ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Περιβάλλον Ανάλυσης Αναφορών Αστοχιών Συστημάτων Λογισμικού Για Τον Εντοπισμό Ελαττωμάτων Πηγαίου Κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΚΡΥΣΤΑΛΕΝΙΑΣ ΤΑΤΣΗ

Επιβλέπων : Κωνσταντίνος Κοντογιάννης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Περιβάλλον Ανάλυσης Αναφορών Αστοχιών Συστημάτων Λογισμικού Για Τον Εντοπισμό Ελαττωμάτων Πηγαίου Κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΚΡΥΣΤΑΛΕΝΙΑΣ ΤΑΤΣΗ

Επιβλέπων : Κωνσταντίνος Κοντογιάννης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Οκτωβρίου 2016 .

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Κωνσταντίνος Κοντογιάννης Καθηγητής Ε.Μ.Π.	Ανδρέας-Γεώργιος Σταφυλοπάτης Καθηγητής Ε.Μ.Π.	Γεώργιος Στάμου Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

(Υπογραφή)

.....

ΚΡΥΣΤΑΛΕΝΙΑ ΤΑΤΣΗ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο σκοπός της παρούσας διπλωματικής εργασίας ήταν η σχεδίαση, η ανάλυση και υλοποίηση ενός βασικού συστήματος για εντοπισμό σφαλμάτων του πηγαίου κώδικα. Με το σύστημα αυτό ο χρήστης θα μπορεί να εντοπίζει ποιες μέθοδοι χρειάζονται τροποποίηση προκειμένου να επιλυθούν αστοχίες ενός προγράμματος λογισμικού. Η μεθοδολογία, που αναπτύχθηκε, στηρίζεται στις αναφορές σφαλμάτων που διατίθενται για τις εφαρμογές λογισμικού που χρησιμοποιήθηκαν για την επικείμενη μελέτη.

Συγκεκριμένα, δημιουργήθηκε ένα περιβάλλον το οποίο δέχεται ως είσοδο μία αναφορά σφάλματος και στη συνέχεια δημιουργεί μια λίστα με τις υπόλοιπες αναφορές του εκάστοτε υπό μελέτη συστήματος ώστε να δώσει στον προγραμματιστή μία εικόνα σχετικά με την ομοιότητα του συγκεκριμένου σφάλματος με προηγούμενα ήδη καταχωρημένα σφάλματα. Έπειτα οι μέθοδοι του συστήματος ταξινομούνται λαμβάνοντας υπόψιν τις συσχετίσεις μεταξύ τους καθώς και μετρικές σχέσεις για τον καθορισμό βάρους σε κάθε μία από αυτές. Κατ' αυτό τον τρόπο σχηματίζεται μία λίστα των μεθόδων που προκαλούν το σφάλμα που περιγράφει η αναφορά αστοχίας που θέσαμε ως είσοδο.

Η μεθοδολογία αυτή εφαρμόστηκε σε έξι προγράμματα λογισμικού και μπορεί να αποτελέσει τη βάση για την ανάπτυξη ενός πιο εξελιγμένου συστήματος εντοπισμού αστοχιών στον πηγαίο κώδικα, γεγονός που μπορεί να διευκολύνει τους προγραμματιστές καθώς έτσι δε θα χρειάζεται να ξοδέψει ιδιαίτερο χρόνο αναζητώντας τις ελλαττωματικές μεθόδους σε όλη την έκταση του κώδικα, αλλά θα τους δίνεται οι δυνατότητα να παρεμβαίνουν στις μεθόδους που πρέπει να τροποποιηθούν.

Λέξεις Κλειδιά: Σύστημα Εντοπισμού Αστοχιών, Αναφορές Αστοχιών, Αστοχία πηγαίου κώδικα, Λέξεις πηγαίου κώδικα, Τεχνικές Ανάκτησης πληροφορίας.

Abstract

The scope of this thesis was the design and the development of a methodology in order to recognize and localize bugs in source code. The bug localization system gives user the availability to detect which source code methods need to be modified in order to solve a bug in a software. This methodology is based on bug reports which are stored in bug repositories such as Bugzilla.

Specifically, the environment, which has been developed, takes as input a bug report and then creates a list with the most relevant bug reports of the Bugzilla repository for the specific software. As a result developers have a cluster of the most similar to the input bug reports. After that, the bug localization system processes the data and classifies the methods by taking into account the relations between them and the metrics for weight calculation in each of them. Finally, the output of the system is a list of methods that have a high probability to cause a bug in the software.

This methodology has been applied on six software programs and can be used as a foundation for the development of a more advanced bug localization system. The usefulness of the system created is that it can help developers not to spend considerable time searching the entire source code for bugs. Instead it gives them the opportunity to directly detect the methods which should be modified so as to solve the bug.

Keywords: Bug Localization System, Bugzilla, Bug Reports, Source code bugs, Source Code Entities, Recover Information Techniques, Latent Semantic Indexing

Περιεχόμενα

1

Εισαγωγή

1.1 Αξιολόγηση και Εντοπισμός Αστοχιών Λογισμικού

Σε μία κοινωνία όπου η πληροφορία αποτελεί ένα από τα κυριότερα – αν όχι το πιο κύριο- χαρακτηριστικά αυτής, σε μια κοινωνία όπου η αυτοματοποίηση και η απλοποίηση είναι απώτερος σκοπός, σε μία κοινωνία όπου η άμεση επεξεργασία της πληροφορίