτέρω ανάλυση μέθοδοι στις οποίες δεν έχει νόημα να εφαρμοστεί η συγκεκριμένη αναδόμηση.

Στην συνέχεια, βάσει του Αφηρημένου Συντακτικού Δέντρου (Abstract Syntax Tree) κατασκευάζεται το Γράφημα Ροής Ελέγχου (Control Flow Graph) του προγράμματος. Το Αφηρημένο Συντακτικό Δέντρο αναπαριστά τον πηγαίο κώδικα σε δενδρική μορφή. Το Γράφημα Ροής Ελέγχου παρουσιάζει τη ροή ελέγχου μέσα στο

πρόγραμμα. Από το τελευταίο εξάγονται οι εξαρτήσεις του προγράμματος και δημιουργείται το Γράφημα Εξαρτήσεων του Προγράμματος (Program Dependence Graph). Αυτό αποτελεί την απεικόνιση της εσωτερικής δομής ενός προγράμματος. Αναφέρεται τόσο στην εξάρτηση των δεδομένων στο πρόγραμμα όσο και στην εξάρτηση των ελέγχων για κάθε λειτουργία του προγράμματος.

Με τη βοήθεια του Γραφήματος Εξαρτήσεων εξάγονται τα στατικά slices. Τα slices απεικονίζουν τις εξαρτήσεις των εντολών του προγράμματος με μια συγκεκριμένη μεταβλητή σε ένα ορισμένο σημείο του προγράμματος. Εξαρτώνται από το σημείο και τη μεταβλητή για την οποία παράγονται. Διακρίνονται σε προς τα πίσω (backward), προς τα εμπρός (forward) και μετρικά (metric). Το forward slice απεικονίζει το σύνολο των εντολών, των οποίων το αποτέλεσμα εξαρτάται από τη μεταβλητή από το σημείο που ορίζεται το slice και κάτω. Το backward slice απεικονίζει το σύνολο των εντολών που επηρεάζουν την τιμή της μεταβλητής στο σημείο που ορίζεται το slice. Το metric slice είναι ο συνδυασμός των παραπάνω.

Από τα slices υπολογίζονται οι τιμές coverage και overlap μιας μεθόδου. Η μετρική coverage συγκρίνει το μήκος του slice με το μήκος της μεθόδου και ορίζεται ως ο μέσος λόγος του μήκους του slice προς το μήκος της μεθόδου. Το μήκος υπολογίζεται σε πλήθος εντολών. Το overlap υπολογίζει πόσες εντολές αποτελούν αποκλειστικότητα ενός και μόνου συγκεκριμένου slice, δηλαδή δεν εμφανίζονται και σε άλλα slices. Υπολογίζεται ως ο μέσος λόγος του συνόλου των εντολών που εμφανίζονται στην τομή των slices προς το μήκος του κάθε slice. Οι τιμές coverage και overlap υπολογίζονται τόσο για τις τοπικές μεταβλητές όσο και για τις παραμέτρους των μεθόδων καθώς και για όλα τα είδη slicing.

Για να εντοπισθούν οι υπολογιζόμενες μετρικές που υποδεικνύουν την πιθανή ανάγκη εφαρμογής της αναδόμησης Replace Method with Method Object, εξήχθη ένα σύνολο αποτελεσμάτων για ένα δείγμα προγραμμάτων. Στη συνέχεια, μελετήθηκε κάθε μέθοδος ξεχωριστά για να κριθεί εάν χρειάζεται να εφαρμοστεί η συγκεκριμένη αναδόμηση.

Τα δεδομένα από την αξιολόγηση και οι τιμές των μετρικών χρησιμοποιήθηκαν ως είσοδος στο Weka, λογισμικό που παρέχει υλοποίηση αλγορίθμων μηχανικής μάθησης για την εξόρυξη δεδομένων. Για την αξιολόγηση των δεδομένων των

προγραμμάτων χρησιμοποιήθηκε ο ταξινομητής J48 που αποτελεί υλοποίηση του αλγορίθμου C4.5. Είναι ένας αλγόριθμος που χρησιμοποιείται για την παραγωγή δέντρων αποφάσεων. Τα δέντρα που παράγονται χρησιμοποιούνται για την ταξινόμηση.

Διάφορα χαρακτηριστικά των μεθόδων εξετάστηκαν για την παραγωγή του τελικού δέντρου αποφάσεων, του κανόνα που ορίζει τις μεθόδους που είναι υποψήφιες για την αναδόμηση Replace Method with Method Object. Τα αποτελέσματα αξιολογήθηκαν βάσει της ακρίβειας (precision) και της ανάκλησης (recall). Η ακρίβεια περιγράφει το λόγο των μεθόδων που κατατάσσονται ορθά ως υποψήφιες για αναδόμηση προς το σύνολο των μεθόδων που κατατάσσονται ως υποψήφιες. Η ανάκληση περιγράφει το λόγο των υποψήφιων μεθόδων που βρέθηκαν προς το σύνολο των υποψηφίων που υπάρχουν.

Η αξιολόγηση έγινε βάσει εξέτασης και χαρακτηρισμού ως υποψηφίων για refactoring ή μη, ενός συνόλου μεθόδων με συγκεκριμένα χαρακτηριστικά (μέγεθος, πλήθος μεταβλητών και παραμέτρων κ.ά.) από διάφορα προγράμματα. Πρέπει να αναφερθεί ότι στη διαδικασία ενέχεται υποκειμενικότητα. Αυτή έγκειται στο γεγονός ότι ο χαρακτηρισμός περιπτώσεων ως υποψηφίων για αναδόμηση έγινε βάσει της προσωπικής κρίσης προγραμματιστών και όχι με κάποιο πιο αντικειμενικό κριτήριο. Για περαιτέρω αξιολόγηση θα πρέπει να γίνει εφαρμογή των κανόνων που εξήχθησαν και σε άλλα προγράμματα και να γίνει επαλήθευση των αποτελεσμάτων από προγραμματιστές.

Η αυτοματοποίηση της ανίχνευσης σύνθετων μεθόδων είναι μια διαδικασία δύσκολη. Οι διάφορες μετρικές (overlap και coverage) και παραλλαγές αυτών δεν είναι ικανές από μόνες τους να διακρίνουν με ακρίβεια περιπτώσεις σύνθετων μεθόδων που χρήζουν αναδόμησης.

Το συμπέρασμα βασίζεται σε υλοποίηση πλήθους μετρικών και εφαρμογή αυτών σε αρκετά προγράμματα διαφόρων μεγεθών. Από τις μετρικές που αξιολογήθηκαν καταλληλότερη για τη διαδικασία αναδείχθηκε η forward overlap, που προκύπτει από το forward slice της μεθόδου. Η εφαρμογή της μηχανικής μάθησης δεν υπέδειξε κάποια σαφή κατώφλια για τις διάφορες μετρικές πέρα των οποίων η εκάστοτε μέθοδος θεωρείται σύνθετη.

Ακόμα, αντί της μετρικής Backward Overlap προτείνεται η χρήση της Metric Overlap που βασίζεται τόσο στα backward όσο και στα forward slices των μεταβλητών. Η τελευταία περιορίζει το ενδεχόμενο η τιμή της να μηδενιστεί, ενώ εκφράζει επίσης το βαθμό επικάλυψης των slices. Παρόλα αυτά και αυτή είναι περιοριστική και θα είχε ενδιαφέρον ο υπολογισμός και η χρήση παραλλαγών αυτής.

Ακολουθεί σύντομη επισκόπηση των κεφαλαίων της διατριβής.

Στο πρώτο κεφάλαιο γίνεται εισαγωγή στην έννοια της ποιότητας του λογισμικού. Παρατίθενται διάφοροι ορισμοί που έχουν διατυπωθεί κατά καιρούς για την αποσαφήνιση των όρων "ποιότητα" γενικά και "ποιότητα λογισμικού" ειδικότερα. Περιγράφονται οι ποικίλες θεωρήσεις με τις οποίες εκλαμβάνεται η έννοια και παρουσιάζονται μοντέλα που έχουν αναπτυχθεί στην προσπάθεια εύρεσης των χαρακτηριστικών που καθιστούν το λογισμικό ποιοτικό. Απεικονίζεται το κόστος στα διάφορα στάδια της διαδικασίας ανάπτυξης του λογισμικού και υποδεικνύεται η σχέση συνεκτικότητας και ποιότητας λογισμικού.

Στο δεύτερο κεφάλαιο καταγράφονται οι διάφορες μέθοδοι και τα στάδια της ανάλυσης του κώδικα. Περιγράφονται οι έννοιες των Αφηρημένων Συντακτικών Δέντρων, του Γράφου Ροής Ελέγχου, του Γράφου Εξαρτήσεων του Προγράμματος και τέλος η έννοια του Τεμαχισμού (Slicing). Δίδονται παραδείγματα για την αποσαφήνιση των όρων και την ευκολότερη κατανόησή τους. Τέλος, γίνεται ανάλυση των μετρικών που βασίζονται στα slices.

Στο τρίτο κεφάλαιο γίνεται εισαγωγή στην έννοια των προτύπων σχεδίασης και υλοποίησης. Αναλύονται οι αξίες και οι αρχές που διέπουν τον προγραμματισμό και τα κίνητρα για την εφαρμογή των προτύπων. Καθίσταται σαφής η συμβολή τους στη συγγραφή ποιοτικού κώδικα και κατ' επέκταση στη βελτίωση της ποιότητας του λογισμικού.

Στο τέταρτο κεφάλαιο παρουσιάζεται η έννοια της αναδόμησης και αναλύονται τα οφέλη από την εφαρμογή τους. Αναφέρονται οι περιπτώσεις στις οποίες εφαρμόζονται και οι περιπτώσεις στις οποίες δεν εφαρμόζονται. Αναλύονται τα διάφορα προβλήματα που μπορεί να προκύψουν κατά την εφαρμογή τους, οι

επιπτώσεις που έχουν στην απόδοση των προγραμμάτων και τέλος, γίνεται συσχέτιση μεταξύ των αναδομήσεων και της σχεδίασης λογισμικού.

Στο πέμπτο κεφάλαιο περιγράφονται δομές κώδικα που αλλοιώνουν την ποιότητά του. Αναλύονται η Μακροσκελής και η Σύνθετη Μέθοδος. Μετά την παρουσίαση των προβλημάτων παρατίθενται και οι προτεινόμενες αναδομήσεις για την αντιμετώπισή τους. Δίδονται παραδείγματα που αποτυπώνουν τα παραπάνω και βοηθούν στην κατανόηση τους.

Στο έκτο κεφάλαιο γίνεται η περιγραφή της διαδικασίας αυτοματοποίησης του εντοπισμού των Σύνθετων Μεθόδων και της εφαρμογής της αντίστοιχης αναδόμησης. Περιγράφεται το εργαλείο που χρησιμοποιείται, τα στάδια ανάλυσης του εξεταζόμενου προγράμματος, οι προϋποθέσεις εξέτασης των διάφορων μεθόδων του και τα χαρακτηριστικά τους.

Στο έβδομο κεφάλαιο περιγράφεται η μεθοδολογία που ακολουθήθηκε για την εξαγωγή των συμπερασμάτων. Παρουσιάζονται τα αποτελέσματα που προέκυψαν από την ανάλυση των μεθόδων και ο κανόνας που εξήχθη για τον εντοπισμό Σύνθετων Μεθόδων.

Στο όγδοο κεφάλαιο συνοψίζονται τα συμπεράσματα που προέκυψαν και γίνεται αναφορά σε πεδία που παραμένουν ανοικτά και προσφέρονται για μελλοντική έρευνα.

Abstract

In the recent years, the radical development of technologies has led to the development of a plethora of software. However, the software quality varies. Programmers are asked to write large sections of software, that is continuously expanded by the same or different programmers. Furthermore, the lack of time due to the pressure for immediate publishing and delivery acts as a disincentive for the adequate checking of software to assure its quality. The lack of experience of programmers is yet another factor which may lead to the writing of non qualitative code.

The need to improve code so that it covers some basic characteristics is the motivation for this thesis. Its purpose is the creation of an automated tool that can be easily and quickly used from programmers of object-oriented languages.

The standardization of design and implementation patterns is an attempt to write qualitative code. Each pattern describes a problem that appears repeatedly. Then, it describes the core of the problem's solution so that it can be used thousands of times, without necessarily being implemented in the same way. The patterns aim in satisfying the values and principles of programming.

The use of refactoring is required to implement a pattern to preexisting code. Refactoring is the procedure of modification of a software system in a way that it retains its behavior while its structure is being improved. For certain cases of problematic design or development, standardized suggestions of code refactoring exist. The structures of code for which refactoring is required are known as bad smells.

In the present thesis, an effort is made to expand JDeodorant, a plugin for the Eclipse development environment. The tool offers the ability of identification of bad smells that decrease software's quality and suggests solutions. The functionality added is the detection and elimination of Complex Methods.

Every Long Method that has an increased complexity is called Complex Method. The refactoring proposed to eliminate the latter is Replace Method with Method Object. This refactoring defines the conversion of a method to a class that has the local variables and parameters of the initial method as fields.

Replace Method with Method Obect refactoring aims in improving code's coherence. Coherence is an important factor for the evaluation of software's quality. It is defined as the qualitative characteristic that estimates the uniqueness of a program module's purpose.

The cases that require this type of refactoring are rare. Its use, however, leads to the improvement of plenty of the software's quality characteristics, such as maintainability, extensibility, usability and reusability. Due to their infrequency, the identification of such type of bad smells in an automated way is a challenge.

The approach followed for the identification of bad smells includes the analysis of the code and detection of sections that have certain characteristics. Some initial criteria are posed to exclude from further analysis methods for which the application of this refactoring is needless.

Later, based on program's Abstract Syntax Tree, the Control Flow Graph is created. Abstract Syntax Trees depict the source code in a tree form. The Control Flow Graph presents the flow of control through the program. The dependencies of the program are extracted from the latter and as a result the Program Dependence Graph is being built. This depicts a program's inner structure. It refers both to the data dependence in the program and control dependence for each one of the program's functions.

Based on the Dependence Graph, static slices are extracted. Slices depict the dependence of the program's statements with a precise variable at a certain point of the program. They depend on the point and the variable for which they are produced. There are different types of slices; backward, forward and metric. The forward slice depicts the set of subsequent statements whose result is being affected by the value of the variable at the point of the slice. The backward slice depicts the statements that

affected the value of the variable till the point of the slice. The metric slice is the combination of both backward and forward.

The values of Coverage and Overlap of a method are estimated based on the slices. Metric coverage compares the slice's length to the method's length and is defined as the average ratio of the slice's length to the method's length. Length is defined as the amount of statements of the method. Overlap estimates the number of statements being exclusively in one and only slice and therefore do not appear in other slices. It can be estimated as the average ratio of the sum of statements appearing in the intersection of slices to each slice's length. The values of Coverage and Overlap are calculated both for the local variables and the parameters of each method and for all types of slicing as well.

A set of results is extracted from a sample of programs in order to detect the estimated metrics which may indicate the probable need to apply Replace Method with Method Object refactoring. Almost each method of these programs was studied separately to examine whether this type of refactoring indeed needs to be applied.

The data extracted from the evaluations and the metrics' values were used as an entry to Weka, a software which provides implementation of machine learning algorithms for data mining. To evaluate the programs' data the J48 classifier, which is the implementation of the algorithm C4.5, was used. This is an algorithm for the production of decision trees. The trees produced are used for the classification.

Different characteristics of the methods were examined for the production of the final decision tree, the rule which defines the candidate methods for the Replace Method with Method Object refactoring. The results were evaluated according to precision and recall. Precision describes the rate of methods correctly classified as candidates for the refactoring divided by the sum of methods that are classified as candidates. Recall describes the rate of candidate methods found divided by the sum of all candidate methods existing.

The evaluation has been made based on the examination of a set of methods with specified characteristics (length, number of variables and parameters, etc.) and judgment of methods as candidates of refactoring or not. It must be noted that the

process involves subjectivity. This lies on the fact that the judgments of cases as candidates for refactoring have been made based on programmers personal opinion and not based on objective criteria. For further evaluation the rules that were derived must be applied to other programs as well and a verification of the results by the programmers must take place.

The automation of the complex method identification is a difficult process. The different metrics (overlap and coverage) and variations of them are not sufficient for the precise detection of Complex Methods that need refactoring.

The results are based on a variety of metrics and their application on several programs of different length. From the metrics examined for the process, Forward Overlap proved to be the most appropriate. It is derived based on the forward slices of the method. The application of machine learning did not suggest any clear thresholds for the different metrics above which each method is considered as complex.

Furthermore, the use of Metric Overlap that is computed based on both forward and backward slices of the method is suggested instead of Backward Overlap. The Metric Overlap minimizes the chance that its value equals zero, while it also expresses the degree of overlapping of slices. Despite that, Metric Overlap is also restrictive but the evaluation of variations of it might be interesting.

A synopsis of the chapters of the thesis is given below.

In the first chapter, an introduction to the concept of quality is being made. Different definitions that have been expressed throughout years for the clarification of the term “quality” in general and “software quality” particularly are listed. The different views through which the concept is perceived are described and models that have been developed in search of the characteristics that make a software qualitative are presented. The cost of the different stages of the software development process is depicted and the relation between cohesion and software quality is indicated.

In the second chapter, the different techniques and the stages of code analysis are recorded. The concepts of Abstract Syntax Tree, Control Flow Graph, Program Dependence Graph and finally the concept of Slicing are described. Examples that

clarify the terms and help in understanding them are given. Finally, an analysis of the metrics that are based on slices is made.

In the third chapter, an introduction to the concept of Design Patterns and Implementation Patterns is made. The values and principles of programming and the motivation for the application of patterns are analyzed. Their contribution in writing qualitative code and subsequently in the improvement of software quality is made clear.

In the fourth chapter, the concept of refactoring is presented and the benefits of their application are analyzed. The cases in which they are applied and the cases in which they are not are mentioned. The potential problems that might be encountered due to their application and their effect in the performance of programs are analyzed and finally, a correlation between refactorings and software design is made.

In the fifth chapter, code structures that affect its quality are described. Long Method and Complex Method are analyzed. After the presentation of problems, the suggested refactorings for their treatment are listed. Examples of the above are given.

In the sixth chapter, the description of the automation process for identification of Complex Method and the application of the corresponding refactoring is made. The tool that is being used, the preconditions of examining the different methods and their characteristics are described.

In the seventh chapter, the methodology for the extraction of conclusions is described. The results that emerged for the analysis of methods and the rules that were extracted for the identification of Complex Methods are presented.

In the eighth chapter, the conclusions that emerged are summarized and fields open for further research are mentioned.

Περιεχόμενα

Περίληψη.....5
Abstract.....11
Κεφάλαιο 1.....21
Ποιότητα Λογισμικού.....21
1.1 Ορισμοί Ποιότητας.....21
1.2 Θεωρήσεις Ποιότητας Λογισμικού.....22
1.2.1 Υπερβατική Θεώρηση (Transcendental View).....22
1.2.2 Θεώρηση Χρήστη (Users View).....22
1.2.3 Κατασκευαστική Θεώρηση (Manufacturing View).....23
1.2.4 Θεώρηση Προϊόντος (Product View).....23
1.2.5 Θεώρηση Βάσει Αξίας (Value-based View).....24
1.3 Στοιχεία που χαρακτηρίζουν ένα λογισμικό ποιοτικό.....24
1.3.1 Μοντέλο McCall 25
1.3.2 Μοντέλο Boehm.....27
1.4 Κατανομή Κόστους Ανάπτυξης Λογισμικού.....29
1.5 Συνεκτικότητα Κώδικα και Ποιότητα Λογισμικού.....30
Κεφάλαιο 2.....32
Ανάλυση Κώδικα.....32
2.1 Αφηρημένο Συντακτικό Δέντρο (Abstract Syntax Tree).....32
2.2 Γράφος Ροής Ελέγχου (Control Flow Graph).....36
2.3 Γράφος Εξαρτήσεων (Program Dependence Graph).....37
2.4 Slicing Προγράμματος.....41
2.5 Block-Based Slicing.....44
2.6 Slice Profiles.....49
2.7 Μετρικές Βασισμένες σε Slices.....50
Κεφάλαιο 3.....54
Πρότυπα Σχεδίασης και Θεωρία Προγραμματισμού.....54
3.1 Αξίες Προγραμματισμού.....55
3.1.1 Επικοινωνία.....56

3.1.2 Απλότητα....56
3.1.3 Ευελιξία....57
3.2 Αρχές Προγραμματισμού....58
3.2.1 Τοπικότητα Επιπτώσεων....58
3.2.2 Ελαχιστοποίηση Επανάληψης....58
3.2.3 Συνύπαρξη Λογικής και Δεδομένων....59
3.2.4 Συμμετρία....59
3.2.5 Δηλωτική Έκφραση....60
3.2.6 Ρυθμός Αλλαγών....60
3.3 Κίνητρα....61
Κεφάλαιο 4....62
Αναδομήσεις....62
4.1 Οφέλη....62
4.1.1 Βελτίωση Σχεδίασης Λογισμικού....62
4.1.2 Ευκολία Κατανόησης....63
4.1.3 Αποσφαλμάτωση....63
4.1.4 Ταχύτητα Προγραμματισμού....63
4.2 Πότε Εφαρμόζονται....64
4.3 Πότε Δεν Εφαρμόζονται....65
4.4 Προβλήματα....66
4.4.1 Βάσεις Δεδομένων....66
4.4.2 Τροποποίηση Διεπαφών....67
4.4.3 Τροποποίηση Σχεδίασης....67
4.5 Αναδομήσεις και Σχεδίαση....68
4.6 Αναδομήσεις και Απόδοση....69
Κεφάλαιο 5....71
Αναδομήσεις στην Πράξη....71
5.1 Μακροσκελής Μέθοδος (Long Method)....71
5.2 Extract Method....73
5.3 Σύνθετη Μέθοδος (Complex Method)....75
5.4 Replace Method with Method Object....78
Κεφάλαιο 6....82

Αυτοματοποίηση Αναδόμησης Replace Method with Method Object.....82
6.1 Παρουσίαση Αρχικού Εργαλείου.....82
6.2 Μέθοδος Εντοπισμού Υποψήφιων Μεθόδων προς Αναδόμηση.....84
6.2.1 Φιλτράρισμα Πεδίου Ορισμού.....86
6.2.2 Προϋποθέσεις Εφαρμογής Αναδόμησης.....87
6.2.3 Υπολογισμός Μετρικών.....88
Κεφάλαιο 7.....90
Αξιολόγηση.....90
7.1 Μεθοδολογία.....90
7.2 Αποτελέσματα.....91
Κεφάλαιο 8.....94
Συμπεράσματα και Μελλοντική Έρευνα.....94
8.1 Συμπεράσματα.....94
8.2 Μελλοντική Έρευνα.....95
Βιβλιογραφία.....97

Πίνακας Εικόνων

Εικόνα 1: Το ποιοτικό μοντέλο του McCall σε μια ιεραρχία 11 ποιοτικών παραγόντων (αριστερά) σε συσχέτιση με τα 23 κριτήρια ποιότητας (δεξιά) [3].....27
Εικόνα 2: Το δέντρο ποιοτικών χαρακτηριστικών του Boehm [3].....28
Εικόνα 3: Μέση κατανομή κόστους ανάπτυξης λογισμικού.....29
Εικόνα 4: Το AST για το πρόγραμμα του πίνακα 2 όπως προκύπτει από το ASTViewer.....35
Εικόνα 5: Τμήμα κώδικα με τον αντίστοιχο CDG [15].....39
Εικόνα 6: CFG (δεξιά) και Basic Blocks (αριστερά) για το πρόγραμμα του Πίνακα 4 [19].....46

Πίνακας Πινάκων

Πίνακας 1: Διαφορές των μοντέλων McCall και Boehm [3].....28
Πίνακας 2: Ενδεικτικό Πρόγραμμα.....34
Πίνακας 3: Σύνοψη του PDG γράφου για την εικόνα 5 [15].....40
Πίνακας 4: Σύνοψη αλγορίθμων για την εξαγωγή slices [18].....43
Πίνακας 5: Ενδεικτικό τμήμα κώδικα γραμμένο σε Java [19].....45
Πίνακας 6: Slice Profile (Backward, Forward, Metric) της μεθόδου gamma.....50
Πίνακας 7: Κώδικας πριν την εφαρμογή της Extract Method.....74
Πίνακας 8: Κώδικας μετά την εφαρμογή της Extract Method.....74
Πίνακας 9: Παράδειγμα σύνθετης μεθόδου.....75
Πίνακας 10: Κώδικας πριν την εφαρμογή της Extract Method.....77
Πίνακας 11: Κώδικας μετά την εφαρμογή της Extract Method.....77
Πίνακας 12: Κώδικας πριν την εφαρμογή της αναδόμησης Replace Method with Method Object.....80
Πίνακας 13: Κώδικας μετά την εφαρμογή της αναδόμησης Replace Method with Method Object.....81
Πίνακας 14: Επιλογές και προεπιλεγμένες τιμές αρχείου παραμέτρων.....87
Πίνακας 15: Κανόνας ανίχνευσης Complex Method.....91
Πίνακας 16: Αξιολόγηση του κανόνα ανίχνευσης.....92

Κεφάλαιο 1

Ποιότητα Λογισμικού

Κατά καιρούς διαπιστώνονται βλάβες σε αυτοματοποιημένα συστήματα. Στις περισσότερες των περιπτώσεων δε γίνεται διάκριση μεταξύ λογισμικού και υλισμικού και υπαίτιος θεωρείται «ο υπολογιστής». Ωστόσο, σε περιπτώσεις μεγάλων καταστροφών του παρελθόντος διαπιστώθηκε ότι τα αίτια σχετίζονταν με σφάλματα λογισμικού. Στη σύγχρονη εποχή τείνουν να αυτοματοποιηθούν πολλές διαδικασίες, μερικές εκ των οποίων είναι κρίσιμης σημασίας καθώς διακυβεύονται ακόμα και ανθρώπινες ζωές. Επομένως, η ποιότητα του λογισμικού αποτελεί ένα ζήτημα διόλου αμελητέο.

1.1 Ορισμοί Ποιότητας

Έχουν γίνει, κατά καιρούς, προσπάθειες αποσαφήνισης της έννοιας της ποιότητας. Στη συνέχεια παρατίθενται ορισμοί που έχουν δοθεί για το σκοπό αυτό.

Ο ορισμός που έχει δοθεί από τον Ελληνικό Οργανισμό Τυποποίησης (ΕΛΟΤ) χαρακτηρίζει την ποιότητα ως «*το σύνολο των χαρακτηριστικών μιας οντότητας που της αποδίδουν την ικανότητα να ικανοποιεί εκφρασμένες και συνεπαγόμενες ανάγκες*». Ο παραπάνω ορισμός έχει καταγραφεί στο πρότυπο ISO 8402.

Σύμφωνα με το λεξικό της Νέας Ελληνικής Γλώσσας [38], ποιότητα είναι «το σύνολο των χαρακτηριστικών πράγματος. Ό,τι το χαρακτηρίζει και το διαφοροποιεί από τα όμοιά του. Το σύνολο των θετικών ιδιοτήτων του».

Ο ορισμός που διατυπώθηκε από την American Society for Quality (ASQ) αναφέρει ότι «Ποιότητα είναι η συλλογή των χαρακτηριστικών και ιδιοτήτων του

προϊόντος που σχετίζονται με την δυνατότητα του να εκπληρώνει τις ζητούμενες ανάγκες των πελατών».

«Ποιότητα είναι η συμμόρφωση με τις απαιτήσεις των χρηστών» για τον Philip Crosby [39], ενώ σύμφωνα με τους Juran J., Grynna «ποιότητα είναι η καταλληλότητα προς χρήση» [40].

1.2 Θεωρήσεις Ποιότητας Λογισμικού

Ο David Garvin [36] εξέτασε τον τρόπο με τον οποίο εκλαμβάνεται η ποιότητα σε διάφορους τομείς όπως η φιλοσοφία, τα οικονομικά, το marketing και η διοίκηση οργανισμών. Κατέληξε ότι «Η ποιότητα είναι μια πολύπλοκη και πολυπρόσωπη ιδέα» και μπορεί να περιγραφεί από πέντε διαφορετικές θεωρήσεις.

1.2.1 Υπερβατική Θεώρηση (Transcendental View)

Σύμφωνα με την υπερβατική θεώρηση, η ποιότητα εκλαμβάνεται ως μια έννοια η οποία μπορεί να αναγνωριστεί, αλλά δεν μπορεί να οριστεί. Ο παραπάνω ορισμός είναι παρόμοιος με την περιγραφή του Πλάτωνα για το ιδανικό ή του Αριστοτέλη για την αντίληψη της μορφής. Κατ' αυτήν τη θεώρηση το ποιοτικό λογισμικό είναι η ιδανική κατάσταση που επιδιώκεται, αλλά μπορεί ποτέ να μην είναι εφικτή.

1.2.2 Θεώρηση Χρήστη (Users View)

Ενώ η υπερβατική θεώρηση εκλαμβάνει την ποιότητα ως κάτι το αιθέριο, η θεώρηση του χρήστη είναι πιο συγκεκριμένη και στηρίζεται στα χαρακτηριστικά του προϊόντος που πληρούν τις ανάγκες του. Εξετάζει την ποιότητα ως την

καταλληλότητα του προϊόντος για ένα συγκεκριμένο σκοπό. Το μειονέκτημα της παραπάνω θεώρησης είναι ότι η ποιότητα αξιολογείται σε ένα συγκεκριμένο πλαίσιο και έτσι μπορεί να είναι υποκειμενική.

1.2.3 Κατασκευαστική Θεώρηση (Manufacturing View)

Η κατασκευαστική θεώρηση επικεντρώνεται στην ποιότητα του προϊόντος κατά την παραγωγή και την παράδοσή του. Εξετάζει το κατά πόσο αυτό παρήχθη σωστά εξαρχής, με κύριο στόχο τη μείωση των δαπανών επανεπεξεργασίας του. Ωστόσο, αυτή η αντιμετώπιση οδηγεί σε αξιολόγηση ποιότητας που είναι ουσιαστικά ανεξάρτητη του παραγόμενου προϊόντος. Τα πρότυπα που στηρίζονται σε αυτήν τη θεώρηση (ISO9001, Capability Maturity Model) προασπίζουν τη συμμόρφωση στη διαδικασία παραγωγής και όχι στις προδιαγραφές.

Είναι λίγα τα στοιχεία εκείνα που επιδεικνύουν ότι η εφαρμογή των παραπάνω προτύπων εγγυάται ποιοτικά προϊόντα. Μάλιστα, επικριτές της δηλώνουν ότι η συμμόρφωση στη διαδικασία παραγωγής εγγυάται αποκλειστικά την ομοιόμορφη παραγωγή. Ωστόσο, η κριτική αυτή είναι αυστηρή. Παρότι τα πρότυπα βασίζονται στην αρχή της «τεκμηρίωσης των όσων υλοποιούνται και υλοποίησης των όσων τεκμηριώνονται», επιμένουν επίσης στη βελτίωση της παραγωγικής διαδικασίας ώστε να ενισχυθεί η ποιότητα του προϊόντος.

1.2.4 Θεώρηση Προϊόντος (Product View)

Η θεώρηση αυτή υποστηρίζει ότι η ποιότητα είναι άρρηκτα συνυφασμένη με τα έμφυτα χαρακτηριστικά του προϊόντος. Είναι μια θεώρηση που συχνά υιοθετείται από τους υπερασπιστές των μετρικών λογισμικού, οι οποίοι στηρίζουν ότι ο υπολογισμός και ο έλεγχος εσωτερικών ιδιοτήτων του προϊόντος οδηγούν σε βελτιωμένη συμπεριφορά του κατά τη χρήση. Η αξιολόγηση της ποιότητας με τη μέτρηση ιδιοτήτων είναι ελκυστική διότι προσφέρει ένα αντικειμενικό και ανεξάρτητο, από το

συγκεκριμένο πλαίσιο εργασίας, μέτρο. Ωστόσο, δεν έχει επιβεβαιωθεί ότι η εσωτερική ποιότητα προσφέρει και εξωτερική, ούτε ποιες πτυχές της επηρεάζουν τη χρήση του προϊόντος, που κρίνεται από το χρήστη.

1.2.5 Θεώρηση Βάσει Αξίας (Value-based View)

Κατά την παραγωγική διαδικασία, οι συμμετέχοντες ενδέχεται να έχουν διαφορετική άποψη για την ποιότητα του προϊόντος, ανάλογα με το ρόλο τους. Οι διαφορετικές οπτικές μπορούν να είναι συμπληρωματικές στα αρχικά στάδια παραγωγής, όμως ενδέχεται να προκύψουν συγκρούσεις σε επόμενα στάδια. Για παράδειγμα, αν στην πορεία γίνουν αλλαγές στις απαιτήσεις, το προϊόν θα πρέπει να επανασχεδιαστεί. Αυτό αντιτίθεται με την κατασκευαστική θεώρηση του ποιοτικού προϊόντος, στην οποία πρέπει να κατασκευάζεται εξαρχής σωστά.

Προκειμένου να αντιμετωπιστούν τέτοιου είδους συγκρούσεις, θεσπίζεται η θεώρηση της ποιότητας του λογισμικού βάσει της αξίας του. Σύμφωνα με αυτή, η ποιότητα του λογισμικού εκλαμβάνεται ως ανάλογη του ποσού που διατίθεται να πληρωθεί από το χρήστη για την παροχή του. Με αυτό τον τρόπο, αναγκάζονται όλοι οι συμμετέχοντες στην παραγωγή να αναλογισθούν τα trade-offs που δημιουργούνται μεταξύ κόστους και ποιότητας.

1.3 Στοιχεία που χαρακτηρίζουν ένα λογισμικό ποιοτικό

Για να επιτευχθεί η αξιολόγηση της ποιότητας λογισμικού απαιτούνται κοινά αποδεκτά κριτήρια. Αυτά πρέπει να καλύπτουν όλες τις θεωρήσεις προκειμένου η αξιολόγηση να είναι αντικειμενική και να υπάρχει σύγκλιση των διαφορετικών απόψεων. Ένας ακόμα λόγος για την ύπαρξη κριτηρίων αξιολόγησης είναι η ανάγκη σύγκρισης λογισμικών κοινού σκοπού. Τα βασικότερα μοντέλα για την αξιολόγηση του λογισμικού παρουσιάζονται στη συνέχεια.

1.3.1 Μοντέλο McCall

Ο McCall ήταν από τους πρώτους που προσπάθησαν να δώσουν ένα μοντέλο για τον χαρακτηρισμό της ποιότητας λογισμικού. Αυτό το μοντέλο προέρχεται από τον αμερικανικό στρατό και αρχικά αφορούσε στους προγραμματιστές και τη διαδικασία ανάπτυξης λογισμικού. Στο μοντέλο, ο McCall επιχειρεί να γεφυρώσει το χάσμα μεταξύ των χρηστών και των προγραμματιστών καθώς επικεντρώνεται σε ένα σύνολο παραγόντων ποιοτικού λογισμικού που αφορούν τόσο στην οπτική του χρήστη όσο και στις προτεραιότητες του προγραμματιστή.

Το πρότυπο του McCall εξετάζει την ποιότητα του λογισμικού βάσει τριών βασικών αξόνων. Αυτοί είναι[3][4]:

1. Αναθεώρηση λογισμικού

Ο άξονας αυτός αναφέρεται στη δυνατότητα του λογισμικού να υποστεί αλλαγές και περιλαμβάνει την αξιολόγηση των παρακάτω χαρακτηριστικών

- Συντηρησιμότητα (maintanability): εκφράζει το βαθμό ευκολίας τροποποίησης τους λογισμικού και της διόρθωσης των σφαλμάτων.
- Ευελιξία (flexibility): εκφράζει την ευκολία προσαύξησης της λειτουργικότητας του λογισμικού.
- Ελεγξιμότητα (testability): εκφράζει το βαθμό ευκολίας ελέγχου του λογισμικού και εντοπισμού της πηγής ενός σφάλματος.

2. Μεταφερσιμότητα λογισμικού

Ο άξονας αυτός αναφέρεται στη δυνατότητα του λογισμικού να μεταφερθεί σε άλλο περιβάλλον λειτουργίας και περιλαμβάνει τα παρακάτω χαρακτηριστικά:

- Μεταφερσιμότητα (portability): εκφράζει τη δυνατότητα λειτουργίας του λογισμικού σε διαφορετικά προγραμματιστικά περιβάλλοντα.
- Επαναχρησιμότητα (reusability): χαρακτηρίζει την ευκολία επαναχρησιμοποίησης κάποιων μονάδων λογισμικού σε άλλα προϊόντα λογισμικού.

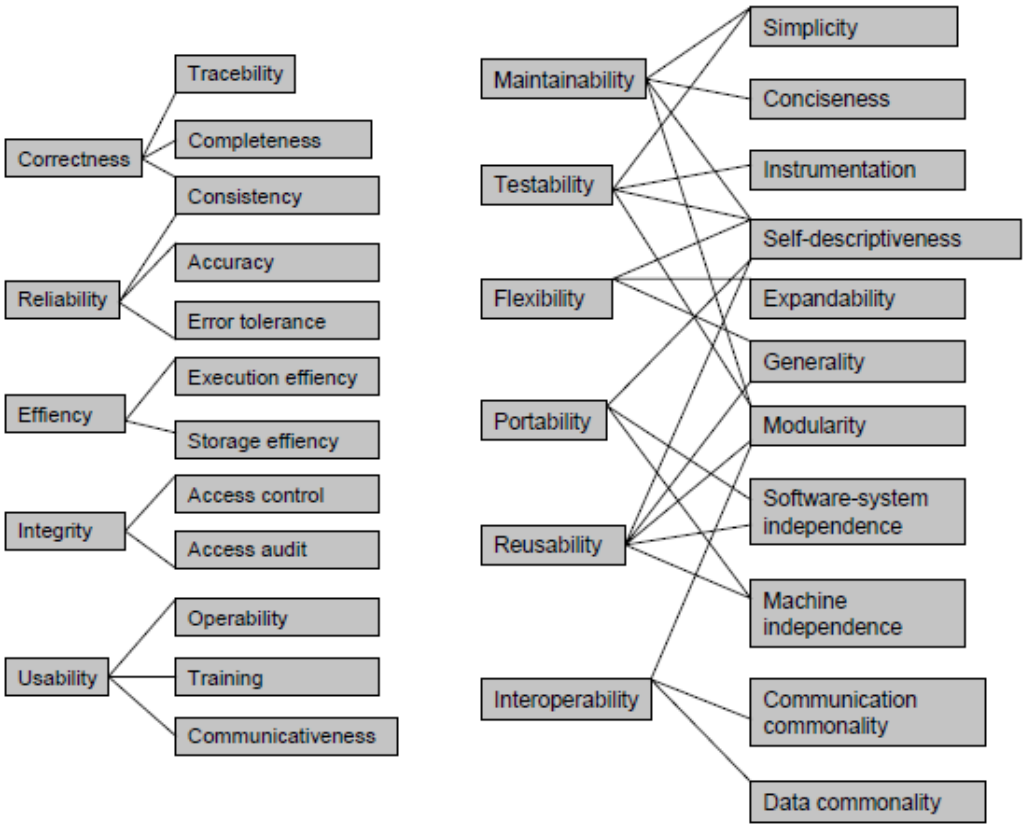
- Διαλειτουργικότητα (interoperability): εκφράζει την ευκολία με την οποία ένα σύστημα μπορεί να ανταλλάξει δεδομένα ή υπηρεσίες με ένα άλλο σύστημα.

3. *Λειτουργίες του λογισμικού*

Ο άξονας αυτός αφορά στο σύνολο των λειτουργιών που οφείλει να εκτελεί το λογισμικό. Η ποιότητα των λειτουργιών του συστήματος εξαρτάται από τα παρακάτω χαρακτηριστικά:

- Ορθότητα (correctness): χαρακτηρίζει το βαθμό στον οποίο το λογισμικό συμπεριφέρεται σύμφωνα με όσα αναφέρουν οι λειτουργικές απαιτήσεις.
- Αξιοπιστία (reliability): χαρακτηρίζει τη δυνατότητα του λογισμικού να συνεχίζει να λειτουργεί για κάποιο χρονικό διάστημα χωρίς αποτυχία και να εκτελεί τις απαιτούμενες λειτουργίες με την απαιτούμενη ακρίβεια.
- Αποδοτικότητα (efficiency): η αποδοτικότητα αφορά στην εκτέλεση του λογισμικού και στη χρήση των πόρων (π.χ. χρόνος εκτέλεσης, απαιτούμενη μνήμη).
- Ακεραιότητα (integrity): εκφράζει την ικανότητα του συστήματος να αντιστέκεται σε πιθανές απειλές από μη εξουσιοδοτημένους χρήστες. Επεκτείνει την ασφάλεια και σε άλλες απειλές όπως οι ιοί υπολογιστών και οι επιθέσεις άρνησης υπηρεσίας (denial of service).
- Ευχρηστία (usability): εκφράζει την ευκολία χρήσης του λογισμικού από τους τελικούς χρήστες, την ταχύτητα εκμάθησης των λειτουργιών του, την προσαρμοστικότητα του λογισμικού στις ανάγκες συγκεκριμένων χρηστών και άλλα θέμα στη χρήση του λογισμικού.

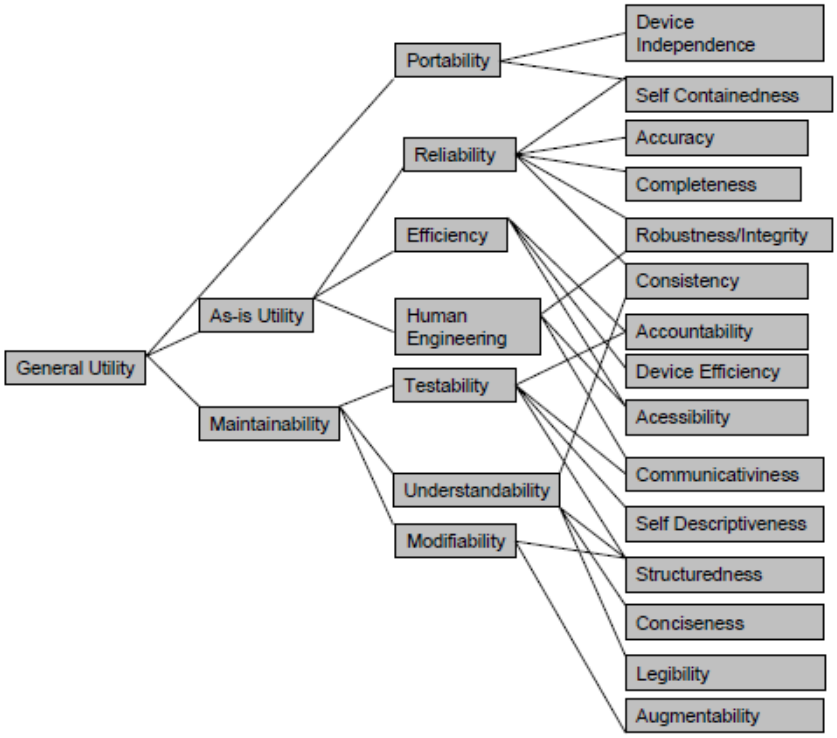
Το μοντέλο, εν συνεχεία, αναλύει τους τρεις βασικούς άξονες ποιοτικών χαρακτηριστικών σε μια ιεραρχία παραγόντων όπως αυτοί αναφέρθηκαν, κριτηρίων και μετρικών. Οι 11 αυτοί παράγοντες που αναλύθηκαν αφορούν στην οπτική του χρήστη όσον αφορά το λογισμικό από εξωτερική άποψη. Τα 23 κριτήρια που ακολουθούν στην ιεραρχία αφορούν στην εσωτερική δομή του λογισμικού και τους προγραμματιστές του συστήματος. Στην Εικόνα 1 παρουσιάζεται η ιεραρχία αυτή.



Εικόνα 1: Το ποιοτικό μοντέλο του McCall σε μια ιεραρχία 11 ποιοτικών παραγόντων (αριστερά) σε συσχέτιση με τα 23 κριτήρια ποιότητας (δεξιά) [3]

1.3.2 Μοντέλο Boehm

Παρόλο που τα μοντέλα του McCall και του Boehm (Εικόνα 2) φαίνονται να είναι παρόμοια, η διαφορά τους έγκειται στο γεγονός ότι ο McCall επικεντρώνεται κατά βάση στην ακριβή μέτρηση των υψηλού επιπέδου ποιοτικών χαρακτηριστικών, ενώ το μοντέλο του Boehm βασίζεται σε ένα εύρος χαρακτηριστικών με κύριο άξονα τη συντηρησιμότητα του λογισμικού. Η συντηρησιμότητα του κώδικα, όπως φαίνεται σε επόμενο διάγραμμα (Εικόνα 3), συμβάλει σε πολύ μεγάλο βαθμό στο κόστος ανάπτυξης του λογισμικού. Στον Πίνακα 1 συνοψίζονται οι διαφορές των μοντέλων του McCall και Boehm [3].



Εικόνα 2: Το δέντρο ποιοτικών χαρακτηριστικών του Boehm [3]

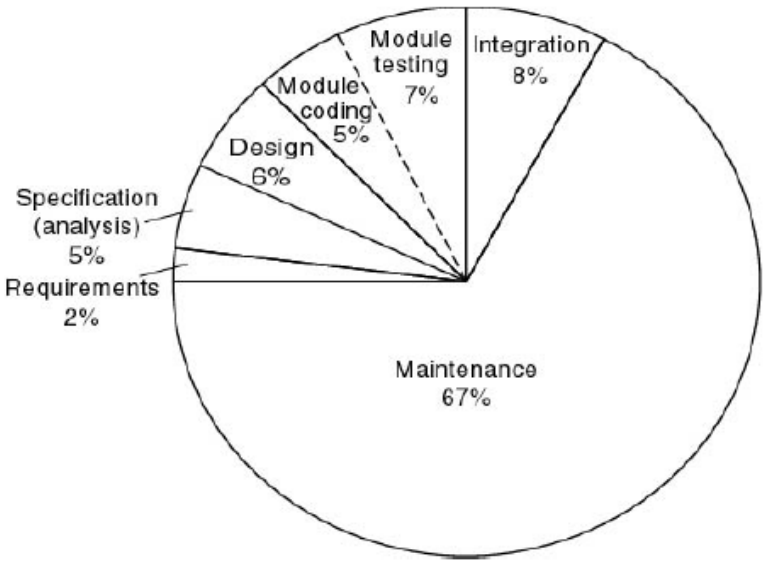
Criteria/goals	***McCall, 1977***	***Boehm, 1978***
Correctness	*	*
Reliability	*	*
Integrity	*	*
Usability	*	*
Effiency	*	*
Maintainability	*	*
Testability	*	
Interoperability	*	
Flexibility	*	*
Reusability	*	*
Portability	*	*
Clarity		*
Modifiability		*
Documentation		*
Resilience		*
Understandability		*
Validity		*
Functionality		
Generality		*
Economy		*

Πίνακας 1: Διαφορές των μοντέλων McCall και Boehm [3]

Υπάρχουν και άλλα μοντέλα αξιολόγησης ποιότητας λογισμικού τα οποία δεν αναλύονται περαιτέρω, ωστόσο ο ενδιαφερόμενος αναγνώστης μπορεί να απευθυνθεί στο αντίστοιχο λήμμα [3].

1.4 Κατανομή Κόστους Ανάπτυξης Λογισμικού

Στο παρακάτω διάγραμμα παρουσιάζεται η μέση κατανομή κόστους για την ανάπτυξη του λογισμικού και γίνεται σαφές γιατί τίθενται τα παραπάνω κριτήρια αξιολόγησης της ποιότητας του λογισμικού. Σε αντίθεση με ό,τι ενδεχομένως να πίστευε κανείς, η ανάπτυξη των τμημάτων του λογισμικού απαιτεί μόλις του 5% του κόστους ανάπτυξης του λογισμικού και η σύνθεση τους για την πλήρη ανάπτυξη του συστήματος μόλις το 8%. Η συντήρηση όμως του λογισμικού απαιτεί το 67% του κόστους ανάπτυξής του. Είναι επομένως ιδιαίτερης σημασίας ήδη από τα πρώτα στάδια ανάπτυξης του λογισμικού να λαμβάνεται υπόψιν αυτό το κόστος και η ανάπτυξη να γίνεται με βασικό γνώμονα την εύκολη συντήρησή του. Η αναδόμηση του συνολικού λογισμικού σε μικρότερες μονάδες λογισμικού είναι μία από τις τεχνικές που χρησιμοποιούνται για τη διευκόλυνση της συντηρησιμότητας του κώδικα.



Εικόνα 3: Μέση κατανομή κόστους ανάπτυξης λογισμικού

1.5 Συνεκτικότητα Κώδικα και Ποιότητα Λογισμικού

Οι αντικειμενοστραφείς γλώσσες προγραμματισμού, όπως η Java, και η ανάπτυξη περιβαλλόντων προγραμματισμού αποτελούν ένα πολύ σημαντικό βήμα στην κατασκευή λογισμικού το οποίο είναι συντηρήσιμο, επεκτάσιμο, κατανοητό και επαναχρησιμοποιήσιμο. Μεταξύ των χαρακτηριστικών που προσφέρουν οι αντικειμενοστραφείς γλώσσες είναι η ενθυλάκωση και η απόκρυψη της πληροφορίας, ο ορισμός κλάσεων από το χρήστη, η ανταλλαγή μηνυμάτων, η κληρονομικότητα και ο πολυμορφισμός [8]. Τα προγραμματιστικά περιβάλλοντα από την άλλη προσφέρουν την δυνατότητα χρήσης ήδη ορισμένων βιβλιοθηκών, κλάσεων και framework εφαρμογών, τυποποιημένων πρωτοκόλλων και εργαλείων ανάπτυξης [9].

Ωστόσο η χρήση αντικειμενοστραφών γλωσσών προγραμματισμού δεν είναι ικανή από μόνη της να οδηγήσει στην ποιοτική ανάπτυξη λογισμικού. Με τη χρήση τους η πολυπλοκότητα των συστημάτων ενδέχεται να αυξάνεται λόγω των πολλών αντικειμένων και των συσχετισμών μεταξύ τους. Επομένως μαζί με τις γλώσσες αντικειμενοστραφούς προγραμματισμού θα πρέπει να ορίσουμε και μεθόδους ανάδειξης καλής συγγραφής αντικειμενοστραφούς κώδικα, οι οποίες παράλληλα θα είναι σε θέση να εντοπίζουν τα προβληματικά σημεία του κώδικα. Η συνεκτικότητα αποτελεί μια από αυτές τις μεθόδους.

Η συνεκτικότητα (cohesion) είναι ένας από τους σημαντικότερους παράγοντες για να αξιολογήσουμε την ποιότητα του λογισμικού καθώς στόχο έχει, εν μέρει, τη μείωση της πολυπλοκότητας. Επομένως, σχετίζεται άμεσα με την συντηρησιμότητα, την αξιοπιστία και την επαναχρησιμότητα του κώδικα. Η συνεκτικότητα μιας μονάδας λογισμικού (module) ορίζεται ως η ποιοτική ιδιότητα που ως στόχο έχει την μέτρηση της μοναδικότητας του σκοπού της μονάδας. Μια μονάδα υψηλής συνεκτικότητας έχει μία βασική λειτουργία και είναι αδιάσπαστη [7]. Μία μονάδα λογισμικού χαμηλής ποιότητας μπορεί να αποτελεί σοβαρό μειονέκτημα στην ποιότητα του συνολικού λογισμικού. Προκειμένου, λοιπόν, να σχεδιαστεί ένα καλής ποιότητας λογισμικό, είναι επιθυμητό να οριστούν καλές μετρικές για την αξιολόγηση της συνεκτικότητας. [5]

Έχει υπολογισθεί ότι το 50% των υψηλής συνεκτικότητας μονάδων είναι ελεύθερο σφαλμάτων, σε αντίθεση με το μόλις 18% των χαμηλής συνεκτικότητας μονάδων. [6] Επίσης έχει αποδειχθεί ότι η υψηλή συνεκτικότητα οδηγεί σε χαμηλότερη αναλογία σφαλμάτων και ότι οι μονάδες με υψηλή σύζευξη (coupling) είναι περισσότερο επιρρεπείς σε σφάλματα. Η χαλαρή σύζευξη των διαδικασιών οδηγεί στην μικρότερη ανταλλαγή πληροφοριών μεταξύ τους στη ροή της εργασίας. Οι υψηλά συνεκτικές διαδικασίες από την άλλη, είναι πιθανότερο να γίνονται ευκολότερα κατανοητές και εκτελέσιμες από ότι μεγάλα κομμάτια μη σχετιζόμενων εργασιών που ομαδοποιούνται. Επειδή η δημιουργία μεγάλων διαδικασιών οδηγεί σε μικρότερο βαθμό σύζευξης και η δημιουργία μικρών διαδικασιών αυξάνει τη συνεκτικότητα, φαίνεται ότι η υψηλή συνεκτικότητα και η χαλαρή σύζευξη αποτελούν την καλύτερη επιλογή για τη βελτιστοποίηση της ποιότητας εκτέλεσης και της συντηρησιμότητας του λογισμικού. [6]

Κεφάλαιο 2

Ανάλυση Κώδικα

Παρακάτω αναφέρονται κάποιες από της βασικότερες τεχνικές ανάλυσης κώδικα που έχουν προταθεί προκειμένου να εξυπηρετήσουν στην προσπάθεια βελτίωσης κώδικα. Συγκεκριμένα θα γίνει αναφορά στα αφηρημένα συντακτικά δέντρα που αποτελούν μια μορφή αναπαράστασης του πηγαίου κώδικα σε δενδρική μορφή. Με τη βοήθεια των αφηρημένων συντακτικών δέντρων κατασκευάζονται οι γράφοι ροής ελέγχου (Control Flow Graph – CFG) οι οποίοι απεικονίζουν τη ροή ελέγχου του προγράμματος και αποτελούν υπογράφο του γράφου εξαρτήσεων προγράμματος (Program Dependence Graph – PDG). Ο τελευταίος αποτελεί απεικόνιση των εξαρτήσεων του προγράμματος είτε αυτές είναι εξαρτήσεις ελέγχου (control dependence) είτε εξαρτήσεις δεδομένων (data dependence). Με τη βοήθεια του PDG εξάγονται τα τεμάχια κώδικα (slices) που σχετίζονται με μία μεταβλητή και τις εξαρτήσεις της μεταβλητής αυτής από και προς τις διάφορες εντολές του κώδικα.

2.1 Αφηρημένο Συντακτικό Δέντρο (Abstract Syntax Tree)

Η κατασκευή του αφηρημένου συντακτικού δέντρου είναι ένα από το πρώτα βήματα στην ανάλυση κώδικα. Το αφηρημένο συντακτικό δέντρο αποτελεί ουσιαστικά την αναπαράσταση του πηγαίου κώδικα σε μια δενδρική μορφή. Η κατασκευή του συντακτικού δέντρου γίνεται από τους parsers. Φύλλα του δέντρου αποτελούν τα τερματικά σύμβολα της γραμματικής που χρησιμοποιείται. Ενδιάμεσοι κόμβοι είναι τα μη τερματικά σύμβολα που μπορούν να αναλυθούν περαιτέρω. Ουσιαστικά, κάθε κόμβος του δέντρου αποτελεί μια δομή του προγράμματος, όπως για παράδειγμα μια μέθοδο. Το αφηρημένο συντακτικό δέντρο αντιπροσωπεύει τη γενική δομή του προγράμματος και όχι συγκεκριμένες συντακτικές δομές.

Ένα αφηρημένο συντακτικό δέντρο (AST) χαρακτηρίζεται σύμφωνα με τον Klocwork [10] από τα παρακάτω στοιχεία:

- Αποτελεί μια απεικόνιση της συντακτικής δομής ενός λογισμικού
- Είναι περισσότερο συνυφασμένο με την τυπική ανάλυση από ότι με την συγκεκριμένη σύνταξη (concrete syntax) ή την επιφανειακή σύνταξη (surface syntax) που σχετίζονται με τον τρόπο με τον οποίο παριστάνεται ο κώδικας
- Αποτελεί μία πιο ακριβή τυποποίηση για τη λεπτομερή εύρεση γνώσης συγκριτικά με τους λιγότερο τυπικούς συλλέκτες γνώσεις (knowledge collectors) όπως scanners, tokenizers κ.ά.
- Αποτελεί ένα καλά ορισμένο MOF (Meta-Object Facility) μοντέλο, Meta-Model ή Meta-Meta-Model.

Επιπλέον ένα AST ενδέχεται [10]:

- Να αποτελεί μια αναστρέψιμη απεικόνιση που επιτρέπει την ανακατασκευή της επιφανειακής σύνταξης του υπάρχοντος συστήματος από την αφηρημένη σύνταξη (surface syntax <=> abstract syntax).
- Να μπορεί να εμπλουτιστεί με την ανάλυση περιορισμών (constrain analysis), ροών δεδομένων (data-flow analysis), ροών ελέγχου (control-flow analysis), αξιωμάτων (axiomatic analysis) και εντολών (denotational analysis) και επιπλέον με όψεις (views) όπως εννοιολογική (conceptual), σεναρίου (scenario), αλληλεπίδρασης (interaction), σύμπραξης (collaboration), κατασκευής (build) κ.ά.
- Να μπορεί να χρησιμοποιηθεί για την άντληση μετρικών μηχανικής λογισμικού (software engineering metrics) και τεκμηρίωσης.
- Να μπορεί να αντιστοιχηθεί (mapped) ή μετατραπεί σε άλλα αφηρημένα συντακτικά μοντέλα με τη χρήση κανόνων επανεγγραφής.
- Να μπορεί να ερωτάται και να χειρίζεται (manipulated) με την χρήση επερωτήσεων και γλωσσών χειρισμού μοντέλων (π.χ. QVT [43], JTGEN [44] και JRGEN [44]).

Στη θεωρία συντακτικής ανάλυσης (parsing) των γλωσσών προγραμματισμού η αφηρημένη σύνταξη (abstract syntax) μιας γλώσσας διαφοροποιείται από το AST μιας εφαρμογής. Η αφηρημένη σύνταξη είναι ένα μοντέλο που αποτελείται από τις δομικές μονάδες (constructs) από τις οποίες η γλώσσα συντίθεται. Το AST είναι ένα κατευθυνόμενο άκυκλο γράφημα (DAG) που αποτελείται από υλοποιήσεις (instances) των δομικών μονάδων της αφηρημένης συντακτικής γλώσσας. Μια καλά ορισμένη BNF ή EBNF (ή γραμματικές προδιαγραφές) χρησιμοποιείται για να ορίσει την αφηρημένη σύνταξη μιας γλώσσας προγραμματισμού. Ένας συντακτικός αναλυτής, από την άλλη, δημιουργεί το AST κάνοντας συντακτική ανάλυση μιας εφαρμογής σε μία γλώσσα ορισμένη από μια EBNF. Φυσικά μπορούν να χρησιμοποιηθούν και άλλες μέθοδοι πέραν της συντακτικής ανάλυσης για την κατασκευή ενός AST.

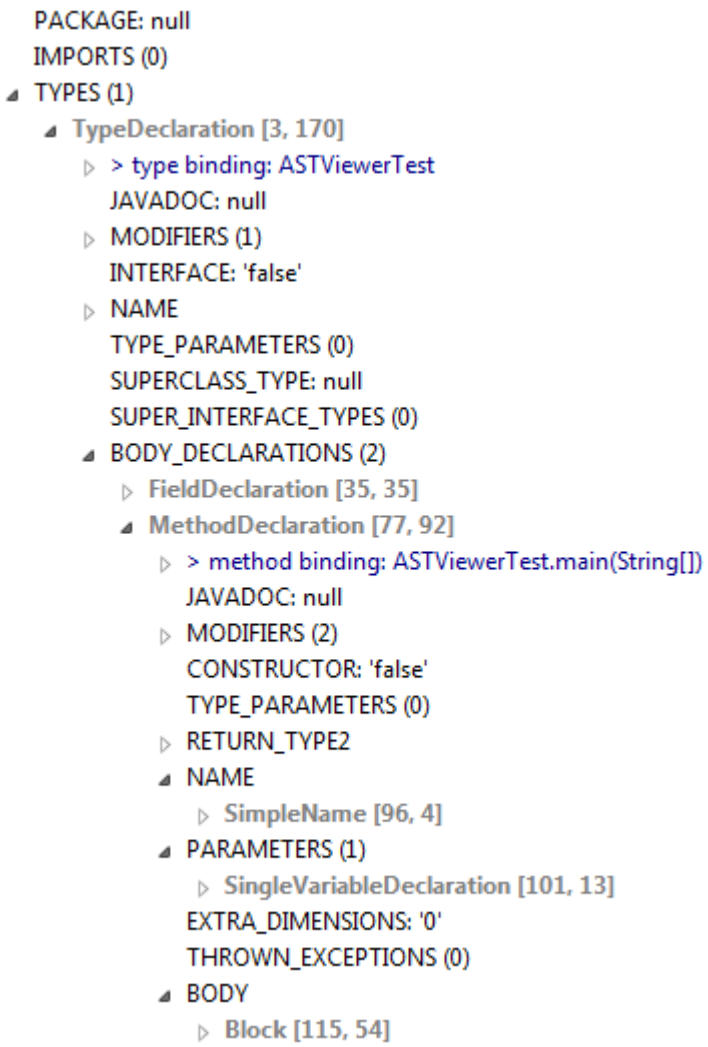
Στην πλατφόρμα ανάπτυξης λογισμικού του Eclipse [11] προσφέρεται ένα plug-in το οποίο δίνει την δυνατότητα επισκόπησης του συντακτικού δέντρου των προγραμμάτων, το AST Viewer [12]. Για λόγους πληρότητας παρακάτω παρουσιάζεται ενδεικτικός κώδικας και το AST όπως αυτό παράγεται από το plug-in.

```
public class ASTViewerTest {

    static String result = “successfull”;

    public static void main(String[] args) {
        System.out.print(“ASTViewer test ”+result);
    }
}
```

Πίνακας 2: Ενδεικτικό Πρόγραμμα

Το AST που προκύπτει από το ASTViewer για το συγκεκριμένο πρόγραμμα φαίνεται στην Εικόνα 4. Στο AST φαίνονται οι διάφορες παράμετροι του κώδικα (package, imports, types, κ.ά). Αυτές μπορούν να αναλυθούν περαιτέρω με την επιλογή επέκτασης, ώστε να παρουσιαστεί το δέντρο σε μεγαλύτερο βάθος. Στην

προκειμένη περίπτωση έχει επεκταθεί ο κλάδος BODY_DECLARATIONS (2) στον οποίο φαίνεται η δήλωση της μεταβλητής, που αποτελεί field (FieldDeclaration [35, 35]) και η δήλωση της μεθόδου (MethodDeclaration [77, 92]). Όλοι οι κλάδοι αναλύονται μέχρι να καταλήξουν σε φύλλα.



Εικόνα 4: Το AST για το πρόγραμμα του πίνακα 2 όπως προκύπτει από το ASTViewer

2.2 Γράφος Ροής Ελέγχου (Control Flow Graph)

Οι γράφοι ροής ελέγχου (CFGs) απεικονίζουν τη ροή ελέγχου μέσα σε ένα πρόγραμμα. Για κάθε CFG ισχύουν τα παρακάτω [14]:

- Οι εντολές του προγράμματος αποτελούν κόμβους του CFG και συνήθως αναφέρονται ως basic blocks.
- Οι ακμές είναι κατευθυνόμενες και αντιπροσωπεύουν τη ροή ελέγχου μεταξύ των basic blocks. Ακολουθούνται είτε μετά από την επαλήθευση κάποιας συνθήκης είτε κατά τη ροή του προγράμματος.
- Υπάρχει πάντα ένας και μοναδικός κόμβος εισόδου, ο οποίος μπορεί είτε να απεικονίζει κάποια εντολή είτε να είναι τεχνητός. Το τελευταίο συμβαίνει στην περίπτωση που περισσότερες της μιας εντολής μπορούν να αποτελέσουν την αρχή εκτέλεσης ενός συγκεκριμένου τμήματος κώδικα.
- Υπάρχει πάντα ένας και μοναδικός κόμβος εξόδου, ο οποίος μπορεί είτε να απεικονίζει κάποια εντολή είτε να είναι τεχνητός. Αυτό συμβαίνει όταν υπάρχουν περισσότερες της μιας εντολής στην οποία μπορεί να σταματήσει η εκτέλεση ενός συγκεκριμένου τμήματος κώδικα.

Οι προσεγγίσεις που ακολουθούνται διαφέρουν ανάλογα με τον τρόπο αντιμετώπισης των κλάδων, τον τρόπο συνένωσής τους και την παρουσίαση των εντολών που εκτελούνται πάντα μαζί και ακολουθιακά. Δηλαδή ένας κόμβος του CFG μπορεί να αποτελεί ένα σύνολο εντολών του προγράμματος. Ένα τέτοιο σύνολο από εντολές ονομάζεται τμήμα (segment) [14]. Όταν τα basic blocks αποτελούνται από τμήματα, για το CFG ισχύει επίσης ότι:

- Μοναδικός κόμβος του τμήματος από τον οποίο μπορούν να εξέρχονται ακμές είναι η τελευταία εντολή του τμήματος.
- Μοναδικός κόμβος του τμήματος στον οποίο μπορούν να εισέρχονται ακμές είναι η πρώτη εντολή του τμήματος.

Από τα παραπάνω γίνεται σαφές ότι το CFG αποτελεί την αναπαράσταση των μονοπατιών που δύναται να διασχιστούν κατά την εκτέλεση του προγράμματος. Επίσης, θα πρέπει να σημειωθεί ότι δεν είναι όλα τα μονοπάτια προσπελάσιμα (feasible).

Βασικός λόγος για την δημιουργία των CFG είναι ο έλεγχος των μονοπατιών ενός προγράμματος για την αποσφαλμάτωσή του. Υπάρχουν μετρικές οι οποίες βασίζονται στο CFG που έχουν ως στόχο την αξιολόγηση της ποιότητας ενός τμήματος κώδικα. Ωστόσο το CFG χρησιμοποιείται και για την εξαγωγή του γράφου εξαρτήσεων ενός προγράμματος (Program Dependence Graph – PDG) που αναλύεται παρακάτω.

2.3 Γράφος Εξαρτήσεων (Program Dependence Graph)

Η απεικόνιση της εσωτερικής δομής ενός προγράμματος γίνεται με την χρήση με του γράφου εξαρτήσεων (Program Dependence Graph – PDG). Το PDG αναφέρεται τόσο στην εξάρτηση δεδομένων στο πρόγραμμα (data dependence) όσο και στην εξάρτηση των ελέγχων για κάθε λειτουργία του προγράμματος (control dependence).

Το PDG αναπαριστά το πρόγραμμα ως έναν γράφο στον οποίο [13]:

- Οι εντολές ή οι δηλώσεις μεταβλητών ή οι τελεστές και οι τελεσταίοι αποτελούν τους κόμβους του.
- Οι ακμές που εφάπτονται στους κόμβους αντιπροσωπεύουν:
    1. τις τιμές των δεδομένων απ' τις οποίες εξαρτώνται οι λειτουργίες του κόμβου ή
    2. τις συνθήκες ελέγχου από τις οποίες εξαρτάται η εκτέλεση των ενεργειών.

Η εξάρτηση προκύπτει ως αποτέλεσμα δύο διαφορετικών φαινομένων.

1. Εξάρτηση μεταξύ δύο εντολών όταν υπάρχει μια μεταβλητή σε μια εντολή που η τιμή της αλλοιώνεται όταν η σειρά των εντολών αναστραφεί. Αυτού του είδους οι εξαρτήσεις αποτελούν τις εξαρτήσεις δεδομένων (data dependences).
2. Εξάρτηση μεταξύ δύο εντολών όταν η τιμή μιας μεταβλητής ελέγχει άμεσα την εκτέλεση μιας εντολής. Τέτοιες εξαρτήσεις ονομάζονται εξαρτήσεις ελέγχου (control dependences). Η ανάλυση ροής ελέγχου χρησιμοποιείται για την κατασκευή του υπογράφου εξαρτήσεων ελέγχου (Control Dependency Graph – CDG).

Οι εξαρτήσεις δεδομένων (data dependencies) μπορούν να κατηγοριοποιηθούν περαιτέρω σε εξαρτήσεις βρόγχου (loop-carried) και ανεξάρτητες βρόγχων (loop-independent). Μια loop-carried εξάρτηση προκύπτει από τις επαναλήψεις του βρόγχου. Αντίθετα μια loop-independent εξάρτηση προκύπτει από την σειρά εκτέλεσης ανεξαρτήτως της ύπαρξης βρόγχων.

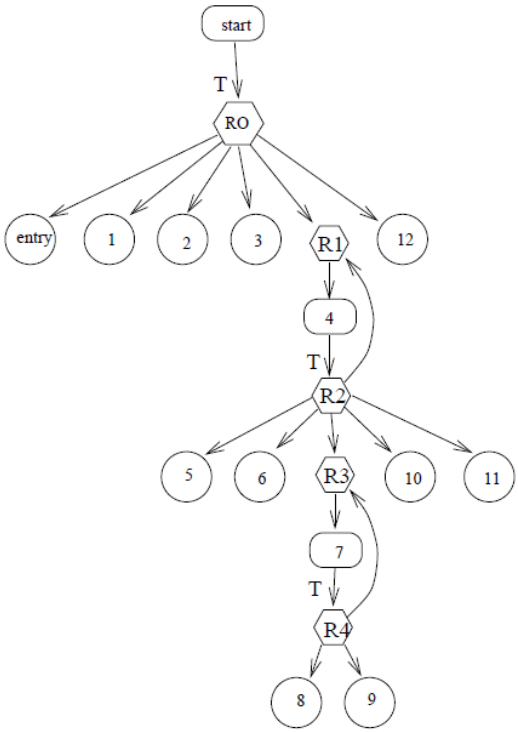
Ένας πιο φορμαλιστικός ορισμός για την εξάρτηση δεδομένων δίδεται από τους Hennessy και Patterson [16]. Σύμφωνα με τον ορισμό που δίδεται, μια εντολή (instruction) j είναι data dependent από μια άλλη εντολή i αν ισχύει μια από τις παρακάτω συνθήκες:

- Η εντολή i παράγει ένα αποτέλεσμα που μπορεί να χρησιμοποιηθεί από την εντολή j.
- Η εντολή j είναι data dependent από την εντολή k που με τη σειρά της είναι data dependent από την εντολή i. Δηλαδή όταν υπάρχει μια αλυσίδα εξαρτήσεων από το i στο j.

Θα πρέπει να διευκρινιστεί ότι μια εξάρτηση μέσα στην ίδια εντολή, όπως για παράδειγμα a+=a, δεν θεωρείται εξάρτηση. Από τα παραπάνω γίνεται σαφές ότι όταν υπάρχει εξάρτηση δεδομένων μεταξύ δύο εντολών αυτές δεν μπορούν να εκτελεστούν παράλληλα ή να επικαλυφθούν.

Στην Εικόνα 5 παρουσιάζεται ένα τμήμα κώδικα μαζί με το αντίστοιχο CDG που αποτελεί υπογράφο του PDG. Στον Πίνακα 3 συνοψίζονται οι ακμές του PDG για τον κώδικα της εικόνας 5.

```
Program Sums
1.      read(n);
2.      i = 1;
3.      sum = 0;
4.      while (i<= n) do
5.          sum = 0;
6.          j = 1;
7.          while (j <= i) do
8.              sum = sum + j;
9.              j = j + 1;
            endwhile;
10.         write(sum, i);
11.         i = i + 1;
        endwhile;
12.     write(sum, i);
end Sums
```



Εικόνα 5: Τμήμα κώδικα με τον αντίστοιχο CDG [15]

PDG node	control dependence		data dependence	
	successors	predecessors	successors	predecessors
entry	none	R0	none	none
1	none	R0	4	none
2	none	R0	4,7,10,11,12	none
3	none	R0	12	none
4	R2	R1	none	1,2,11
5	none	R1	8,10,12	none
6	none	R1	7,8,9	none
7	R4	R3	none	2,6,9,11
8	none	R4	8,10,12	5,6,8,9
9	none	R4	7,8,9	6,9
10	none	R2	none	2,5,8,11
11	none	R2	4,7,10,11,12	2,11
12	none	R0	none	2,3,5,8,11
exit	none	R0	none	none
R0	entry,1,2,3,R1,12,exit	none	none	none
R1	4	R0,R2	none	none
R2	3,6,R3,10,11,R1	4	none	none
R3	7	R2,R4	none	none
R4	8,9,R3	7	none	none

Πίνακας 3: Σύνοψη του PDG γράφου για την εικόνα 5 [15]

Οι Hennessy και Patterson [16] παρουσιάζουν κι ένα άλλο είδος εξαρτήσεων δεδομένων, τις ονοματικές εξαρτήσεις (name dependencies). Αυτές είναι οι ακόλουθες.

- Anti-dependence: Καλείται η εξάρτηση μεταξύ δύο μεταβλητών όταν η μία διαβάζει το περιεχόμενο μιας θέσης μνήμης στην οποία έγραψε μια άλλη. Σε αυτή την περίπτωση η σειρά των εντολών δεν πρέπει να αντιστραφεί.
- Output dependence υπάρχει στην περίπτωση που δύο μεταβλητές γράφουν στην ίδια θέση μνήμης. Και σε αυτήν την περίπτωση η σειρά των εντολών πρέπει να διατηρηθεί.

Οι παραπάνω κατηγορίες ανήκουν σε ξεχωριστή ομάδα εξαρτήσεων και όχι στις εξαρτήσεις δεδομένων καθώς δεν υπάρχει πραγματική συσχέτιση τιμών ή πέρασμα τιμών από μια εντολή σε άλλη.

2.4 Slicing Προγράμματος

Ο τεμαχισμός (slicing) αποτελεί μια μέθοδο διαχωρισμού του πηγαίου κώδικα. Εφαρμόζεται αφότου έχει ολοκληρωθεί η συγγραφή του, επομένως είναι μια διαδικασία χρήσιμη κατά τη διάρκεια της συντήρησης του προγράμματος και όχι κατά το σχεδιασμό. Στόχο έχει την αποδόμηση μιας σύνθετης μεθόδου σε μικρότερες με τέτοιο τρόπο ώστε τα τεμάχια (slices) που προκύπτουν να έχουν την ίδια λειτουργικότητα με το αρχικό πρόγραμμα. Αυτό επιτυγχάνεται με την εξαγωγή του συνόλου των εντολών που επηρεάζουν την τιμή μιας συγκεκριμένης μεταβλητής σε ένα τμήμα κώδικα. Υπάρχουν συνήθως πολλά διαφορετικά slices για το ίδιο τμήμα κώδικα και το ίδιο κριτήριο τεμαχισμού, ανάλογα με τον επιθυμητό βαθμό απλότητας του slice. Κριτήριο τεμαχισμού (P,V) ορίζεται το ζεύγος του συνόλου μεταβλητών V με το σημείο P του υπό εξέταση κώδικα. Τα slices που έχουν ενδιαφέρον είναι τα σημαντικά μικρότερα και απλούστερα από το αρχικό πρόγραμμα. [17]

Αναλυτικότερα, το slice ενός προγράμματος με κριτήριο τεμαχισμού (P,V) αποτελείται από όλες τις εντολές και τα κατηγορήματα που επηρεάζουν τις τιμές των μεταβλητών του V στο σημείο P. [15]

Σύμφωνα με τον ορισμό που δόθηκε από τον Weiser υπάρχουν δύο επιθυμητές ιδιότητες σε ένα slice. Πρώτον, το slice θα πρέπει να προκύπτει από το αρχικό πρόγραμμα με τη διαγραφή εντολών. Δεύτερον, τα slices θα πρέπει να έχουν την ίδια συμπεριφορά με το αρχικό πρόγραμμα με τέτοιο τρόπο ώστε να μην μπορεί να γίνει διάκριση του αρχικού προγράμματος και του slice [17]. Κατά τον Weiser τα slices αποτελούν από μόνα τους εκτελέσιμα τμήματα κώδικα. Ωστόσο υπάρχουν ορισμοί [15] που περιγράφουν τα slices ως ένα υποσύνολο εντολών και κατηγορημάτων ελέγχου που, άμεσα ή έμμεσα, επηρεάζουν τις τιμές των μεταβλητών στο κριτήριο τεμαχισμού. Το υποσύνολο αυτό δεν αποτελεί απαραίτητα εκτελέσιμο τμήμα κώδικα από μόνο του.

Τα slices διακρίνονται σε στατικά (static slices) και δυναμικά (dynamic slices). Τα στατικά slices υπολογίζονται για οποιαδήποτε είσοδο στο πρόγραμμα, χωρίς να λαμβάνονται υπόψιν περιορισμοί εισόδου, ενώ τα δυναμικά slices υπολογίζονται βάσει συγκεκριμένης εισόδου του προγράμματος (test case). Τα στατικά slices μπορεί

να είναι backward ή forward. Δοσμένου ενός PDG γράφου, ένα slice με κριτήριο τεμαχισμού (P,v) μπορεί να υπολογισθεί με την προς τα πίσω διάσχιση από το P προς όλους τους κόμβους του PDG στους οποίους το P έχει πρόσβαση (backward slicing) [15]. Από την άλλη, το forward slice αποτελείται από όλες τις εντολές και τα κατηγορήματα ελέγχου που εξαρτώνται από το (P,v). Μια δήλωση εξαρτάται από το κριτήριο τεμαχισμού σε δύο περιπτώσεις. Στην πρώτη περίπτωση, οι τιμές των μεταβλητών που υπολογίζονται στη δήλωση εξαρτώνται από τις τιμές που υπολογίζονται στο κριτήριο τεμαχισμού. Στη δεύτερη περίπτωση, οι τιμές που υπολογίζονται στο κριτήριο τεμαχισμού καθορίζουν την εκτέλεση ή μη της συγκεκριμένης δήλωσης. [18] Το forward slice υπολογίζεται με παρόμοιο τρόπο, με τη διαφορά ότι η διάσχιση γίνεται προς τα εμπρός αντί για προς τα πίσω.

Για τη δημιουργία των στατικών slices λαμβάνονται υπόψιν όλες οι εξαρτήσεις ενός προγράμματος. Αντίθετα, για τη δημιουργία των δυναμικών slices λαμβάνονται υπόψη μόνο οι εξαρτήσεις που σχετίζονται με μια συγκεκριμένη εκτέλεση του προγράμματος. Ένα κριτήριο δυναμικού τεμαχισμού ορίζει εκτός των P και V και την είσοδο ενός προγράμματος. Έτσι, μια εντολή μπορεί να περιέχεται σε κάποιο slice, αλλά να απουσιάζει από κάποιο άλλο, ακόμα κι αν εξετάζεται το ίδιο ζεύγος (P,V). Τυπικά, ένα κριτήριο δυναμικού τεμαχισμού αποτελείται από μια τριάδα (είσοδος, σημείο κώδικα, μεταβλητή). Επομένως, για τον δυναμικό τεμαχισμό, η είσοδος είναι προκαθορισμένη (fixed), σε αντίθεση με τον στατικό τεμαχισμό.

	Computation Method [a]	Interprocedural Solution	Control Flow [b]	Data Types [c]	Concurrency
Weiser	D	yes	S	S	no
Lyle	D	no	A	S, A	no
Gallagher, Lyle	D	no	A[d]	S	no
Jiang et al.	D	yes	A[d]	S, A, P[e]	no
Lyle, Binkley	D	no	S[f]	S, P	no
Hausler	F	no	S	S	no
Bergeretti, Carré	I	yes[g]	S	S	no
Ottenstein	G	no	S	S, A	no
Horwitz et al.	G	no	S	S	no
Horwitz et al.	G	yes	S	S	no
Binkley	G	yes[h]	S	S	no
Binkley	G	yes[i]	S	S	no
Jackson, Rollins	G	yes	S	S	no
Reps et al.	G	yes	S	S	no
Lakhotia	G	yes	S	S	no
Agrawal et al.	G	no	S	S, A, P	no
Ball, Horwitz	G	no	A	S	no
Choi, Ferrante	G	no	A	S	no
Agrawal	G	no	A	S	no
Cheng	G	no	S	S	yes
Ernst	O	yes	A	S, A, P	no
Field et al.	R	no	S	S, P	no

Πίνακας 4: Σύνοψη αλγορίθμων για την εξαγωγή slices [18]

[a]D = data-flow equations, F = functional/denotational semantics, I = information-flow relations, G = reachability in a dependence graph, O = dependence graphs in combination with optimization techniques . R = dependence tracking in term graph rewriting systems.

[b]S = structured, A = arbitrary.

[c]S = scalar variables, A = arrays/records, P = pointers.

[d]Solution incorrect (see [18]).

[e]Solution incorrect (see [18]).

[f]Only straight-line code is considered.

[g]Non-recursive procedures only.

[h]Takes parameter aliasing into account.

[i]Produces slices that are executable programs.

2.5 Block-Based Slicing

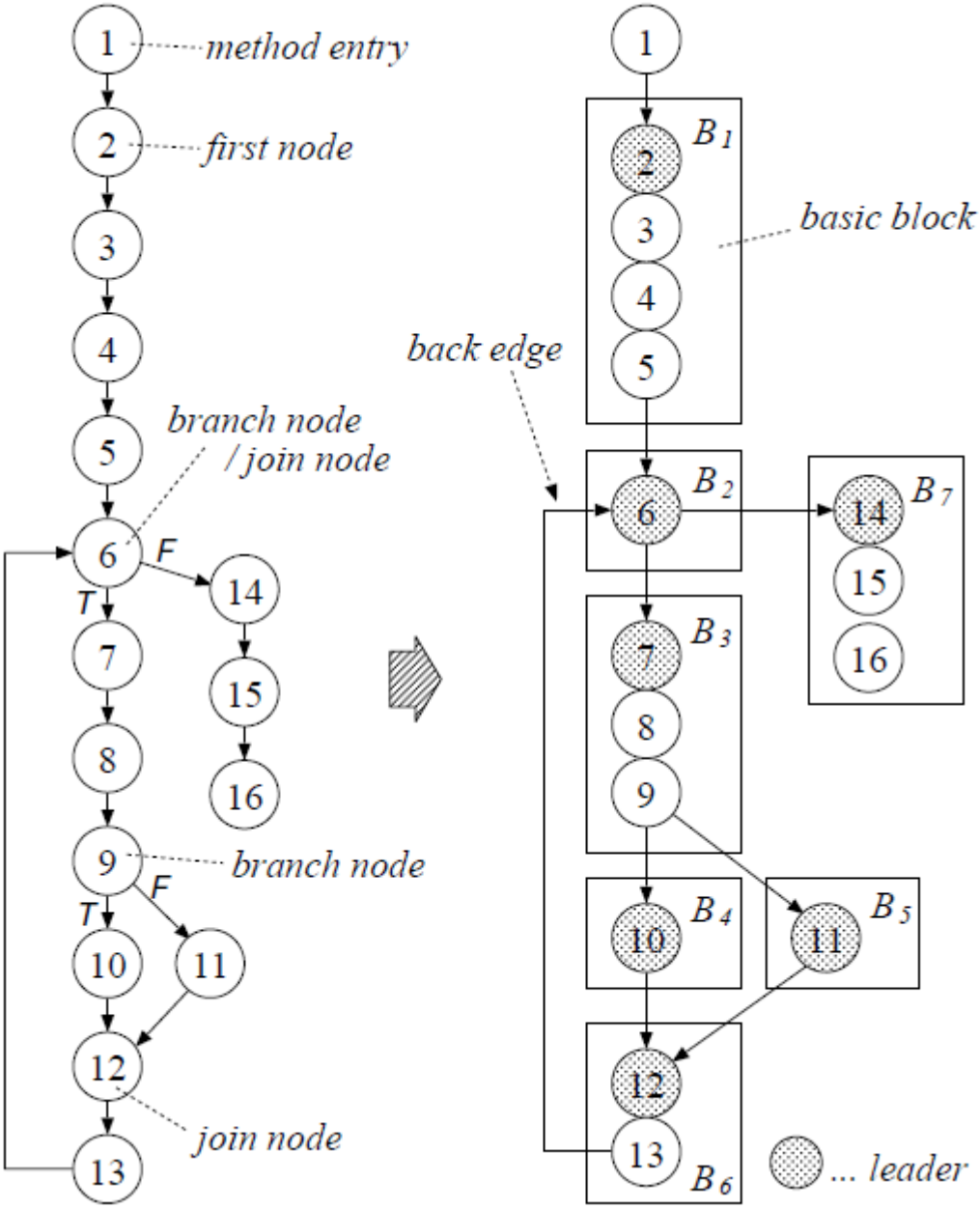
Για τον υπολογισμό των slices έχουν δοθεί διάφοροι αλγόριθμοι. Οι ιδιότητες κάποιων εξ αυτών συνοψίζονται στον Πίνακα 4. Ο Maruyama [19] ανέπτυξε έναν άλλο αλγόριθμο υπολογισμού των slices, το block based slicing. Ο μηχανισμός αυτός ασχολείται με πηγαίο κώδικα γραμμένο σε αντικειμενοστραφή γλώσσα προγραμματισμού, ο οποίος δεν περιέχει μη δομημένη ροή ελέγχου (όπως break ή continue) και οι εξαρτήσεις όλων των εντολών που περιέχει μπορούν να αναλυθούν στατικά. Τα Basic Blocks ορίζονται ως η μεγαλύτερη ακολουθία εντολών που μπορούν να εκτελεστούν χωρίς εσωτερικά ή εξωτερικά άλματα.

Στον Πίνακα 5 φαίνεται ένα ενδεικτικό κομμάτι κώδικα γραμμένο σε Java, ενώ στην Εικόνα 6 το αντίστοιχο CFG και τα basic-blocks όπως αυτά διαμορφώνονται. Ως leader node του block χαρακτηρίζεται ο κόμβος-εντολή για τον οποίο ισχύει ένα από τα παρακάτω στο CFG:

1. Είναι ο πρώτος κόμβος
2. Είναι κόμβος στον οποίο συνευρίσκονται ροές ελέγχου
3. Είναι κόμβος ο οποίος ακολουθεί έναν κόμβο διακλάδωσης (branch node) (συνθήκες if, while ή for)

```
 class Customer {
     private Vector _rentals = new Vector();
     ...
1    public String statement() {
2        double totalAmount = 0;
3        int renterPoints = 0;
4        Enumeration rentals = _rentals.elements();
5        String result = "Rental Record\n";
6        while (rentals.hasMoreElements()) {
7            Rental each =
(Rental)rentals.nextElement();
8            double thisAmount = each.getCharge();
9            if (each.getMovie().getPriceCode() ==
Movie.NEW_RELEASE)
10               renterPoints = renterPoints + 2;
             else
11               renterPoints++;
12           result = result +
each.getMovie().getTitle() + "\t" +
String.valueOf(thisAmount) + "\n";
13           totalAmount = totalAmount + thisAmount;
         }
14       result = result + "Amount: " +
String.valueOf(totalAmount) + "\n";
15       result = result + "Points: " +
String.valueOf(renterPoints);
16       return result;
     }
 }
```

Πίνακας 5: Ενδεικτικό τμήμα κώδικα γραμμένο σε Java [19]



Εικόνα 6: CFG (δεξιά) και Basic Blocks (αριστερά) για το πρόγραμμα του Πίνακα 4 [19]

Στη συνέχεια δίδονται κάποιοι ορισμοί που αφορούν στα basic-blocks όπως αυτοί διατυπώνονται από τον Maruyama [19].

- Ως Reach(B) ορίζονται τα βασικά blocks τα οποία είναι προσβάσιμα από ένα βασικό block B με προς τα εμπρός διάσχιση, χωρίς να ακολουθηθούν οι προς τα πίσω ακμές των βρόγχων.

$$Reach(B) = \{ B_i \in B_B \mid B \Rightarrow^* B_i \}$$

όπου B_B είναι το σύνολο όλων των βασικών blocks της μεθόδου στον PDG γράφο.

Έτσι για το παραπάνω παράδειγμα, τα Reach των blocks είναι τα παρακάτω:

Reach(B_1*)* = (B_1;B_2;B_3;B_4;B_5;B_6;B_7)

Reach(B_2*)* = (B_2;B_3;B_4;B_5;B_6;B_7)

Reach(B_3*)* = (B_3;B_4;B_5;B_6)

Reach(B_4*)* = (B_4;B_6)

Reach(B_5*)* = (B_5;B_6)

Reach(B_6*)* = (B_6)

Reach(B_7*)* = (B_7)

- Ένα block θεωρείται ότι κυριαρχείται από ένα κόμβο r αν υπάρχει μεταβατική εξάρτηση ελέγχου από το r στο block.

$$Dom(B) = \{ B_i \in B_B \mid r \rightarrow_c^* B_i \\ \wedge\, r \rightarrow_c leader(B) \wedge r \in N(G) \}$$

όπου N(G) είναι το σύνολο όλων των κόμβων στο PDG γράφο της μεθόδου.

Τα κυρίαρχα blocks για το προηγούμενο παράδειγμα είναι τα παρακάτω

Dom(B_1*)* = *Dom(*B_2*)* = *Dom(*B_7*)* = (B_1;B_2;B_3;B_4;B_5;B_6;B_7)

Dom(B_3*)* = *Dom(*B_6*)* = (B_3; B_4;B_5;B_6)

Dom(B_4*)* = *Dom(*B_5*)* = (B_4; B_5)

- Ως συνοριακά block (boundary) ενός κόμβου n ορίζονται τα blocks για τα οποία ισχύει

$$Blocks(n) = \{ B \in B_B \mid n \in C \\ \wedge C \in [\, Reach(B) \cap Dom(B) \,] \}$$

- Τέλος, ως περιοχή (region) ορίζεται μια ακολουθία διαδοχικών βασικών blocks. Ως μια περιοχή βάσει κάποιου block (block-based region – $R(B_n)$), όπου $B_n \in$ *Blocks(*n*)*, ορίζεται το σύνολο των κόμβων που βρίσκονται σε βασικά blocks που είναι είναι προσβάσιμα από το B_n.

Στη συνέχεια, περιγράφεται συνοπτικά η μέθοδος που ακολουθείται από τον Maruyama για την εξαγωγή των slices.

1. Είσοδος για την εξαγωγή των slices αποτελεί όπως και στις προηγούμενες περιπτώσεις ένα κριτήριο τεμαχισμού, έστω (n, u) για την εντολή n της μεθόδου m και τη μεταβλητή u, η οποία ορίζεται ή χρησιμοποιείται στην εντολή n.

2. Αρχικά, κατασκευάζεται ο γράφος ροής ελέγχου (CFG) για τη μέθοδο προκειμένου να εξαχθούν τα βασικά blocks.

3. Στη συνέχεια, κατασκευάζεται ο γράφος εξαρτήσεων (PDG) της μεθόδου, ο οποίος περιέχει όλες τις ακμές εξάρτησης ελέγχου και δεδομένων μεταξύ των εντολών της μεθόδου.

4. Το σύνολο των συνοριακών blocks, Blocks(n), υπολογίζεται για την εντολή n του κριτηρίου τεμαχισμού.

5. Για κάθε συνοριακό block $B_n \in Blocks(n)$, κατασκευάζεται ένας υπογράφος του PDG ο οποίος περιέχει μόνο τους κόμβους που ανήκουν στην block-based περιοχή του B_n.

6. Το block-based slice SB(n, u, Bn) για το κριτήριο τεμαχισμού (n, u) και το συνοριακό block Bn είναι το σύνολο των εντολών που μπορεί να επηρεάζουν τον

υπολογισμό της μεταβλητής u στην εντολή n (backward slice), όπως εξήχθη από τον υπογράφο εξαρτήσεων του προγράμματος που αντιστοιχίζεται στο $R(B_n)$.

Η προσέγγιση που ακολουθείται από τον Maruyama δίνει την δυνατότητα της εξαγωγής περισσότερων του ενός slices για το ίδιο κριτήριο τεμαχισμού με τη χρήση κατάλληλων block-based περιοχών, σε αντίθεση με τους κλασικούς αλγορίθμους τεμαχισμού που υπολογίζουν ένα slice για ένα δεδομένο κριτήριο τεμαχισμού χρησιμοποιώντας όλη τη μέθοδο ως περιοχή. [20]

Το backward slicing σχηματίζει τεμάχια τα οποία περιέχουν όλες τις εντολές και τις συνθήκες ελέγχου που μπορεί να επηρεάσουν την τιμή μιας μεταβλητής σε ένα δεδομένο σημείο. Ωστόσο υπάρχει και το forward slicing, τα τεμάχια του οποίου περιέχουν όλες τις εντολές και τις συνθήκες ελέγχου που ενδέχεται να επηρεαστούν από την τιμή μιας μεταβλητής σε ένα δεδομένο σημείο [21][22]. Ο αλγόριθμος για την εφαρμογή του forward slicing μπορεί να είναι παρόμοιος με αυτόν του backward slicing, αλλά ακολουθείται η αντίθετη πορεία στο γράφο εξαρτήσεων.

Ο συνδυασμός των backward και forward slicing ο οποίος προτείνεται από τους Ott και Thuss [21] αποτελεί το metric slicing. Το slice που προκύπτει περιέχει τις εντολές των slices που εξάγονται από το forward slicing και τις εντολές των slices που εξάγονται από το backward slicing.

2.6 Slice Profiles

Το slice profile αποτελεί μια μέθοδο οπτικοποίησης των slices και προτάθηκε αρχικά από τους Ott και Thuss [21]. Παρακάτω παρουσιάζεται το slice profile της μεθόδου gamma με την εφαρμογή των διαφόρων ειδών slicing (forward, backward και metric).

Statement	Backward Slice			Forward Slice			Metric Slice		
	important Value1	important Value2	important Value3	important Value1	important Value2	important Value3	important Value1	important Value2	important Value3
1	|	|	|	|			|	|	|
2		|	|		|			|	|
3		|	|	|			|	|	|
4		|	|	|	|		|	|	|
5			|	|	|	|	|	|	|
6				|	|	|	|	|	|

Πίνακας 6: Slice Profile (Backward, Forward, Metric) της μεθόδου gamma

Για το backward slicing, ο υπολογισμός των slices γίνεται ξεκινώντας από την τελευταία δήλωση στην οποία η μεταβλητή ορίζεται και διασχίζοντας τον κώδικα προς τα πίσω, προσθέτοντας στο slice τις εντολές που συμβάλλουν στη διαμόρφωση της τελικής τιμής της μεταβλητής. Ο υπολογισμός του forward slicing γίνεται προς την αντίθετη κατεύθυνση, δηλαδή ξεκινώντας από την πρώτη δήλωση στην οποία η μεταβλητή ορίζεται και διασχίζοντας τον κώδικα προς τα εμπρός, προσθέτοντας εκείνες τις εντολές των οποίων ο υπολογισμός καθορίζεται από την τιμή της μεταβλητής. Το metric slicing προκύπτει από την ένωση του backward και forward slicing.

2.7 Μετρικές Βασισμένες σε Slices

Η συνεκτικότητα ενός προγράμματος αποτελεί μέτρο ποιότητας του πηγαίου κώδικα του προγράμματος. Ωστόσο από μόνη της δεν αποτελεί μετρήσιμο μέγεθος. Επομένως θα πρέπει να διατυπωθούν κάποιες βασικές μετρικές για την ποσοτικοποίηση της συνεκτικότητας ενός τμήματος κώδικα. Στη συνέχεια αναφέρονται κάποιες μετρικές, οι οποίες βασίζουν τους υπολογισμούς τους στα slices. Ονομαστικά οι μετρικές που παρουσιάζονται είναι [17][21]:

1. Coverage
2. Overlap

3. Parallelism
4. Tightness
5. Min Coverage
6. Max Coverage
7. Clustering

Στα παρακάτω, ως V_M θα ορίζεται το σύνολο των μεταβλητών της μεθόδου, V_O ένα υποσύνολο των μεταβλητών της μεθόδου M που περιέχει μόνο τις μεταβλητές που εξάγονται από τη μέθοδο. Το SL_i αντιπροσωπεύει το slice που αντιστοιχεί στη μεταβλητή $u_i \in V_O$ και SL_{int} είναι η τομή όλων των SL_i, δηλαδή

$$\mathrm{SL}_{int} = \bigcap_{i=1}^{|V_O|} \mathrm{SL}_i$$

Coverage

Συγκρίνει το μήκος του slice με το μήκος της αρχικής μεθόδου. Το μήκος υπολογίζεται σε εντολές. Το coverage ορίζεται ως ο μέσος λόγος του μήκους του slice προς το μήκος της αρχικής μεθόδου. Δηλαδή

$$Coverage(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_i|}{\mathrm{length}(M)}$$

Overlap

Εκφράζει τη μοναδικότητα εμφάνισης των εντολών σε ένα μόνο slice και υπολογίζει πόσες εντολές αποτελούν αποκλειστικότητα του συγκεκριμένου slice, δηλαδή δεν εμφανίζονται και σε άλλα slices. Υπολογίζεται ως μέσος λόγος του συνόλου των εντολών που εμφανίζονται στην τομή των slices ως προς το μήκος του κάθε slice. Δηλαδή

$$Overlap(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_{int}|}{|SL_i|}$$

Parallelism

Εκφράζει τον αριθμό των slices που έχουν λίγες κοινές εντολές με τα υπόλοιπα slices. Υπολογίζεται ως το σύνολο των slices που έχουν ανά ζεύγη επικάλυψη με όλα τα υπόλοιπα slices μικρότερη από ένα κατώφλι τ, δηλαδή

$$Parallelism(M) = |\{SL_i \text{ such that } |SL_i \bigcap SL_j| \leq \tau \text{ for all } j \neq i\}|$$

Tightness

Υπολογίζει το σύνολο των εντολών που εμφανίζονται σε κάθε slice και εκφράζεται ως ο λόγος της τομής των slices προς το συνολικό μήκος της αρχικής μεθόδου. Ένας υψηλός λόγος αποτελεί ένδειξη ότι ενδεχομένως όλες οι εντολές θα έπρεπε να ομαδοποιηθούν μαζί λόγο των κοινών τους δεδομένων.

$$Tightness(M) = \frac{|SL_{int}|}{\text{length}(M)}$$

Min Coverage

Εκφράζει το λόγο του μήκους του μικρότερου slice προς το μήκος της μεθόδου. Δηλαδή

$$MinCoverage(M) = \frac{1}{\text{length}(M)} \min_i |SL_i|$$

Max Coverage

Εκφράζει το λόγο του μήκους του μεγαλύτερου slice προς το μήκος της μεθόδου. Δηλαδή

$$MaxCoverage(M) = \frac{1}{\text{length}(M)} \max_i |SL_i|$$

Clustering

Αφορά στα slices όπως αυτά ορίστηκαν αρχικά από τον Weiser [17], ως εκτελέσιμες μονάδες κώδικα, και εκφράζει το βαθμό στον οποίο τα προκύπτοντα slices αντανακλούν την αρχική δομή του προγράμματος. Μπορεί να υπολογισθεί ως ο

μέσος λόγος των αρχικά γειτονικών εντολών προς τις συνολικές εντολές σε κάθε slice. Μια χαμηλή τιμή αποτελεί ένδειξη ότι τα slices συνδέονται με τέτοιο τρόπο ώστε να δημιουργούν το φαινόμενο που εκφράζεται ως spaghetti code, ενώ αντίθετα μια υψηλή τιμή δηλώνει ότι τα slices αντανακλούν την αρχική δομή του κώδικα με την ομαδοποίηση των εντολών.

Κεφάλαιο 3

Πρότυπα Σχεδίασης και Θεωρία Προγραμματισμού

Κάθε πρότυπο περιγράφει ένα πρόβλημα το οποίο συναντάται επανειλημμένα σε ένα περιβάλλον. Εν συνεχεία, περιγράφει τον πυρήνα της λύσης του προβλήματος με τέτοιο τρόπο, ώστε η λύση να μπορεί να επαναχρησιμοποιηθεί χιλιάδες φορές, χωρίς απαραίτητα να εφαρμόζεται με τον ίδιο τρόπο. Παρότι ο παραπάνω ορισμός διατυπώθηκε για πρότυπα σχεδίασης σε κατασκευή κτιρίων και πόλεων, ο ορισμός εξακολουθεί να υφίσταται και όταν πρόκειται για πρότυπα σχεδίασης σε αντικειμενοστραφείς γλώσσες προγραμματισμού. [1]

Ένα πρότυπο σχεδίασης χαρακτηρίζεται από τα παρακάτω τέσσερα βασικά στοιχεία [2]:

1. Ένα *όνομα,* το οποίο μας επιτρέπει να το χρησιμοποιούμε τόσο για να περιγράψουμε εν τάχει τη λύση του, όσο και για να περιγράψουμε το πρόβλημα και τις συνέπειες του.

2. Το *πρόβλημα,* το οποίο μας υποδεικνύει πότε μπορεί να εφαρμοστεί το πρότυπο σχεδίασης. Περιγράφει το πρόβλημα που παρουσιάζεται και το γενικό πλαίσιό του. Στο πρόβλημα μπορεί να περιλαμβάνονται και οι συνθήκες οι οποίες πρέπει να απαντώνται προκειμένου να εφαρμοστεί το συγκεκριμένο πρότυπο.

3. Τη *λύση,* η οποία περιγράφει και χαρακτηρίζει τα στοιχεία που απαρτίζουν το πρότυπο. Η λύση δεν περιγράφει ένα συγκεκριμένο πλάνο ή υλοποίηση, διότι κατ' αυτό τον τρόπο αναιρείται η ιδιότητα του προτύπου να μπορεί να επαναχρησιμοποιηθεί σε πολλές περιπτώσεις. Αντίθετα το πρότυπο παρέχει μια αφηρημένη περιγραφή ενός σχεδιαστικού προβλήματος και μια γενική δομή με την οποία μπορεί να λυθεί.

4. Τις *συνέπειες,* που περιγράφουν τα αποτελέσματα και το trade-off από την εφαρμογή του προτύπου. Οι συνέπειες είναι σημαντικές προκειμένου να μπορέσουμε

να αξιολογήσουμε το πρότυπο και τυχόν εναλλακτικές και να κατανοήσουμε το κόστος και τα οφέλη από την εφαρμογή του προτύπου. Εφόσον αναφερόμαστε στις αντικειμενοστραφείς γλώσσες προγραμματισμού, σε αυτές περιλαμβάνονται και η περιγραφή των συνεπειών που έχει η εφαρμογή του προτύπου στην ευελιξία, την επεκτασιμότητα (extensibility) και τη μεταφερσιμότητα (portability).

Πέρα από τα τυποποιημένα πρότυπα, ο καθένας μπορεί να ορίσει τους δικούς του άτυπους κανόνες συγγραφής κώδικα. Για παράδειγμα, κατά τη συγγραφή ενός μικρού τμήματος κώδικα εγείρονται κάποια ζητήματα αμιγώς τεχνικά, όπως η αποδοτικότητα και η ευκολία στην ανάγνωση, τη συγγραφή, την επαλήθευση και την τροποποίησή του. Η σύνοψη των παραπάνω ζητημάτων αποτελεί τον προπομπό ενός προτύπου συγγραφής. Κάθε τέτοιο πρότυπο ορίζει μια κατευθυντήρια γραμμή ανάλογη με τις προτεραιότητες που δίδονται στα παραπάνω ζητήματα.

Γενικότερα, η πλειοψηφία των προγραμμάτων διαθέτει κάποια συγκεκριμένα χαρακτηριστικά. Η ανάγνωση του κώδικα είναι συχνότερη από τη συγγραφή του, ως αποτέλεσμα του περισσότερου χρόνου που δαπανάται στην τροποποίηση του λογισμικού, παρά στην αρχική του ανάπτυξη. Επίσης, τα περισσότερα προγράμματα δομούνται με βάση ένα κύριο σύνολο μοντέλων ροής καταστάσεων και ελέγχου. Τέλος, οι χρήστες πρέπει να έχουν την ευχέρεια να κατανοήσουν τόσο τις λεπτομέρειες ενός λογισμικού όσο και τη γενική του ιδέα. Όλα τα παραπάνω, οδήγησαν στην προτυποποίηση συγκεκριμένων κανόνων συγγραφής κώδικα.

Εν κατακλείδι, τα πρότυπα λειτουργούν άριστα ως καθοδηγητές στη λήψη αποφάσεων. Στη συνέχεια, αναλύονται οι αξίες και οι αρχές του προγραμματισμού, στη βελτίωση των οποίων αποσκοπούν τα πρότυπα καθώς και τα κίνητρα για τη χρήση τους. [25]

3.1 Αξίες Προγραμματισμού

Αξίες ονομάζονται τα καθολικά πρωταρχικά χαρακτηριστικά του προγραμματισμού. Αυτά είναι η επικοινωνία του κώδικα με τους προγραμματιστές, η

αποφυγή της μεγάλης πολυπλοκότητας και η διατήρηση της ευελιξίας του λογισμικού και επηρεάζουν κάθε απόφαση που καλείται να λάβει ο προγραμματιστής κατά τη διάρκεια της συγγραφής κώδικα. Οι αξίες αυτές κάποιες φορές μπορεί να συγκρούονται μεταξύ τους, συχνά όμως αλληλοσυμπληρώνονται.

3.1.1 Επικοινωνία

Η επικοινωνία σχετίζεται με την ικανότητα ενός λογισμικού να είναι κατανοητό, εύκολα τροποποιήσιμο και εύχρηστο. Ο κώδικας πρέπει να συγγράφεται με τη σκέψη ότι κάποιος άλλος θα τον διαβάσει κι όχι μόνο για να είναι κατανοητός στον υπολογιστή. Με αυτό τον τρόπο ο κώδικας γίνεται πιο ευανάγνωστος, έχει λιγότερο κόστος συντήρησης, διευκολύνεται η σκέψη του προγραμματιστή και ικανοποιείται σε κάποιο βαθμό η ανάγκη για κοινωνικοποίηση.

Σύμφωνα με τον Knuth [41], ένα πρόγραμμα θα πρέπει να διαβάζεται σαν βιβλίο. Πρέπει να έχει πλοκή, ρυθμό και ευχάριστες μικρές προτάσεις. Εάν αυτό δε συμβαίνει, τότε είναι ευκολότερο να ξαναγραφεί ο κώδικας παρά να γίνει κατανοητός. Το μεγαλύτερο κόστος στο χρόνο ζωής ενός λογισμικού παρατηρείται μετά τη συγγραφή του κώδικα. Επομένως, για να παραχθεί ένα φθηνό λογισμικό, θα πρέπει να είναι εύκολο να διαβαστεί.

Επίσης, όταν ο προγραμματιστής έχει στο μυαλό του τον τρόπο με τον οποίο θα αντιληφθεί τον κώδικα κάποιος άλλος αναγνώστης, τότε ξεφεύγει από την απομονωμένη οπτική του και οδηγείται σε πιο κατανοητή συγγραφή. Άλλωστε ένα λογισμικό δεν αφορά μόνο τον συντάκτη του και γι' αυτό το λόγο δε θα πρέπει να αγνοούνται τα στοιχεία της κοινωνικότητας.

3.1.2 Απλότητα

Ο Edward Tufte [42] αναφέρει πως αν από ένα γράφο αφαιρεθούν στοιχεία που δεν προσθέτουν πληροφορία, τότε το αποτέλεσμα είναι ένας γράφος πολύ πιο κατανοητός από τον αρχικό. Ο περιορισμός της πλεονάζουσας πολυπλοκότητας διευκολύνει όσους διαβάζουν, χρησιμοποιούν ή τροποποιούν ένα πρόγραμμα. Ενώ

μέρος της πολυπλοκότητας προκύπτει από τη φύση του προς επίλυση προβλήματος και είναι απαραίτητη, η υπόλοιπη πολυπλοκότητα προκύπτει κατά τη συγγραφή του κώδικα. Η τελευταία καλείται πλεονάζουσα και είναι αυτή που αφαιρεί αξία από το λογισμικό καθώς μειώνει την πιθανότητα να εκτελεστεί ορθά και το καθιστά λιγότερο ευέλικτο σε μελλοντικές αλλαγές.

Η απλότητα είναι συνδεδεμένη με το επίπεδο του προγραμματιστή. Επομένως κάτι που για έναν έμπειρο προγραμματιστή είναι απλό, για κάποιον άλλον μπορεί να είναι πολύπλοκο. Έτσι θα πρέπει να δίδεται ιδιαίτερη σημασία στην ευκολία κατανόησης του προγράμματος από άλλους προγραμματιστές που αναμένεται να το διαβάσουν, να το χρησιμοποιήσουν ή να το τροποποιήσουν.

Η αναζήτηση απλούστερων λύσεων ενδέχεται να κατευθύνει τον προγραμματιστή στην εύρεση νέων τρόπων σκέψης και υλοποίησης, οδηγώντας σε καινοτομικές λύσεις. Γενικότερα, η συγγραφή κώδικα θα πρέπει να γίνεται με τέτοιο τρόπο, ώστε οποιαδήποτε αφαίρεση τμήματός του να οδηγεί σε απώλεια πληροφορίας.

Από τα παραπάνω, γίνεται αντιληπτό ότι οι έννοιες επικοινωνία και απλότητα είναι αλληλοεξαρτώμενες.

3.1.3 Ευελιξία

Είναι πολύ συχνό μετά την αρχική ανάπτυξη του λογισμικού, αυτό να υπόκειται σε αλλαγές προκειμένου να αυξηθεί η λειτουργικότητά του, να προσαρμοστεί σε νέες απαιτήσεις ή να βελτιωθεί η αποδοτικότητά του. Η ευελιξία είναι το στοιχείο το οποίο ορίζει πόσο εύκολα μπορούν να επιτευχθούν τα παραπάνω. Ωστόσο η αναζήτησή της μπορεί να οδηγήσει στην αύξηση της πολυπλοκότητας του συστήματος.

Από την άλλη όμως, η απλότητα και η επικοινωνία έχουν σαν αποτέλεσμα την αύξηση της ευελιξίας καθώς όσο πιο γρήγορα μπορεί να αναγνωσθεί, να κατανοηθεί και να τροποποιηθεί το λογισμικό, τόσο πιο πολλές επιλογές δίνονται στους οργανισμούς για μελλοντικές αλλαγές.

Επομένως, είναι προτιμητέα η χρήση προτύπων που εξασφαλίζουν την ευελιξία μέσω της απλότητας και τις επικοινωνίας καθώς έτσι ικανοποιούνται περισσότερο οι αξίες του προγραμματισμού.

3.2 Αρχές Προγραμματισμού

Οι αρχές είναι επίσης γενικές ιδέες, αλλά περισσότερο επικεντρωμένες στον προγραμματισμό απ' ότι οι αξίες. [25] Η μελέτη των αρχών είναι ικανή να οδηγήσει σε εύρεση νέων προτύπων. Πέραν αυτού, ο γενικός τους χαρακτήρας παρέχει τη δυνατότητα ευκολότερης κατανόησης των κινήτρων για τη χρήση ενός προτύπου και ταυτόχρονα διευκολύνει τη σύγκριση των προτύπων. Ένα πολύ βασικό χαρακτηριστικό των αρχών είναι, επίσης, ότι μπορούν να προσφέρουν λύσεις σε νέες κατηγορίες προβλημάτων, λόγω της γενικότερης φύσης τους συγκριτικά με τα πρότυπα. Συνοψίζοντας, η κατανόηση των αρχών οδηγεί στην ταχύτερη εκμάθηση και την ακέραιη αντιμετώπιση νέων καταστάσεων.

3.2.1 Τοπικότητα Επιπτώσεων

Οι τροποποιήσεις στον κώδικα θα πρέπει να επηρεάζουν μόνο τα σημεία στα οποία πραγματοποιούνται. Σε αντίθετη περίπτωση, το κόστος για μία τροποποίηση αυξάνεται δραματικά, καθώς δεν υφίσταται η αξία της επικοινωνίας.

Η ανάγκη για τοπικότητα των επιπτώσεων βρίσκεται πίσω από τη δημιουργία πολλών προτύπων σχεδίασης.

3.2.2 Ελαχιστοποίηση Επανάληψης

Η επανάληψη ενός τμήματος κώδικα αυξάνει το κόστος μιας τροποποίησης, διότι μία αλλαγή σε ένα σημείο του κώδικα είναι πιθανό να επαναληφθεί σε κάποιο άλλο.

Εκτός από το κόστος των αλλαγών, ο προγραμματιστής θα πρέπει να αναζητήσει όλα τα παρόμοια σημεία του κώδικα και να αποφανθεί αν είναι αναγκαία η τροποποίηση σε αυτά. Εύκολα κανείς διαπιστώνει τη συσχέτιση που υπάρχει μεταξύ της επανάληψης και της τοπικότητας των επιπτώσεων.

Όμως η επανάληψη δε σχετίζεται μόνο με τον κώδικα, αλλά και με την εννοιολογική θεμελίωσή του. Μια επανάληψη στην ιεραρχία των τάξεων θα επέφερε πολλαπλές αλλαγές σε ενδεχόμενη εννοιολογική τροποποίηση.

Η επανάληψη δε γίνεται πάντα εύκολα αντιληπτή. Ένας τρόπος για να αποφευχθεί, είναι η δημιουργία μικρών τμημάτων κώδικα που εξυπηρετούν έναν συγκεκριμένο σκοπό. Με αυτό τον τρόπο επιτυγχάνεται και η βελτίωση της επικοινωνίας του κώδικα.

3.2.3 Συνύπαρξη Λογικής και Δεδομένων

Οι συνέπειες των τροποποιήσεων παραμένουν τοπικές, εάν συνυπάρχουν λογική και δεδομένα. Έτσι, μία τροποποίηση επιφέρει αλλαγές ταυτόχρονα σε δεδομένα και λογική. Όπως συμβαίνει και με την επανάληψη, η μη συνύπαρξη λογικής και δεδομένων δε γίνεται εύκολα αντιληπτή, ενώ η επίτευξη της συνύπαρξης βελτιώνει την επικοινωνία.

3.2.4 Συμμετρία

Η συμμετρία συναντάται συχνά σε προγράμματα και βοηθάει στην ευκολότερη ανάγνωση του κώδικα, αφού αν κατανοηθεί το ένα μέρος της συμμετρίας, το υπόλοιπο κατανοείται γρηγορότερα. Ορίζεται κυρίως σε εννοιολογικό επίπεδο. Ένα παράδειγμα συμμετρίας θα μπορούσε να είναι η κλήση του ζεύγους συναρτήσεων input() και output().

Μπορεί να αποτελέσει πρωταρχικό βήμα για την ανίχνευση επανάληψης. Αν μια παρόμοια σκέψη υπάρχει σε πολλά σημεία στον κώδικα, τότε η δημιουργία συμμετρίας μεταξύ των σημείων είναι ένα καλό πρώτο βήμα για την ενοποίησή τους.

3.2.5 Δηλωτική Έκφραση

Είναι σημαντικό διαβάζοντας κανείς τον κώδικα, να μπορεί εύκολα να καταλάβει τις λειτουργίες που επιτελούνται. Για ένα πολύπλοκο πρόγραμμα, η κατανόηση του κώδικα προκύπτει από τη μελέτη της κατάστασης του προγράμματος και την αποτύπωση των ροών ελέγχου και δεδομένων. Για ένα πιο απλό, αρκεί μία ανάγνωση. Όσο πιο σαφείς γίνονται οι προθέσεις του προγράμματος μέσω του τρόπου έκφρασης του κώδικα, τόσο πιο εύκολα μπορεί να αναγνωσθεί.

3.2.6 Ρυθμός Αλλαγών

Ο ρυθμός με τον οποίο πραγματοποιούνται τροποποιήσεις στη λογική ή τα δεδομένα του κώδικα μπορεί να αποτελέσει ένδειξη για την ομαδοποίηση ή το διαχωρισμό τους. Πρόκειται για ένα είδος χρονικής συμμετρίας. Αν ο ρυθμός των αλλαγών είναι διαφορετικός, τότε ο διαχωρισμός τους θα ενισχύσει την αρχή της τοπικότητας και θα περιορίσει την πιθανότητα σφάλματος από τη μεριά του προγραμματιστή που επεξεργάζεται το λογισμικό.

Όσον αφορά τα δεδομένα, τα πεδία ενός αντικειμένου θα πρέπει να αλλάζουν με τον ίδιο ρυθμό. Αν αυτό δε συμβαίνει, τότε θα πρέπει κάποια από αυτά να γίνουν τοπικές μεταβλητές. Από την άλλη, αν δύο πεδία αλλάζουν με τον ίδιο ρυθμό, σε αντίθεση με τα υπόλοιπα, τότε ίσως ήταν χρήσιμο να ομαδοποιηθούν σε μία βοηθητική τάξη.

3.3 Κίνητρα

Έχοντας διαβάσει κανείς τα παραπάνω, εύκολα καταλήγει στο συμπέρασμα ότι όλα γίνονται για τη μείωση του κόστους. Ήδη από το στάδιο της σχεδίασης, γίνονται προσπάθειες ώστε το λογισμικό να έχει ελαχιστοποιημένο συνολικό κόστος. Ως συνολικό κόστος ορίζεται το άθροισμα του κόστους ανάπτυξης και του κόστους συντήρησης. Το τελευταίο είναι κατά πολύ μεγαλύτερο από το πρώτο, πράγμα που δε γίνεται πάντα αντιληπτό από νέους προγραμματιστές.

Η συντήρηση είναι μια σύνθετη διαδικασία. Περιλαμβάνει την κατανόηση του κώδικα, την τροποποίησή του, τον έλεγχο και την υιοθέτηση της αλλαγής. Γι' αυτό το λόγο απαιτεί πολύ χρόνο, ο οποίος αυξάνεται ακόμα περισσότερο όταν διαπιστώνονται σφάλματα.

Μία στρατηγική για τη μείωση του κόστους είναι η υψηλή επένδυση στην αρχική υλοποίηση έτσι ώστε να ελαχιστοποιηθεί ή να εκμηδενιστεί η ανάγκη για συντήρηση. Η στρατηγική αυτή γενικά αποτυγχάνει να μειώσει το συνολικό κόστος, διότι οι ανακύπτουσες ανάγκες δεν μπορούν πάντα να προβλεφθούν.

Γενικότερα, η στρατηγική της υψηλής αρχικής επένδυσης υστερεί σε δύο σημεία. Το πρώτο είναι η αξία του χρόνου. Πολύς χρόνος σημαίνει πολλά χρήματα και το χρήμα κοστίζει περισσότερο σήμερα απ' όσο θα κοστίζει στο μέλλον. Το δεύτερο σημείο είναι η αβεβαιότητα του μέλλοντος. Τα πρότυπα στοχεύουν στην άμεση αποκόμιση οφέλους, αλλά ταυτόχρονα προσφέρουν έναν καλογραμμένο κώδικα ώστε να διευκολυνθούν μελλοντικές τροποποιήσεις ή επεκτάσεις.

Μια πιο απλή στρατηγική είναι ο κώδικας να συγγράφεται με τέτοιον τρόπο σαν να επικοινωνούν δύο άνθρωποι. Έτσι ο κώδικας γίνεται πιο σαφής, με λιγότερα σφάλματα, διαμοιράζεται ευκολότερα και επεκτείνεται ομαλότερα.

Κεφάλαιο 4

Αναδομήσεις

Αναδόμηση κώδικα ορίζεται η διαδικασία της μετατροπής ενός συστήματος λογισμικού με τέτοιο τρόπο ώστε να μην τροποποιείται η συμπεριφορά του, ενώ ταυτόχρονα βελτιώνεται η δομή του. Αποτελεί έναν πειθαρχημένο τρόπο βελτίωσης του κώδικα και κατ' επέκταση της σχεδίασής του με σκοπό την ελαχιστοποίηση της πιθανότητας εμφάνισης σφαλμάτων. Εφαρμόζεται μετά τη συγγραφή του κώδικα, ενώ για συγκεκριμένες περιπτώσεις προβληματικού σχεδιασμού ή συγγραφής, υπάρχουν τυποποιημένες προτάσεις αναδόμησης. [23]

4.1 Οφέλη

Η αναδόμηση δεν είναι πανάκεια, ωστόσο είναι μία διαδικασία που αν χρησιμοποιηθεί σωστά μπορεί να οδηγήσει στην αποκόμιση ποικίλων οφελών. Στη συνέχεια παρατίθενται κάποια από αυτά. [23]

4.1.1 Βελτίωση Σχεδίασης Λογισμικού

Χωρίς τη χρήση αναδομήσεων η σχεδίαση του λογισμικού φθείρεται. Καθώς γίνονται αλλαγές στον κώδικα η δομή του μεταβάλλεται. Σε βάθος χρόνου ο κώδικας γίνεται δυσνόητος και περισσότερο περίπλοκος. Σε αυτή την περίπτωση, η αναδόμηση λειτουργεί ως ένα εργαλείο βελτίωσης της δομής και πρέπει να εφαρμόζεται τακτικά.

Ένας ακόμα τρόπος στον οποίο η αναδόμηση εξυπηρετεί τη σχεδίαση του λογισμικού είναι η εξάλειψη διπλότυπου κώδικα. Με τον περιορισμό του διευκολύνεται η διαδικασία τροποποίησης του κώδικα.

4.1.2 Ευκολία Κατανόησης

Οι προγραμματιστές προσπαθούν να γράψουν κώδικα με τρόπο τέτοιο ώστε να τον αντιλαμβάνεται ο υπολογιστής. Πολλές φορές αγνοείται το γεγονός ότι κάποιος άλλος πέραν του αρχικού προγραμματιστή θα κληθεί να τον διαβάσει και να τον κατανοήσει. Επίσης, υπάρχει η αντίληψη ότι εφόσον ο αρχικός προγραμματιστής είναι αυτός που συντάσσει τον κώδικα, θα μπορεί οποιαδήποτε στιγμή ανατρέξει σε αυτόν να κατανοήσει για ποιο λόγο έχει γράψει κάτι και τι λειτουργία αυτό επιτελεί. Αυτό έχει σαν αποτέλεσμα ο κώδικας να μην έχει μια εύληπτη δομή.

Ο μετασχηματισμός του σε μορφή φιλικότερη προς τρίτους μπορεί να οδηγήσει σε μία δομή μη ιδανική. Μειώνει, όμως, σημαντικά το χρόνο που απαιτείται για την κατανόηση και την τροποποίησή του. Με τη χρήση αναδομήσεων, ο κώδικας οργανώνεται, γίνεται πιο σαφής και η σχεδίαση του λογισμικού γίνεται ευκολότερα αντιληπτή.

4.1.3 Αποσφαλμάτωση

Η κατανόηση του κώδικα διευκολύνει τον εντοπισμό των σφαλμάτων που περιέχονται σε αυτόν. Κατά τη διαδικασία της αναδόμησης, η λειτουργία του κώδικα γίνεται αντιληπτή και με την αποσαφήνιση της δομής του αναδεικνύονται τα σφάλματα.

4.1.4 Ταχύτητα Προγραμματισμού

Οι αναδομήσεις οδηγούν στην αύξηση της ταχύτητας προγραμματισμού. Αυτό μπορεί να φαίνεται οξύμωρο καθώς απαιτείται χρόνος για την εφαρμογή τους στον ήδη υπάρχοντα κώδικα. Η αποφυγή τους μπορεί να οδηγεί βραχυπρόθεσμα σε γρηγορότερο προγραμματισμό, όμως μακροπρόθεσμα η συγγραφή κώδικα

επιβραδύνεται. Αυτό συμβαίνει γιατί καθώς προστίθεται κώδικας, αναλώνεται περισσότερο χρόνος στην ανεύρεση και διόρθωση λαθών, παρά στην προσθήκη λειτουργικότητας στο σύστημα.

4.2 Πότε Εφαρμόζονται

Ένα ζήτημα που σχετίζεται με τις αναδομήσεις είναι το πότε θα πρέπει να εφαρμοστούν. Οι αναδομήσεις δεν αποτελούν μια διαδικασία με συγκεκριμένο χρονικό πλαίσιο στο οποίο θα υλοποιηθούν. Πραγματοποιούνται συνεχώς οπότε προκύψει η ανάγκη. Οι αναδομήσεις εξυπηρετούν την εφαρμογή άλλων προγραμματιστικών σκοπών και δεν αποτελούν οι ίδιες προγραμματιστικό σκοπό.

Η γενική ιδέα είναι ότι κατά την αρχική συγγραφή του κώδικα απλά γίνεται η υλοποίηση. Στην πρώτη τροποποίηση γίνεται η προσπάθεια αποφυγής διπλότυπου κώδικα, χωρίς απαραίτητα αυτό να επιτυγχάνεται και στην δεύτερη τροποποίηση εφαρμόζεται αναδόμηση.

Η συχνότερη περίπτωση εφαρμογής της αναδόμησης είναι όταν προστίθεται κάποια μέθοδος στον κώδικα. Η αναδόμηση σε αυτή την περίπτωση εξυπηρετεί δύο σκοπούς. Κατά πρώτον, η εφαρμογή της βοηθάει στην κατανόηση του κώδικα στον οποίο θα γίνει η προσθήκη. Κατά δεύτερον, η σχεδίαση του αρχικού κώδικα ενδέχεται να μη διευκολύνει την προσθήκη της μεθόδου. Εφαρμόζοντας όμως κάποια αναδόμηση, η προκύπτουσα σχεδίαση μπορεί να εξυπηρετεί το σκοπό αυτό. Αυτό έχει σαν αποτέλεσμα την επιτάχυνση της διαδικασίας επέκτασης του κώδικα.

Η ανάγκη διόρθωσης των σφαλμάτων αποτελεί μια ακόμα ένδειξη για την εφαρμογή αναδόμησης. Το σκεπτικό είναι ότι αν η δομή του κώδικα δεν είναι αρκετά κατανοητή ώστε να εντοπιστεί το σφάλμα εύκολα, τότε θα πρέπει να βελτιωθεί.

Μια άλλη περίσταση για να εφαρμοστεί αναδόμηση είναι κατά τη διάρκεια της επανεξέτασης του κώδικα. Η επανεξέταση πραγματοποιείται για τη μετάδοση της γνώσης του που περιέχεται στον κώδικα, σε μια ομάδα ανάπτυξης. Οι εμπειρότεροι προγραμματιστές μεταδίδουν τη γνώση σε λιγότερο έμπειρους, δίνοντάς τους τη

δυνατότητα να κατανοήσουν περισσότερες πτυχές από ένα μεγάλο σύστημα. Επίσης, οι προγραμματιστές ωθούνται στη συγγραφή σαφέστερου κώδικα, μέσω της ανατροφοδότησης που λαμβάνουν από την ομάδα ανάπτυξης κατά τη διαδικασία της επανεξέτασης. Τέλος, μέσα από αυτή τη διαδικασία δίδεται η δυνατότητα πρότασης βελτιώσεων και ιδεών από μεγαλύτερο πλήθος ατόμων.

4.3 Πότε Δεν Εφαρμόζονται

Υπάρχουν περιπτώσεις όπου η χρήση αναδομήσεων δεν ενδείκνυται. Η προφανέστερη εξ αυτών είναι όταν ο αρχικός κώδικας έχει συνταχθεί με τέτοιο τρόπο ώστε, παρότι θα μπορούσε να χρησιμοποιηθεί αναδόμηση, το να γραφεί εξαρχής είναι ευκολότερο. Όταν ο υπάρχων κώδικας, για παράδειγμα, δε λειτουργεί ορθά, τότε η χρήση αναδομήσεων δεν προσφέρει ιδιαίτερο όφελος. Οι αναδομήσεις προτιμώνται στην περίπτωση που ο υπάρχων κώδικας επιτελεί σε μεγάλο βαθμό τις λειτουργίες που έχουν προκαθοριστεί.

Στην περίπτωση που δεν είναι επιθυμητό να γραφεί ολόκληρο το λογισμικό από την αρχή, τότε μπορεί να αναδομηθεί ένα μεγάλο ποσοστό του σε μικρότερες μονάδες με ισχυρή ενθυλάκωση. Εν συνεχεία μπορεί, τοπικά πλέον, να αποφασιστεί αν η κάθε μονάδα χρίζει περαιτέρω αναδόμησης ή είναι προτιμότερο να γραφεί εκ νέου.

Μια ακόμα περίπτωση στην οποία δεν ενδείκνυται η χρήση αναδομήσεων είναι όταν πλησιάζει η διορία παράδοσης του λογισμικού. Σε αυτή την περίπτωση τα οφέλη που αποκομίζονται από τις αναδομήσεις ενδέχεται να αναδειχθούν μετά το πέρας της προθεσμίας. Ωστόσο, εκτός από την περίπτωση που η λήξη της διορίας είναι κοντά, η αναδόμηση δεν θα πρέπει να αποφεύγεται λόγω ανεπάρκειας χρόνου. Η εμπειρία έχει αποδείξει ότι η χρήση αναδομήσεων οδηγεί σε αυξημένη παραγωγικότητα. Ακόμα, η ανεπάρκεια χρόνου είναι ένδειξη ότι θα έπρεπε να χρησιμοποιηθεί αναδόμηση.

4.4 Προβλήματα

Παρότι οι αναδομήσεις προσφέρουν πολλά οφέλη στη συγγραφή του λογισμικού, δεν αποτελούν πανάκεια. Υπάρχουν περιπτώσεις που δεν μπορούν να εφαρμοστούν και η χρήση τους μπορεί όχι μόνο να μην αποδίδει, αλλά να προκαλεί και προβλήματα. Στη συνέχεια αναφέρονται κάποιες τέτοιες περιπτώσεις. [23] Η λίστα δεν είναι ολοκληρωμένη καθώς δεν είναι γνωστοί όλοι οι περιορισμοί που μπορεί να προκύψουν από τη χρήση των αναδομήσεων. Η υιοθέτηση και η εφαρμογή κάθε προτύπου από μεγάλη μερίδα προγραμματιστών αναδεικνύει και άλλες τέτοιες περιπτώσεις.

4.4.1 Βάσεις Δεδομένων

Οι βάσεις δεδομένων ανήκουν στις περιπτώσεις στις οποίες οι αναδομήσεις δεν μπορούν να εφαρμοστούν ή δεν ωφελούν. Οι περισσότερες επιχειρησιακές εφαρμογές εξαρτώνται άμεσα από το σχήμα της βάσης που χρησιμοποιούν. Για αυτόν το λόγο είναι δύσκολο να γίνει εφαρμογή των αναδομήσεων.

Ένας ακόμα αποτρεπτικός παράγοντας είναι η μετάπτωση (migration). Ακόμα και αν η βάση έχει δομηθεί με τέτοιο τρόπο ώστε να ελαχιστοποιείται η εξάρτηση του μοντέλου αντικειμένου από το σχήμα της βάσης, η αλλαγή του σχήματός της απαιτεί να γίνει επίσης μετάπτωση των δεδομένων, μια διαδικασία χρονοβόρα και επίπονη.

Στις μη-αντικειμενοστραφείς βάσεις δεδομένων οι παραπάνω περιορισμοί μπορούν να καμφθούν με τη χρήση ενός ενδιάμεσου λογισμικού μεταξύ της βάσης και του μοντέλου αντικειμένου. Αν και αυτό ενδέχεται να προσθέσει επιπλέον πολυπλοκότητα στο σύστημα, προσφέρει ευελιξία. Σε περίπτωση τροποποίησης είτε της βάσης είτε του μοντέλου, μόνο το ενδιάμεσο επίπεδο θα πρέπει να ενημερωθεί και δεν απαιτείται να αλλάξει όλη η δομή του συστήματος.

Οι αντικειμενοστραφείς βάσεις δεδομένων ενδέχεται να διευκολύνουν ή να περιπλέξουν την κατάσταση. Κάποιες εξ αυτών προσφέρουν τη δυνατότητα αυτόματης μετάπτωσης των δεδομένων, χωρίς ωστόσο να αποφεύγεται η

καθυστέρηση που προκύπτει για την ολοκλήρωση της διαδικασίας. Στην περίπτωση που η διαδικασία δεν είναι αυτοματοποιημένη, ο προγραμματιστής οφείλει να είναι προσεκτικός με το χειρισμό της μετάπτωσης.

4.4.2 Τροποποίηση Διεπαφών

Ένα από τα σπουδαιότερα οφέλη των αντικειμενοστραφών γλωσσών προγραμματισμού είναι η δυνατότητα τροποποίησης μονάδων λογισμικού διατηρώντας ανέπαφες τις διεπαφές. Ωστόσο η τροποποίηση μιας διεπαφής μπορεί να έχει απρόβλεπτες συνέπειες στη λειτουργία ενός συστήματος.

Ένα από τα προβλήματα των αναδομήσεων είναι ότι πολλές από αυτές προκαλούν αλλαγές στις διεπαφές. Μια μετονομασία μεθόδου μπορεί να οδηγήσει στην αλλαγή τις διεπαφής. Αν είναι γνωστές όλες οι αναφορές που γίνονται σε αυτή τη μέθοδο της διεπαφής, τότε δεν υπάρχει κάποιο πρόβλημα αφού αυτές μπορούν να εντοπιστούν και να επεξεργαστούν. Ωστόσο το ίδιο δεν είναι δυνατόν όταν είναι άγνωστος ή δεν είναι διαθέσιμος. Σε αυτή την περίπτωση θα πρέπει να διατηρείται και η αρχική και η τροποποιημένη διεπαφή, τουλάχιστον μέχρι οι χρήστες να είναι σε θέση να υιοθετήσουν τη μεταγενέστερη έκδοση. Η διατήρηση και των δύο επιτυγχάνεται τροποποιώντας το σώμα της πρώτης έτσι ώστε να καλεί την πιο πρόσφατη.

Παρότι η διατήρηση των διεπαφών είναι εφικτή, εξακολουθεί να αποτελεί πρόβλημα. Οι πλεονάζουσες μέθοδοι πρέπει να κατασκευαστούν και να διατηρηθούν για κάποιο χρονικό διάστημα. Αυξάνουν την πολυπλοκότητα της διεπαφής με αποτέλεσμα να γίνεται δύσχρηστη. Σε αυτή την περίπτωση ίσως είναι προτιμότερο να μη δημοσιοποιηθεί η διεπαφή αν δεν είναι απαραίτητο.

4.4.3 Τροποποίηση Σχεδίασης

Ένα ερώτημα που παραμένει ανοικτό είναι αν μπορούν να παρακαμφθούν σχεδιαστικά σφάλματα με την εφαρμογή αναδομήσεων. Σε κάποιες περιπτώσεις μία αναδόμηση εφαρμόζεται εύκολα και σε άλλες όχι. Στις περιπτώσεις που δεν είναι

προφανής ο τρόπος με τον οποίο θα μπορούσε να εφαρμοστεί μία αναδόμηση θα πρέπει να δοθεί περισσότερη προσοχή στο σχεδιασμό. Άλλο ανοικτό ζήτημα είναι αν η σχεδίαση είναι τόσο κεντρική που καθιστά δύσκολη την αναίρεση της όποιας αναδόμησης έχει εφαρμοστεί.

4.5 Αναδομήσεις και Σχεδίαση

Οι αναδομήσεις έχουν έναν ειδικό ρόλο ως συμπλήρωμα της σχεδίασης. Κατά τον προγραμματισμό θα πρέπει να λαμβάνεται υπόψη ο σχεδιασμός του λογισμικού. Η σχεδίαση του λογισμικού μπορεί να παρομοιαστεί με το μηχανικό σχέδιο και ο προγραμματισμός με την κατασκευή της οικοδομής. Η διαφορά έγκειται στο ότι το λογισμικό είναι περισσότερο ευέλικτο και απαιτεί περισσότερη σκέψη. [23]

Ένα επιχείρημα υπέρ της χρήσης των αναδομήσεων είναι ότι αυτές μπορούν να αποτελέσουν εναλλακτική της σχεδίασης. Με αυτή τη λογική δεν απαιτείται να προηγηθεί η φάση της σχεδίασης. Παρά το ότι η χρήση αναδομήσεων χωρίς πρότερη σχεδίαση είναι αποτελεσματική, δεν είναι ο ιδανικότερος τρόπος εργασίας. Είναι προτιμότερο να χρησιμοποιηθεί αναδόμηση μόνο αφού έχει παραχθεί μια αρχική άποψη για τον τρόπο με τον οποίο θα δομηθεί ο κώδικας.

Ένας ακόμα λόγος υπέρ της χρήση των αναδομήσεων είναι ότι μεταβάλλουν την αρχική σχεδίαση. Αν δε χρησιμοποιηθεί αναδόμηση τότε η απαίτηση να γίνει σωστή σχεδίαση είναι επιτακτική και αυτό γιατί οποιαδήποτε αλλαγή γίνει στη συνέχεια θα έχει μεγάλο κόστος. Με αυτό τον τρόπο αναλώνεται περισσότερος χρόνος στη σχεδίαση για να αποφευχθούν τυχόν μελλοντικές μεταβολές της.

Με τη χρήση αναδομήσεων η έμφαση δε δίδεται στον αρχικό σχεδιασμό καθώς οι αλλαγές σε μετέπειτα στάδια γίνονται εύκολα. Έτσι, η σχεδίαση γίνεται απλούστερη χωρίς να θυσιάζεται η ευελιξία του συστήματος.

4.6 Αναδομήσεις και Απόδοση

Ερωτήματα προκύπτουν σχετικά με την επίδραση των αναδομήσεων στην απόδοση του λογισμικού. Η απόδοση δεν μπορεί πάντα να θυσιαστεί στο βωμό της ευκολίας κατανόησης του κώδικα. Οι αναδομήσεις ενδέχεται να μειώνουν την ταχύτητα του λογισμικού, ωστόσο ενισχύουν την ευελιξία ρύθμισης της αποδοτικότητας. [23]

Η βασικότερη προσέγγιση για τη συγγραφή αποδοτικού λογισμικού είναι ο χρονικός προϋπολογισμός (time budgeting). Χρησιμοποιείται σε δύσκολα συστήματα πραγματικού χρόνου. Η υλοποίησή της γίνεται με τον καθορισμό των πόρων που επιτρέπεται να χρησιμοποιήσει κάθε μονάδα λογισμικού κατά τη διαδικασία αποσύνθεσης της σχεδίασης. Η κάθε μονάδα οφείλει να μην ξεπεράσει τον προϋπολογισμό της, αλλά αυτό μπορεί να υπερκεραστεί με την ανταλλαγή πόρων μεταξύ των μονάδων. Αυτή η προσέγγιση, ωστόσο, είναι υπερβολική για άλλου είδους συστήματα, όπως εταιρικά πληροφοριακά συστήματα.

Μια δεύτερη προσέγγιση είναι αυτή της διαρκούς παρακολούθησης. Σύμφωνα με αυτή, ο προγραμματιστής κάνει ό,τι περνάει από το χέρι του για να διατηρήσει υψηλή την απόδοση. Όμως οι αλλαγές για τη βελτίωση της απόδοσης καθιστούν το λογισμικό δυσνόητο και καθυστερούν την ανάπτυξή του. Επιπλέον, η προσπάθεια βελτίωσης της απόδοσης συγκεκριμένων σημείων του κώδικα μπορεί να μην είναι προσοδοφόρα. Η ανάλυση πολλών προγραμμάτων αποδεικνύει ότι μεγάλο μέρος του χρόνου εκτέλεσής τους αναλώνεται σε μικρότερα τμήματα κώδικα. Με την ισομερή βελτιστοποίηση του κώδικα, το 90% του χρόνου αναλώνεται χωρίς ιδιαίτερο όφελος. Η τρίτη προσέγγιση εκμεταλλεύεται την παραπάνω γνώση.

Σύμφωνα με την τελευταία προσέγγιση, το λογισμικό αρχικά αναπτύσσεται με έναν καλά δομημένο τρόπο, χωρίς να λαμβάνεται υπόψιν η αποδοτικότητα. Η απόδοση επιχειρείται να βελτιωθεί όταν η ανάπτυξη έχει ήδη ωριμάσει. Με τη χρήση ενός ελεγκτή γίνεται εντοπισμός των σημείων του κώδικα που εκτελούνται συχνότερα. Στη συνέχεια γίνεται προσπάθεια βελτίωσης της απόδοσής τους, όπως θα γινόταν και με την προσέγγιση της διαρκούς παρακολούθησης.

Η καλή δομή του λογισμικού εξυπηρετεί την παραπάνω διαδικασία με δύο τρόπους. Πρώτον, δίδεται χρόνος για τη ρύθμιση της αποδοτικότητάς του. Χάρη στη δομή του κώδικα η προσθήκη μεθόδων γίνεται εύκολα και γρήγορα και εξοικονομείται χρόνος για τη βελτιστοποίηση της αποδοτικότητας. Δεύτερον, ο ελεγκτής είναι σε θέση να αναδείξει μικρότερα και πιο κατανοητά κομμάτια κώδικα στα οποία αναλώνεται ο περισσότερος χρόνος εκτέλεσης του λογισμικού. Η βελτίωσή τους μπορεί να γίνει ευκολότερα και αποδοτικότερα. Οι αναδομήσεις μπορεί αρχικά να καθυστερούν την ανάπτυξη του λογισμικού, στη συνέχεια όμως και κατά τη φάση της βελτιστοποίησης, επιταχύνουν τη διαδικασία.

Κεφάλαιο 5

Αναδομήσεις στην Πράξη

Σκοπός των αναδομήσεων είναι ο κώδικας που προκύπτει μετά την εφαρμογή τους να είναι πιο απλουστευμένος, ορθότερα κατανεμημένος και να απεικονίζεται πιο κατανοητά. Αυτό έχει σαν αποτέλεσμα την ευκολότερη συντήρηση, επέκταση και επαναχρησιμοποίηση κώδικα. Τα παραπάνω οδηγούν σε ποιοτικότερο λογισμικό.

Με την ανάγνωση του κώδικα διαπιστώνεται ότι υπάρχουν δομές για τις οποίες η εφαρμογή αναδομήσεων είναι επιθυμητή ως και επιτακτική. Τέτοιες δομές χαρακτηρίζονται ως κακές οσμές (bad smells). [23]

Στα πλαίσια της παρούσας διπλωματικής εργασίας θα σταθούμε στην αναδόμηση μεγάλων και σύνθετων μεθόδων με σκοπό τη βελτίωση της συνεκτικότητας του κώδικα. Αρωγοί στη διαδικασία αυτή θα σταθούν οι κακές οσμές και οι αναδομήσεις για την αντιμετώπισή τους, που αναλύονται στη συνέχεια.

5.1 Μακροσκελής Μέθοδος (Long Method)

Ο κύκλος ζωής του αντικειμενοστραφούς λογισμικού αυξάνεται και οι συνθήκες συντήρησής του βελτιώνονται όταν αποτελείται από σύντομες μεθόδους. Αυτό συμβαίνει γιατί με τη χρήση σύντομων μεθόδων αποκομίζονται οφέλη όπως τα ακόλουθα.

Δυνατότητα επαναχρησιμοποίησης κώδικα

Με αυτό τον τρόπο μια σύντομη μέθοδος μπορεί να κληθεί σε πολλά σημεία μέσα στον κώδικα. Επίσης, μπορεί να τοποθετηθεί σε μία υπερτάξη έτσι ώστε να είναι προσβάσιμη από όλες τις υποτάξεις της.

Διαχωρισμός σκοπού και υλοποίησης

Η ομαδοποίηση μικρών τμημάτων κώδικα και η οργάνωσή τους σε σύντομες μεθόδους δίνει την ευκαιρία στον προγραμματιστή να περιγράψει τη λειτουργία που επιτελείται μέσα από αυτές. Έτσι ο σκοπός της μεθόδου διαχωρίζεται από την υλοποίησή της καθώς ο προγραμματιστής δε χρειάζεται να διαβάσει το σώμα της μεθόδου, παρά μόνο το όνομά της.

Απομόνωση αλλαγών

Σε περίπτωση που χρειαστεί να πραγματοποιηθεί μια αλλαγή, αλλά ο προγραμματιστής δε θέλει η αλλαγή να επηρεάσει πολλά σημεία του κώδικα, μπορεί να δημιουργήσει μια υποτάξη η οποία θα περιέχει μέθοδο που επαναορίζει την αρχική.

Κωδικοποίηση υποθετικής λογικής

Η αντικατάσταση της υποθετικής λογικής με μηνύματα μπορεί να αποφέρει ταυτόχρονη μείωση των διπλοτύπων και αύξηση σαφήνειας και ευελιξίας. [23]

Πότε όμως μια μέθοδος θεωρείται μακροσκελής; Στη διεθνή βιβλιογραφία εντοπίζεται πληθώρα απόψεων. Το ιδανικό μέγεθος μιας μεθόδου προσδιορίζεται άλλοτε με χρήση ποσοτικών μεθόδων και άλλοτε με εμπειρική προσέγγιση.

Η πρώτη προσπάθεια να τεθεί ένα όριο στο μέγεθος των μεθόδων έγινε από την IBM η οποία τις περιόρισε σε το πολύ 50 γραμμές και την TRW η οποία έθεσε ως άνω όριο τις 2 σελίδες. [26] Η μελέτη των Basili και Periccone έδειξε ότι το μέγεθος μιας μεθόδου (έως 200 γραμμές) είναι αντιστρόφως ανάλογο με τον αριθμό των σφαλμάτων ανά γραμμή κώδικα [27], ενώ άλλη μελέτη έδειξε ότι με το πλήθος των σφαλμάτων δε σχετίζεται το μέγεθος της μεθόδου, αλλά η πολυπλοκότητά της και τα δεδομένα που επεξεργάζεται. [28] Νεότερες μελέτες έδειξαν ότι οι σύντομες μέθοδοι (32 ή λιγότερες γραμμές) δεν συνεπάγονταν λιγότερο κόστος ή χαμηλότερη αναλογία σφαλμάτων και γραμμών, ενώ οι μεγαλύτερες μέθοδοι (65 ή περισσότερες γραμμές) ήταν λιγότερο κοστοβόρες κατά τη συγγραφή. [107,108] Μία εμπειρική μελέτη όμως

έδειξε ότι οι μικρές μέθοδοι (143 ή λιγότερες γραμμές) είναι 2,4 φορές λιγότερο ακριβές στην επιδιόρθωση, παρόλο που η αναλογία σφαλμάτων-γραμμών ήταν 23 τοις εκατό μεγαλύτερη. [31] Τέλος, η IBM οδηγήθηκε στο συμπέρασμα ότι πιο επιρρεπείς σε σφάλματα είναι οι μέθοδοι που έχουν τουλάχιστον 500 γραμμές κώδικα, ενώ μια άλλη μελέτη έδειξε ότι τις λιγότερες διορθώσεις απαιτούν οι μέθοδοι που κυμαίνονται μεταξύ 100 και 150 γραμμών. [110,111]

Από τα παραπάνω μπορεί κανείς να διαπιστώσει ότι δεν υπάρχει συγκεκριμένο κατώφλι για να χαρακτηριστεί μια μέθοδος μακροσκελής. Αντιθέτως υπάρχουν διαφορετικά κατώφλια, ανάλογα με την πολιτική που ακολουθεί ο εκάστοτε προγραμματιστής (πχ. λιγότερο ακριβή συγγραφή κώδικα / λιγότερο ακριβή επιδιόρθωση κώδικα / χαμηλότερη αναλογία σφαλμάτων-γραμμών κώδικα / κ.ά.). Παρόλα αυτά μέθοδοι με περισσότερες από 200 γραμμές γίνονται δύσκολα κατανοητές και θα πρέπει να αποφεύγονται. [34]

Μια διαφορετική - εμπειρική - προσέγγιση στον ορισμό της μακροσκελούς μεθόδου δίνουν οι Beck [25] και Fowler [23]. Ο πρώτος αναφέρει πως μια μέθοδος πρέπει να είναι τόση όση χρειάζεται ώστε να εσωκλείει ένα και μόνο κομμάτι λογικής, ανεξάρτητα από το πλήθος των γραμμών της. Έτσι διευκολύνεται η κατανόηση, η επαναχρησιμοποίηση και ο επαναορισμός (override) των μεθόδων, ενώ ταυτόχρονα αποτρέπεται η επανάληψη πολλαπλών γραμμών κώδικα. Από την άλλη, ο Fowler προτείνει την «αντικατάσταση» των σχολίων με μικρές μεθόδους που εξυπηρετούν έναν και μόνο σκοπό, δίνοντάς τους επαρκώς περιγραφικό όνομα.

5.2 Extract Method

Η αναδόμηση Extract Method είναι ο συνηθέστερος τρόπος αντιμετώπισης μιας μακροσκελούς μεθόδου. Με αυτό τον τρόπο ένα τμήμα κώδικα της αρχικής μεθόδου μεταφέρεται σε νέα μέθοδο, έτσι ώστε να μειώνεται το μέγεθος της πρώτης, να αυξάνεται η συνεκτικότητα και να διευκολύνεται η συντήρηση του λογισμικού.

Η εφαρμογή της συγκεκριμένης αναδόμησης δεν είναι πάντα απλή. Για τη διευκόλυνση της διαδικασίας είναι χρήσιμο να μειωθεί το πλήθος των προσωρινών

μεταβλητών (Replace Temp with Query [23]) ή να διαχωριστούν οι λειτουργίες μιας προσωρινής μεταβλητής (Split Temporary Variable [23]). Κάποιες φορές η εφαρμογή της αναδόμησης Extract Method είναι ακριβή καθώς οι τοπικές μεταβλητές – σπανιότερα και οι παράμετροι – περιπλέκονται σε μεγάλο βαθμό. Σε αυτή την περίπτωση καταφεύγουμε στην αναδόμηση Replace Method with Method Object (βλ. Κεφ. 3.4).

Στη συνέχεια παρουσιάζεται η αναδόμηση Extract Method μέσα από ένα παράδειγμα. Το όνομα της νέας μεθόδου είναι επιλεγμένο έτσι, ώστε να μην απαιτούνται σχόλια για την κατανόηση του κώδικα.

```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}
```

Πίνακας 7: Κώδικας πριν την εφαρμογή της Extract Method

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}
```

Πίνακας 8: Κώδικας μετά την εφαρμογή της Extract Method

Κίνητρα για την εφαρμογή της Extract Method υπάρχουν πολλά και αυτό φαίνεται από τη συχνότητα χρήσης της. Το μεγάλο μέγεθος μιας μεθόδου και η ανάγκη χρήσης σχολίων για την επεξήγηση ενός τμήματος κώδικα είναι σαφείς ενδείξεις ότι πρέπει να τροποποιηθεί ο κώδικας. Μια διαφορετική ένδειξη θα ήταν η ανεπαρκής αντιστοιχία του ονόματος μιας μεθόδου και των ενεργειών που επιτελεί. Αν η αναδόμηση βελτιώνει τη σαφήνεια, τότε προτείνεται η χρήση της.

5.3 Σύνθετη Μέθοδος (Complex Method)

Σύνθετη μέθοδος καλείται κάθε μακροσκελής μέθοδος η οποία έχει αυξημένη πολυπλοκότητα. [24] Σε τέτοιες μεθόδους παρατηρείται δυσκολία στην εφαρμογή της αναδόμησης Extract Method. Για παράδειγμα, μπορεί να χρειαστεί ένα εξαχθέν τμήμα κώδικα να επιστρέφει δύο τιμές στην αρχική μέθοδο. Αυτό δεν μπορεί όμως να συμβεί λόγω του περιορισμού της μιας επιστρεφόμενης τιμής που θέτει η γλώσσα προγραμματισμού. Δίνεται η ακόλουθη μέθοδος για την καλύτερη κατανόηση των ανωτέρω.

```
   public int gamma (int inputVal, int quantity, int
yearToDate) {
       int importantValue1 = (inputVal * quantity) + delta();
       int importantValue2 = (inputVal * yearToDate) + 100;
       if ((yearToDate - importantValue1) > 100)
           importantValue2 -= 20;
       int importantValue3 = importantValue2 * 7;
       // and so on.
       return importantValue3 - 2 * importantValue1;
   }
```

Πίνακας 9: Παράδειγμα σύνθετης μεθόδου

Η σκέψη είναι οι δύο πρώτες γραμμές τις μεθόδου να εξαχθούν σε μία νέα μέθοδο. Όμως κάτι τέτοιο δεν είναι εφικτό, αφού πρέπει να επιστραφούν δύο τιμές ώστε να αποθηκευθούν στις μεταβλητές importantValue1 και importantValue2. Εφόσον μία μέθοδος μπορεί να επιστρέφει μόνο μία τιμή, πρέπει να βρεθεί διαφορετικός τρόπος προσέγγισης για την εξαγωγή του κώδικα.

Μια άλλη περίπτωση είναι η κλήση της εξαχθείσας μεθόδου να πραγματοποιείται με πολλές παραμέτρους λόγω της μεγάλης εξάρτησης με την αρχική μέθοδο. Για παράδειγμα, η εξαγωγή της if-else εντολής της epsilon σε νέα μέθοδο, θα έχει το αποτέλεσμα που φαίνεται στον Πίνακα 11. Οι παράμετροι που πρέπει να μεταβιβαστούν στη νέα μέθοδο για ένα τόσο μικρό παράδειγμα είναι τέσσερις. Αν η αρχική μέθοδος ήταν ακόμα μεγαλύτερη και τα σημεία που έπρεπε να εξαχθούν ακόμη περισσότερα, τότε το πρόβλημα θα πολλαπλασιαζόταν.

Εκτός από την επιβάρυνση του συστήματος, η συνεχής ανταλλαγή παραμέτρων καθιστά δυσκολότερη την κατανόηση της ροής του κώδικα. Επίσης, οι μέθοδοι με πολλές παραμέτρους δυσκολεύουν τον προγραμματιστή στα σημεία που αυτές καλούνται. Σύμφωνα με τον McConnell [34], οι 7 παράμετροι είναι το ανώτερο όριο ώστε να γίνονται κατανοητές. Όταν ξεπερνάται αυτό το όριο, οι προγραμματιστές δε δίνουν την απαραίτητη προσοχή στις παραμέτρους και αυτό έχει σαν αποτέλεσμα να αυξάνεται η πιθανότητα σφάλματος. Το ιδανικό πλήθος παραμέτρων για τον Martin [35] είναι 0. Αποδεκτό πλήθος παραμέτρων αποτελούν η 1 και οι 2, ενώ θα πρέπει να αποφεύγονται οι 3 παράμετροι όπου αυτό είναι εφικτό. Για περισσότερες από 3 παραμέτρους θα πρέπει να υπάρχει πολύ ειδικός λόγος, αλλά σε καμία περίπτωση δεν ενδείκνυται η συγγραφή τέτοιων μεθόδων.

```
public int epsilon (int inputVal,int quantity,int yearToDate) {
    int importantValue1 = (inputVal * quantity) + delta();
    int importantValue2 = (inputVal * yearToDate) + 100;

    if ((yearToDate - importantValue1) > 100)
        importantValue2 = importantValue1;
    else if ((yearToDate - importantvalue1 < 100)
        importantvalue2 = importantvalue3;
    else
```

```
        importantvalue2 = importantvalue1 + importantvalue3;

    int importantValue3 = importantValue2 * 7;

    return importantValue3 - 2 * importantValue1;
}
```

Πίνακας 10: Κώδικας πριν την εφαρμογή της Extract Method

```
public int epsilon (int inputVal,int quantity,int yearToDate) {
    int importantValue1 = (inputVal * quantity) + delta();
    int importantValue2 = (inputVal * yearToDate) + 100;

    importantValue2 = zeta(yearToDate, importantvalue1,
importantvalue2, importantvalue3);

    int importantValue3 = importantValue2 * 7;

    return importantValue3 - 2 * importantValue1;
}

private int zeta (int yearToDate, int importantvalue1, int
importantvalue2, int importantvalue3)
{
    if ((yearToDate - importantValue1) > 100)
        return importantValue1;
    else if ((yearToDate - importantvalue1 < 100)
        return importantvalue3;
    else
        return importantvalue2+importantvalue1+importantvalue3;
   }
```

Πίνακας 11: Κώδικας μετά την εφαρμογή της Extract Method

Η επιλογή της κατάλληλης αναδόμησης για τον περιορισμό των παραμέτρων εξαρτάται από τη σημασιολογική συσχέτιση που έχουν μεταξύ τους οι παράμετροι. Αν η συσχέτιση είναι ικανή ώστε να δικαιολογεί τη δημιουργία ενιαίου αντικειμένου με πεδία τις παραμέτρους, τότε προτείνεται η χρήση της αναδόμησης Introduce Parameter Object [23]. Αν όμως δεν υπάρχει σημασιολογική συσχέτιση, τότε θα πρέπει να βρεθεί διαφορετικός τρόπος για να αποφευχθεί η ανταλλαγή πολλών παραμέτρων.

Την αποφυγή χρήσης μεθόδων με πολλές μη σχετικές παραμέτρους αλλά και τη λύση στην επιστροφή πολλαπλών τιμών από μία μέθοδο, έρχεται να δώσει η αναδόμηση Replace Method with Method Object.

5.4 Replace Method with Method Object

Η αναδόμηση Replace Method with Method Object μετατρέπει μία μέθοδο σε τάξη με πεδία τις τοπικές μεταβλητές και τις παραμέτρους της αρχικής μεθόδου. Με αυτό τον τρόπο διευκολύνεται η αποσύνθεση της μεθόδου, καθώς δεν απαιτείται πλέον επιστροφή πολλαπλών τιμών από τις εξαχθείσες μεθόδους ούτε απαιτείται η ανταλλαγή παραμέτρων, μιας και οι μεταβλητές γίνονται ορατές σε όλη την τάξη.

Τα βήματα με τα οποία γίνεται η εφαρμογή της αναδόμησης, όπως αυτά περιγράφονται από τους Beck [25] και Fowler [23], είναι τα ακόλουθα.

1. Δημιουργία τάξης με όνομα βασισμένο στο όνομα της αρχικής μεθόδου.

2. Δημιουργία πεδίων στην καινούρια τάξη για κάθε παράμετρο και τοπική μεταβλητή της μεθόδου καθώς και κάθε πεδίο που επεξεργάζεται. Αρχικά, είναι συνετό να δοθούν στα πεδία τα ίδια ονόματα που είχαν στην αρχική μέθοδο και εν συνεχεία, αν αυτό δεν είναι επιθυμητό, μπορούν να αλλάξουν. Ένα ακόμα final πεδίο ορίζεται για το αντικείμενο που φιλοξενούσε τη αρχική μέθοδο.

3. Δημιουργία κατασκευαστή με παραμέτρους μια αναφορά στο αρχικό αντικείμενο και τις παραμέτρους της αρχικής μεθόδου.

4. Αντιγραφή της μεθόδου στην καινούρια τάξη με όνομα compute(). Οι παράμετροι, οι τοπικές μεταβλητές και τα πεδία που χρησιμοποιούνταν στην αρχική μέθοδο αποτελούν πλέον πεδία της τάξης. Οι κλήσεις σε μεθόδους της αρχικής τάξης γίνονται μέσω του πεδίου αντικειμένου που έχει οριστεί.

5. Αντικατάσταση του αρχικού σώματος της μεθόδου στην παλιά τάξη με δημιουργία ενός αντικειμένου της νέας τάξης και κλήση της μεθόδου compute().

6. Αν στη μέθοδο ορίζονταν παράμετροι της αρχικής τάξης, τότε αυτές θα πρέπει να επαναοριστούν στην τροποποιημένη μέθοδο, μετά την κλήση της αρχικής από τη νέα τάξη.

Αφού ολοκληρωθεί η διαδικασία και βεβαιωθεί ότι ο προκύπτων κώδικας λειτουργεί όπως ο αρχικός, μπορεί πλέον να εφαρμοστεί η αναδόμηση Extract Method στη νέα μέθοδο.

Ακολουθεί ένα παράδειγμα χρήσης της αναδόμησης Replace Method with Method Object.

```
public class FirstClass {

    public int epsilon (int inputVal, int quantity, int
yearToDate) {
        int importantValue1 = (inputVal * quantity) + delta();
        int importantValue2 = (inputVal * yearToDate) + 100;

        if ((yearToDate - importantValue1) > 100)
            importantValue2 = importantValue1;
        else if ((yearToDate - importantvalue1 < 100)
            importantvalue2 = importantvalue3;
        else
            importantvalue2 = importantvalue1+importantvalue3;

        int importantValue3 = importantValue2 * 7;
```

```
        return importantValue3 - 2 * importantValue1;
    }
}
```

Πίνακας 12: Κώδικας πριν την εφαρμογή της αναδόμησης Replace Method with Method Object

```
public class FirstClass {

    public int epsilon (int inputVal, int quantity, int
yearToDate) {
        return new SecondClass(inputVal, quantity,
yearToDate).compute();
    }
}

public class SecondClass {

    private final FirstClass firstClass;
    private int inputVal;
    private int quantity;
    private int yearToDate;
    private int importantvalue1;
    private int importantvalue2;
    private int importantvalue3;

    public SecondClass (FirstClass source, int _inputVal, int
_quantity, int _yearToDate) {
        firstClass = source;
        inputVal = _inputVal;
        quantity = _quantity;
        yearToDate = _yearToDate;
```

```
    }

    public int compute () {
        int importantValue1 = (inputVal * quantity) + delta();
        int importantValue2 = (inputVal * yearToDate) + 100;

        if ((yearToDate - importantValue1) > 100)
            importantValue2 = importantValue1;
        else if ((yearToDate - importantvalue1 < 100)
            importantvalue2 = importantvalue3;
        else
            importantvalue2 = importantvalue1+importantvalue3;

        int importantValue3 = importantValue2 * 7;

        return importantValue3 - 2 * importantValue1;
    }
}
```

Πίνακας 13: Κώδικας μετά την εφαρμογή της αναδόμησης Replace Method with Method Object

Κεφάλαιο 6

Αυτοματοποίηση Αναδόμησης Replace Method with Method Object

Ορισμένες απλές αναδομήσεις έχουν υιοθετηθεί από σύγχρονα περιβάλλοντα ανάπτυξης κώδικα. Σε μια τέτοια χρήση αυτοματοποιημένης αναδόμησης στοχεύει και η παρούσα εργασία, με την επέκταση του πρόσθετου JDeodorant για το περιβάλλον του Eclipse. Το πρόσθετο θα πρέπει να ανιχνεύει τις σύνθετες μεθόδους και να προτείνει στον προγραμματιστή την αναδόμησή τους. Η λύση που προτείνεται είναι η εφαρμογή της αναδόμησης Replace Method with Method Object για τη συγκεκριμένη μέθοδο που επιλέγει ο χρήστης. Αφού μελετηθεί η προτεινόμενη λύση, ο χρήστης είναι αυτός που αποφασίζει αν τελικά θα την εφαρμόσει ή θα διατηρήσει τον κώδικα ως έχει.

6.1 Παρουσίαση Αρχικού Εργαλείου

Το JDeodorant στην έκδοση που παραχωρήθηκε στον γράφοντα περιελάμβανε τις ακόλουθες λειτουργίες.

Ανίχνευση Feature Envy

Παρατηρείται το φαινόμενο κάποιες μέθοδοι μιας τάξης να «ενδιαφέρονται» περισσότερο για δεδομένα που βρίσκονται σε κάποια άλλη. Αυτό έχει ως αποτέλεσμα να καλούν πολλές φορές μεθόδους οι οποίες δίδουν πρόσβαση στα δεδομένα της δεύτερης.

Προτεινόμενη Αναδόμηση: Move Method

Σε αυτή την περίπτωση η προτεινόμενη αναδόμηση είναι η Move Method. Η μέθοδος που χρησιμοποιεί περισσότερο δεδομένα μιας άλλης τάξης, μεταφέρεται σε αυτήν. Στην περίπτωση που η μέθοδος επεξεργάζεται δεδομένα από πολλές διαφορετικές τάξεις, τότε μεταφέρεται στην τάξη στην οποία βρίσκονται τα περισσότερα από αυτά τα δεδομένα.

Ανίχνευση Type Checking

Ένα ακόμα φαινόμενο το οποία χρήζει ιδιαίτερης μεταχείρισης είναι το Type Code, κατά το οποίο γίνεται έλεγχος ανάμεσα σε διαφορετικές τιμές πιθανές τιμές.

Προτεινόμενη Αναδόμηση: Replace Type Code with State/Strategy

Σε αυτή την περίπτωση η αναδόμηση η οποία εφαρμόζεται είναι η Replace Type Code with State/Strategy. Δημιουργείται μια νέα τάξη που το όνομά της περιγράφει το σκοπό του type code και στη συνέχεια υποτάξεις για κάθε μία τιμή του type code.

Πέραν των παραπάνω, άλλες κακές οσμές που ανιχνεύονται και προτείνεται η αντιμετώπιση τους είναι οι εξής:

Ανίχνευση Long Method

Προτεινόμενη Αναδόμηση: Extract Method

Ανίχνευση Complex Method

Προτεινόμενη Αναδόμηση: Replace Method with Method Object

Για περισσότερες λεπτομέρειες για τις παραπάνω οσμές και αναδομήσεις, ο αναγνώστης παραπέμπεται στο κεφάλαιο 5.

Η τελευταία λειτουργία είναι αυτή που θα μας απασχολήσει στο παρόν κεφάλαιο. Το εργαλείο υπολογίζει το slice profile κάθε μεθόδου του υπό εξέταση προγράμματος. Ο υπολογισμός του slice profile προκύπτει από το backward slicing των τοπικών μεταβλητών της μεθόδου. Στη συνέχεια υπολογίζεται το Overlap και το Coverage της μεθόδου με βάση τα παραπάνω. Επίσης, με τη χρήση του αλγορίθμου Spectral Graph Partitioning χωρίζεται εικονικά η κάθε μέθοδος σε δύο τμήματα. Σκοπός αυτού του διαχωρισμού είναι να εντοπισθούν τμήματα της μεθόδου που δεν έχουν μεγάλη συσχέτιση μεταξύ τους. Με τον ίδιο τρόπο προτείνεται και η Extract Method σε άλλη λειτουργία του εργαλείου.

Κατά την εκτέλεσή του, το εργαλείο εμφανίζει στο χρήστη για κάθε μέθοδο, το όνομά της, τις τιμές των μετρικών Overlap και Coverage, δύο στήλες με τις μεταβλητές του κάθε τμήματος που προκύπτει από το Spectral Graph Partitioning και άλλες δύο στήλες με τις αντίστοιχες τιμές Overlap.

Η παραπάνω διαδικασία περιγράφει τον τρόπο με τον οποίο υπολογίζονται οι μετρικές που έχουν ως στόχο να παρουσιάσουν τα στοιχεία των μεθόδων στον χρήστη. Ο τελευταίος είναι αυτός που κρίνει σε ποια μέθοδο μεταξύ των εμφανιζόμενων ενδείκνυται να εφαρμοστεί η αναδόμηση Replace Method with Method Object. Ο χρήστης είναι σε θέση να καλέσει τη διαδικασία ανίχνευσης των μεθόδων ενός προγράμματος μέσω της διεπαφής χρήστη που προσφέρει το εργαλείο. Ακόμα, έχει τη δυνατότητα να επιλέξει την εξαγωγή των πληροφοριών σε αρχείο και την εφαρμογή της αναδόμησης σε συγκεκριμένη μέθοδο.

6.2 Μέθοδος Εντοπισμού Υποψήφιων Μεθόδων προς Αναδόμηση

Στο συγκεκριμένο υποκεφάλαιο περιγράφεται συνοπτικά η διαδικασία που ακολουθεί το εργαλείο, όπως έχει διαμορφωθεί στα πλαίσια της διπλωματικής εργασίας. Στα επόμενα υποκεφάλαια αναλύεται περαιτέρω κάθε στάδιο της διαδικασίας που δεν έχει ήδη αναφερθεί στο Κεφάλαιο 2.

Παρατηρήθηκε ότι το εργαλείο στην αρχική του μορφή δίνει πολλές μεθόδους ως υποψήφιες για αναδόμηση. Αυτό έχει σαν αποτέλεσμα να μην είναι σαφή τα όρια μεταξύ των περιπτώσεων στις οποίες η εφαρμογή της αναδόμησης είναι αναγκαία, των περιπτώσεων που η εφαρμογή της κρίνεται από τον προγραμματιστή ανάλογα με την οπτική του και των περιπτώσεων που δεν προσφέρουν κάποιο ιδιαίτερο όφελος. Έτσι, η παραγωγή πολλών αποτελεσμάτων οδηγεί σε υπερπληροφόρηση και μειώνει την αξία του εργαλείου. Για το λόγο αυτό έπρεπε να βρεθεί ένας τρόπος να περιοριστούν τα αποτελέσματα και να αναδειχθούν οι μέθοδοι που έχουν αυξημένη πιθανότητα να αποτελούν κακή οσμή.

Το πρώτο στάδιο της διαδικασίας περιλαμβάνει τη δημιουργία του Αφηρημένου Συντακτικού Δέντρου του λογισμικού. Από την ανάλυσή του προκύπτουν πληροφορίες για τη δομή του προγράμματος που χρησιμοποιούνται κατά κόρον στη συνέχεια από το εργαλείο.

Στη συνέχεια, πραγματοποιείται ένα φιλτράρισμα των μεθόδων που θα εξετασθούν από το εργαλείο. Έτσι, απορρίπτονται εξαρχής οι μέθοδοι των οποίων τα χαρακτηριστικά δηλώνουν εμφανώς ότι δε χρήζουν της αναδόμησης Replace Method with Method Object.

Κύριος σκοπός των αναδομήσεων είναι η μείωση του κόστους. Το κόστος και ο χρόνος είναι άρρηκτα συνδεδεμένα, καθώς λιγότερος χρόνος συνεπάγεται μικρότερο κόστος. Επομένως, ένα εργαλείο το οποίο στοχεύει στην αυτοματοποίηση λειτουργιών των αναδομήσεων, θα πρέπει να εξυπηρετεί και τους σκοπούς αυτών, όντας εύχρηστο και γρήγορο. Γι' αυτό το λόγο οι συνθήκες που περιορίζουν το πλήθος των υπό εξέταση μεθόδων έχουν τοποθετηθεί σε στρατηγικά σημεία, ώστε να αποφεύγονται οι άσκοπες λειτουργίες.

Σε επόμενο βήμα, δημιουργείται ο Γράφος Ροής Ελέγχου από τον οποίο παράγεται ο Γράφος Εξαρτήσεων. Από τον τελευταίο εξάγονται τα slices ώστε να υπολογισθούν οι τιμές των Overlap και Coverage για τη μέτρηση της συνεκτικότητας. Επεκτείνοντας την αρχική υλοποίηση, πέρα από τα backward slices, υπολογίζονται και τα forward slices κάθε μεταβλητής. Ως αποτέλεσμα δημιουργούνται τριών ειδών slice profiles, αυτά που βασίζονται στα backward, τα forward και τα metric slices.

Η αρχική υλοποίηση περιελάμβανε μόνο τον υπολογισμό των slices των τοπικών μεταβλητών. Για τη δημιουργία μιας πιο σφαιρικής άποψης για την πολυπλοκότητα της μεθόδου, προστέθηκε ο υπολογισμός των slices των παραμέτρων στο slice profile της. Πιο συγκεκριμένα, διατηρήθηκε η δημιουργία του slice profile βασισμένου στις τοπικές μεταβλητές, ενώ προστέθηκε το slice profile βασισμένο σε παραμέτρους καθώς και ένα slice profile για το σύνολο τοπικών μεταβλητών και παραμέτρων.

Τέλος, δεν πρέπει να παραλειφθεί ότι στις υποψήφιες προς αναδόμηση μεθόδους δεν εμφανίζονται αυτές για τις οποίες ισχύουν κάποια ιδιαίτερα χαρακτηριστικά. Δηλαδή απορρίπτονται οι μέθοδοι για τις οποίες δεν ισχύουν ορισμένες προϋποθέσεις. Η παραβίαση αυτών είτε καθιστά αδύνατη την αναδόμηση είτε οδηγεί σε τροποποίηση της συμπεριφοράς του προγράμματος.

Ο χρήστης, μέσω της γραφικής διεπαφής, έχει τη δυνατότητα να εκκινήσει την παραπάνω διαδικασία ώστε να ανιχνευθούν οι μέθοδοι που ενδέχεται να περιλαμβάνουν κακή οσμή Complex Method. Οι μέθοδοι που εμφανίζονται ικανοποιούν συγκεκριμένα κριτήρια που σύμφωνα με την αξιολόγηση, υποδεικνύουν σε κάποιο βαθμό την ύπαρξη της κακής οσμής Complex Method.

Αφού εμφανιστούν οι μέθοδοι, μαζί με τα διάφορα χαρακτηριστικά τους, ο χρήστης μπορεί να επιλέξει την αποθήκευσή τους σε ένα αρχείο CSV, στο οποίο μπορεί να απευθυνθεί αργότερα. Επιλέγοντας μία μέθοδο από τη λίστα, ο χρήστης έχει την δυνατότητα να επιβλέψει την προτεινόμενη αναδόμηση και να αποφασίσει αν επιθυμεί την εφαρμογή της ή όχι. Στην περίπτωση που επιλέξει να την εφαρμόσει τότε ο κώδικας μεταβάλλεται αντίστοιχα.

6.2.1 Φιλτράρισμα Πεδίου Ορισμού

Δίδεται η δυνατότητα στο χρήστη να αποκλείσει μαζικά μεθόδους που θεωρεί εξαρχής ότι δε χρειάζονται αναδόμηση, με βάση συγκεκριμένα χαρακτηριστικά. Αυτό γίνεται με τη χρήση ενός αρχείου παραμέτρων (configuration file). Οι επιλογές που δίνονται στο χρήστη είναι το μέγιστο πλήθος των παραμέτρων της μεθόδου που είναι αποδεκτό, το μέγιστο πλήθος των τοπικών μεταβλητών και το μέγιστο μήκος της μεθόδου. Αρχικά εξετάζεται αν το πλήθος των παραμέτρων ή το πλήθος των τοπικών

μεταβλητών ξεπερνά το όριο του χρήστη και στη συνέχεια ελέγχεται το μήκος της μεθόδου.

Αναλυτικά οι διαφορετικές παράμετροι και οι προεπιλεγμένες τιμές τους, φαίνονται παρακάτω.

max_parameters	3
max_local_variables	3
max_body_length	10

Πίνακας 14: Επιλογές και προεπιλεγμένες τιμές αρχείου παραμέτρων

Η επιλογή της προεπιλεγμένης τιμής για το μέγιστο αποδεκτό πλήθος των παραμέτρων έγινε βάσει της βιβλιογραφίας που παρουσιάστηκε στο υποκεφάλαιο 5.3. Η επιλογή για τις τοπικές μεταβλητές ακολουθεί μια παρόμοια συλλογιστική πορεία. Τέλος, επειδή δεν έχει νόημα η εφαρμογή της συγκεκριμένης αναδόμησης για πολύ μικρές μεθόδους, αποφασίστηκε να μην εξετάζονται μέθοδοι με μήκος μικρότερο του 10. Ωστόσο, οι προεπιλεγμένες τιμές είναι ενδεικτικές και έγκειται στην κρίση του προγραμματιστή η επιλογή αυτών που τον ικανοποιούν.

6.2.2 Προϋποθέσεις Εφαρμογής Αναδόμησης

Έχουν αποκλειστεί οι περιπτώσεις στις οποίες δεν μπορεί να εφαρμοστεί ή είναι αρκετά σύνθετη η εφαρμογή της αναδόμησης. Πιο συγκεκριμένα, δεν εμφανίζονται στο χρήστη όσες μέθοδοι περιέχουν κλήσεις προς την υπερτάξη της αρχικής τάξης. Ίσως από μόνο του αυτό υποδηλώνει τη στενή σχέση μεταξύ μεθόδου και τάξης.

Άλλη περίπτωση, είναι μία μέθοδος να πραγματοποιεί κλήσεις προς ιδιωτικές μεθόδους της αρχικής τάξης. Ένας τρόπος να αντιμετωπισθεί το συγκεκριμένο πρόβλημα είναι να αλλάξει η ορατότητα των συγκεκριμένων μεθόδων. Αυτό όμως δεν είναι πάντα επιθυμητό. Ένας άλλος τρόπος αντιμετώπισης, είναι η εφαρμογή της

αναδόμησης Inline Method για τις συγκεκριμένες μεθόδους. Ωστόσο η εφαρμογή της συγκεκριμένης αναδόμησης ενέχει αυξημένη πολυπλοκότητα καθώς θα πρέπει να εντοπισθούν όλα τα σημεία κλήσης της μεθόδου, όπως επίσης και να εξετασθεί αν η μέθοδος αποτελεί επαναορισμό μεθόδου της υπερτάξης.

6.2.3 Υπολογισμός Μετρικών

Το εργαλείο υπολογίζει ένα σύνολο χαρακτηριστικών τα οποία αφορούν στις μεθόδους των προγραμμάτων. Τα χαρακτηριστικά αυτά υπολογίζονται για να μπορέσει να κριθεί η συμβολή τους στην ανάδειξη των μεθόδων που η αναδόμηση Replace Method with Method Object είναι επιθυμητή. Τα χαρακτηριστικά αυτά παρουσιάζονται συνοπτικά παρακάτω.

- Όνομα Μεθόδου: Δεν αποτελεί κριτήριο αξιολόγησης. Χρησιμοποιείται για να γίνεται η αντιστοιχία κριτηρίων και μεθόδου.

- Μήκος Μεθόδου: Αντιστοιχεί στο μήκος του σώματος της μεθόδου. Υπολογίζεται ως το σύνολο των κόμβων της μεθόδου στο Γράφημα Ελέγχου Ροής.

- Πλήθος Παραμέτρων: Αντιστοιχεί στο πλήθος των παραμέτρων της μεθόδου.

- Τύπος Επιστροφής: Αντιστοιχεί στον τύπο της μεταβλητής που επιστρέφεται από τη μέθοδο. Οι πιθανές τιμές είναι void και non-void.

- Ορατότητα Μεθόδου: Αντιστοιχεί στην ορατότητα της μεθόδου.

- Πλήθος Τοπικών Μεταβλητών: Αντιστοιχεί στο σύνολο των τοπικών μεταβλητών της μεθόδου.

- Πλήθος Πεδίων Τάξης: Αντιστοιχεί στο σύνολο των πεδίων της τάξης που φιλοξενεί τη μέθοδο.

- Πλήθος Πεδίων που Προσπελαύνονται: Αντιστοιχεί στο σύνολο των πεδίων που προσπελαύνονται από τη μέθοδο.

- Πλήθος Αναθέσεων Τιμών σε Πρωταρχικού Τύπου Πεδία: Αντιστοιχεί στο σύνολο των πεδίων πρωταρχικού τύπου, στα οποία γίνεται ανάθεση τιμής στο σώμα της μεθόδου.

• Πλήθος Μεθόδων Τάξης: Αντιστοιχεί στο σύνολο των μεθόδων που έχει η τάξη που φιλοξενεί την υπό εξέταση μέθοδο.

• Πλήθος Μεθόδων Δημόσιας/Ιδιωτικής/Προστατευμένης/Προεπιλεγμένης Ορατότητας: Αντιστοιχούν στο σύνολο των μεθόδων της αρχικής τάξης που έχουν την αντίστοιχη ορατότητα.

• Metric/Backward/Forward Overlap: Αντιστοιχούν στην τιμή του Overlap με βάση το slice profile που δημιουργείται από metric/backward/forward slicing αντίστοιχα.

• Metric/Backward/Forward Coverage: Αντιστοιχούν στην τιμή του Coverage με βάση το slice profile που δημιουργείται από metric/backward/forward slicing αντίστοιχα.

• Metric/Backward/Forward Local Variable Overlap: Αντιστοιχούν στην τιμή του Overlap με βάση το slice profile των τοπικών μόνο μεταβλητών που δημιουργείται από metric/backward/forward slicing αντίστοιχα.

• Metric/Backward/Forward Local Variable Coverage: Αντιστοιχούν στην τιμή του Coverage με βάση το slice profile των τοπικών μόνο μεταβλητών που δημιουργείται από metric/backward/forward slicing αντίστοιχα.

• Metric/Backward/Forward Parameter Overlap: Αντιστοιχούν στην τιμή του Overlap με βάση το slice profile μόνο των παραμέτρων της μεθόδου που δημιουργείται από metric/backward/forward slicing αντίστοιχα.

• Metric/Backward/Forward Parameter Coverage: Αντιστοιχούν στην τιμή του Coverage με βάση το slice profile μόνο των παραμέτρων της μεθόδου που δημιουργείται από metric/backward/forward slicing αντίστοιχα.

• Minimum Metric/Backward/Forward Coverage: Αντιστοιχεί στο Coverage του μικρότερου metric/backward/forward slice που εντοπίζεται στη μέθοδο.

• Minimum Metric/Backward/Forward Coverage: Αντιστοιχεί στο Coverage του μεγαλύτερου metric/backward/forward slice που εντοπίζεται στη μέθοδο.

Κεφάλαιο 7

Αξιολόγηση

Για την αξιολόγηση των αποτελεσμάτων του εργαλείου που αναπτύχθηκε χρησιμοποιήθηκαν διάφορα προγράμματα. Οι μέθοδοι που κρίνονται ως Complex Method και είναι υποψήφιες για αναδόμηση είναι πολλές σε σχέση με το μέγεθος του εκάστοτε προγράμματος. Αυτές οι μέθοδοι εξετάσθηκαν μια προς μια και αποφασίστηκε με βάση την προσωπική κρίση και εμπειρία των εξεταστών ποιες αποτελούν όντως περιπτώσεις Complex Method και προτείνεται η αναδόμηση Replace Method with Method Object ή είναι Long Method και απαιτούν την εφαρμογή Extract Method ή τίποτα από τα δύο.

7.1 Μεθοδολογία

Το εργαλείο χρησιμοποιήθηκε σε διάφορα ενδεικτικά προγράμματα προκειμένου να γίνει η ανάλυση του κώδικα και να εξαχθούν οι μετρικές και τα χαρακτηριστικά που αφορούν στις διάφορες μεθόδους. Τα αποτελέσματα αποθηκεύτηκαν σε αρχεία CSV και με την εξέταση των μεθόδων που περιλαμβάνονταν σε αυτό αποφασίστηκε αν η κάθε μέθοδος παρουσιάζει κάποια κακιά οσμή και η αντίστοιχη αναδόμηση που απαιτείται. Συγκεκριμένα, στο CSV προστέθηκαν τα πεδία Replace Method with Method Object και Extract Method και για κάθε μέθοδο σημειώθηκε αν η εφαρμογή της αντίστοιχης αναδόμησης είναι επιθυμητή ή όχι.

Το τελικό αρχείο χρησιμοποιήθηκε ως είσοδος στο Weka. Το Weka είναι ένα λογισμικό που παρέχει την υλοποίηση αλγορίθμων μηχανικής μάθησης για την εξόρυξη δεδομένων. Περιλαμβάνει διάφορα εργαλεία για την προεπεξεργασία δεδομένων, την ταξινόμηση, την παλινδρόμηση (regression), τη συσταδοποίηση

(clustering), την εξαγωγή κανόνων συσχέτισης και την απεικόνιση. Για την αξιολόγηση των δεδομένων των προγραμμάτων χρησιμοποιήθηκε ο J48 που αποτελεί υλοποίηση του αλγορίθμου C4.5. Είναι ένας αλγόριθμος που χρησιμοποιείται για την παραγωγή δέντρων αποφάσεων.

Βάσει των αποτελεσμάτων που παρήχθησαν από το Weka, εντοπίστηκαν τα χαρακτηριστικά της μεθόδου και οι μετρικές που υποδεικνύουν με μεγαλύτερη ακρίβεια την ενδεχόμενη ανάγκη χρήσης Replace Method with Method Object.

7.2 Αποτελέσματα

Παρήχθησαν διάφορα αποτελέσματα από τη χρήση του εργαλείου του Weka. Προσθέτοντας και αφαιρώντας πεδία που λαμβάνονται υπόψη για την παραγωγή του δέντρου αποφάσεων, αξιολογήθηκαν τα διάφορα αποτελέσματα. Ο κανόνας που παρουσιάζεται παρακάτω θεωρήθηκε ο καταλληλότερος για να κριθεί αν κάποια μέθοδος είναι υποψήφια για την αναδόμηση Replace Method with Method Object.

```
Local_Variables <= 17
|       Class_Fields <= 21: No (366.0/1.0)
|       Class_Fields > 21
|       |       Length <= 42: No (18.0)
|       |       Length > 42: Yes (3.0)
Local_Variables > 17
|       Length <= 74
|       |       Local_Variables <= 21: No (9.0)
|       |       Local_Variables > 21
|       |       |       F_Overlap <= 0.141: No (5.0/1.0)
|       |       |       F_Overlap > 0.141: Yes (2.0)
|       Length > 74: Yes (5.0)
```

Πίνακας 15: Κανόνας ανίχνευσης Complex Method

Ο λόγος που επιλέχθηκε ο συγκεκριμένος κανόνας είναι γιατί παρουσίαζε το μεγαλύτερο ποσοστό ακρίβειας (precision) και ανάκλησης (recall). Η ακρίβεια περιγράφει το λόγο των μεθόδων που ο αλγόριθμος βάσει αυτών των κριτηρίων κατατάσσει σωστά ως υποψήφιες για αναδόμηση Replace Method with Method Object προς το σύνολο των μεθόδων που κατατάσσονται ως υποψήφιες. Η ανάκληση περιγράφει το λόγο των υποψήφιων μεθόδων που βρέθηκαν προς το σύνολο των υποψηφίων που υπάρχουν.

```
=== Stratified cross-validation ===
=== Summary ===
Correctly Classified Instances         400               98.0392 %
Incorrectly Classified Instances         8                1.9608 %
Kappa statistic                          0.5904
Mean absolute error                      0.0249
Root mean squared error                  0.1396
Relative absolute error                 41.9082 %
Root relative squared error             82.5573 %
Total Number of Instances              408
=== Detailed Accuracy By Class ===
               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC
Area  Class
                 0.5       0.005      0.75      0.5       0.6
0.795    Yes
                 0.995     0.5        0.985     0.995     0.99
0.795    No
Weighted Avg.    0.98      0.485      0.978     0.98      0.978    0.795


=== Confusion Matrix ===
   a   b   <-- classified as
   6   6 |   a = Yes
   2 394 |   b = No
```

Πίνακας 16: Αξιολόγηση του κανόνα ανίχνευσης

Τα αποτελέσματα δείχνουν ότι αρχικά εξετάζεται το πλήθος των τοπικών μεταβλητών της μεθόδου. Αν αυτό είναι μικρότερο από 17 τότε εξετάζεται το πλήθος των πεδίων της τάξης που φιλοξενεί τη μέθοδο. Αν και αυτό είναι μικρότερο από 21 τότε δε θεωρείται ότι η μέθοδος είναι υποψήφια προς αναδόμηση. Αν πάλι το πλήθος είναι σχετικά μεγάλο, τότε ελέγχεται και το μήκος της μεθόδου. Μέθοδοι οι οποίες έχουν μεγάλο μήκος θεωρούνται υποψήφιες, ενώ για τις μικρότερες δεν κρίνεται απαραίτητη η αναδόμηση.

Όταν το πλήθος των τοπικών μεταβλητών είναι μεγάλο εξετάζεται το μήκος της μεθόδου. Αν και το μήκος της μεθόδου είναι μεγάλο, τότε αυτή θεωρείται υποψήφια. Αν πάλι είναι κάτω από ένα όριο, εξετάζεται το Forward Overlap της μεθόδου.

Κεφάλαιο 8

Συμπεράσματα και Μελλοντική Έρευνα

Η αξιολόγηση έγινε βάσει εξέτασης και χαρακτηρισμού ως υποψηφίων για αναδόμηση ή μη, ενός συνόλου μεθόδων με συγκεκριμένα χαρακτηριστικά. Μέθοδοι οι οποίες δεν πληρούσαν τις προϋποθέσεις που αναλύθηκαν σε προηγούμενο κεφάλαιο δεν εξετάστηκαν. Τα συμπεράσματα βασίζονται σε υλοποίηση πλήθους μετρικών και εφαρμογή αυτών σε αρκετά προγράμματα διαφόρων μεγεθών και από διάφορους προγραμματιστές.

Θα πρέπει να αναφερθεί ότι στη διαδικασία της αξιολόγησης ενέχεται υποκειμενικότητα. Αυτή έγκειται στο γεγονός ότι ο χαρακτηρισμός περιπτώσεων ως υποψηφίων για αναδόμηση έγινε βάσει της προσωπικής κρίσης προγραμματιστών και όχι με κάποιο πιο αντικειμενικό κριτήριο. Άλλωστε, η απόφαση για την εφαρμογή ή μη μιας αναδόμησης δεν είναι απόφαση που λαμβάνεται με αντικειμενικά κριτήρια. Οι προτεραιότητες που τίθενται για την ανάπτυξη και τη συντήρηση του λογισμικού εξαρτώνται άμεσα από αυτόν που τις θέτει.

Για περαιτέρω αξιολόγηση θα πρέπει να γίνει εφαρμογή των κανόνων που εξήχθησαν και σε άλλα project και να γίνει επαλήθευση των αποτελεσμάτων από προγραμματιστές.

8.1 Συμπεράσματα

Η αυτοματοποίηση της ανίχνευσης σύνθετων μεθόδων είναι μια διαδικασία δύσκολη. Οι διάφορες μετρικές και οι παραλλαγές αυτών που εξετάσθηκαν, δεν είναι ικανές από μόνες τους να διακρίνουν με ακρίβεια περιπτώσεις σύνθετων μεθόδων που χρήζουν αναδόμησης.

Από τις μετρικές που αξιολογήθηκαν, καταλληλότερη για τη διαδικασία αναδείχθηκε η Forward Overlap, που βασίζεται στο overlap που προκύπτει από το forward slice της μεθόδου. Η εφαρμογή της μηχανικής μάθησης όμως, δεν υπέδειξε κάποια σαφή όρια για τις διάφορες μετρικές πέρα των οποίων η εκάστοτε μέθοδος θεωρείται σύνθετη.

Από την υλοποίηση προέκυψε ακόμα ότι η χρήση της μετρικής Backward Overlap δε χρησιμεύει ιδιαίτερα καθώς μηδενίζεται συχνά. Αντί αυτής προτείνεται η χρήση της Metric Overlap, η οποία έχει περιορίζει την πιθανότητα μηδενισμού και εκφράζει επίσης το βαθμό επικάλυψης των slices. Βασίζεται τόσο στα backward όσο και στα forward slices των μεταβλητών. Παρόλα αυτά και αυτή είναι περιοριστική και θα είχε ενδιαφέρον ο υπολογισμός και η χρήση παραλλαγών αυτής που δεν εξετάσθηκαν, όπως το μέσο overlap ζευγών από slices.

8.2 Μελλοντική Έρευνα

Αντικείμενο μελλοντικής έρευνας θα μπορούσε να αποτελέσει η δημιουργία ενός αλγορίθμου που δε θα ανιχνεύει μόνο μία συγκεκριμένη κακή οσμή, αλλά θα μπορεί να εντοπίζει ένα σύνολο αυτών και θα προτείνει την αντίστοιχη αναδόμηση για την εξάλειψη της κάθε μίας. Μία πρώτη προσέγγιση προς αυτή την κατεύθυνση έχει πραγματοποιηθεί από τον Θ. Χαϊκάλη. [24]

Ένα ακόμα ενδιαφέρον αντικείμενο μελλοντικής έρευνας αποτελεί η προσθήκη μηχανικής μάθησης στις λειτουργίες του εργαλείου. Να μπορεί δηλαδή το εργαλείο να ανιχνεύει τις αναδομήσεις που εφαρμόζει ο κάθε χρήστης και να συμπεραίνει τα προσωπικά του κριτήρια για τις περιπτώσεις εφαρμογής. Έτσι, το εργαλείο προσαρμόζεται στον κάθε χρήστη ξεχωριστά και τα αποτελέσματα που εμφανίζει εκφράζουν τις προσωπικές του αντιλήψεις για το πότε ενδείκνυται η εφαρμογή της κάθε αναδόμησης.

Καθώς ένα από τα κύρια προβλήματα που αντιμετωπίστηκαν ήταν ο εντοπισμός με ασφάλεια περιπτώσεων που χρήζουν αναδόμησης, μελλοντική εργασία θα

μπορούσε να αποτελέσει η αναζήτηση αναδομήσεων σε διάφορα αποθετήρια κώδικα των συστημάτων ελέγχου εκδόσεων και η εφαρμογή της μεθόδου σε προηγούμενες εκδόσεις αυτών. Με τον τρόπο αυτό μπορούν να συλλεχθούν αυτόματα στοιχεία για τις μεθόδους στις οποίες έχει πραγματοποιηθεί αναδόμηση και να παραχθούν κανόνες που θα είναι ευρύτερα αποδεκτοί.

Βιβλιογραφία

[1] Christopher Alexander, Sara Ishikawa, MurraySilverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, NewYork, 1977.

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissidess. *Design Patterns: Elements of Reusable Object-Oriented Software.*

[3] Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per Jönsson, Simon Kågström, Drazen Milicic, Frans Mårtensson, Kari Rönkkö, Piotr Tomaszewski. *Software quality attributes and trade-offs.* Blekinge Institute of Technology, June 2005.

[4] Εμμανουήλ Γιακουμάκης, Νικόλαος Διαμαντίδης. *Τεχνολογία Λογισμικού.* Εκδόσεις Σταμούλη, Αθήνα 2009.

[5] Timothy K. Shih, Ming-Chi Lee, Teh-Sheng Huang, Lawrence Y. Deng. *Assessing Software Quality Through Visualized Cohesion Metrics*. AJIS vol. 8 no. 2 May 2001.

[6] Irene Vanderfeesten, Hajo A. Reijers, Wil M.P. van der Aalst. *Evaluating workflow process designs using cohesion and coupling metrics*. Science Direct, Computers in Industry 59 (2008) 420–437.

[7] Norman E. Fenton, Shari Lawrence Pfleeger. Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company, 1997.

[8] O.M. Nierstrasz. A Survey of Object-Oriented Concepts. In Object-Oriented Concepts, Databases and Applications, ed. W. Kim and F. Lochovsky, pp. 3-21, ACM Press and Addison-Wesley, 1989.

[9] Johann Eder, Gerti Kappel, Michael Schrefl. *Coupling and Cohesion in Object-Oriented Systems*. 1992

[10] Klocwork & Tata Consultancy Services Ltd. *Architecture Driven Modernization Tutorial: Abstract Syntax Tree Metamodel Standard, ASTM Tutorial 1.0*. Software Revolution Inc, 2005.

[11] WWW: http://www.eclipse.org

[12] WWW: http://www.eclipse.org/jdt/ui/astview/index.php

[13] J. Ferrante, K. Ottenstein, and J. Warren. *The Program Dependence Graph and Its Use in Optimization*. ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319–349, 1987.

[14] Robert Gold. *Control Flow Graphs and Code Coverage*. International Journal of Applied Mathematics and Computer Science, 2010, Vol. 20, No. 4, 739–749

[15] Mary Jean Harrold, Gregg Rothermel, Alex Orso. *Representation and Analysis of Software*.

[16] John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 4th edition, 2007

[17] Mark Weiser. *Program Slicing*. In Proceedings of the 5th International Conference on Software Engineering, 1981.

[18] Frank Tip. *A Survey of Program Slicing Techniques.* CWI (Center for Mathematics and Computer Science) Amsterdam, The Netherlands, The Netherlands, 1994.

[19] K.Maruyama. *Automated Method-Extraction Refactoring by Using Block-Based Slicing*. In Proceedings of the Symposium on Software Reusability, pp.31-40, 2001.

[20] Nikolaos Tsantalis, Alexander Chatzigeorgiou, *Identification of Extract Method Opportunities*, 2009

[21] Linda M. Ott, Jeffrey J. Thuss. *Slice based metrics for estimating cohesion*. In Proceedings of the IEEE-CS International Metrics Symposium, pp. 71-81, 1993.

[22] Susan Horwitz, Thomas Reps, David Binkley. *Interprocedural Slicing Using Dependence Graphs*. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, pp. 26-60, January 1990

[23] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, Boston, MA, 1999.

[24] Θεόδωρος Χαϊκάλης, *Εντοπισμός Μεθόδων με Αλληλεπιδρώντα Slices σε Αντικειμενοστρεφή Συστήματα και Επίλυση μέσω Αναδομήσεων – Υλοποίηση στο Eclipse*, 2009

[25] Kent Beck, *Implementation Patterns*, Addison Wesley, 2007

[26] Thomas J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering SE-12, no. 4: 308-20, 1974

[27] Basili, V. R. , and B. T. Perricone. *Software Errors and Complexity: An Empirical Investigation*. Communications of the ACM 27, no. 1 (1): 42–52, 1984.

[28] Shen, Vincent Y. , et al. *Identifying Error-Prone Software - AnEmpirical Study.* IEEE Transactions on Software Engineering SE-11, no. 4 (4): 317–24, 1985.

[29] Card, David N. , Victor E. Church , and William W. Agresti. *An Empirical Study of Software Design Practices*. IEEE Transactions on Software Engineering SE-12, no. 2 (2): 264–71, 1986

[30] Card, David N. , with Robert L. Glass. *Measuring Software Design Quality*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[31] Selby, Richard W. , and Victor R. Basili. *Analyzing Error-Prone System Structure*. IEEE Transactions on Software Engineering SE-17, no. 2 (2): 141–52., 1991.

[32] Jones, Capers. *Programming Productivity*. New York, NY: McGraw-Hill, 1986a.

[33] Lind, Randy K., and K. Vairavan. *An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort*. IEEE Transactions on Software Engineering SE-15, no. 5 (5): 649–53, 1989.

[34] Steve McConnell, *Code Complete: A Practical Handbook of Software Construction*, Best Practices, Microsoft Press, 2004

[35] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2009

[36] D. Garvin, *What Does "Product Quality" Really Mean?* Sloan Management Review, Fall 1984, pp. 25-45.

[37] Barbara Kitchenham, Sari Lawrence Pfleeger, *Software Quality: The Elusive Target*, IEEE Software, IEEE Vol. 13, No. 1: January 1996, pp. 12-21

[38] Μπαμπινιώτης Γ. *Λεξικό της Νέας Ελληνικής Γλώσσας*, Κέντρο Λεξικολογίας ΕΠΕ, Δεύτερη Έκδοση, Αθήνα 2002

[39] Crosby P., *Quality is Free,* New York:McGraw Hill, 1979

[40] Juran J., Grynna F., *Quality Planning and Analysis,* Boston: McGraw Hill, 2d edition, New York 1995

[41] Donald E. Knuth, *Literate Programming*, Stanford, California: Center for the Study of Language and Information, CSLI Lecture Notes, no. 27, 1992

[42] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1986

[43] WWW: http://www.omg.org/spec/QVT

[44] WWW: http://www.tsri.com/technology.html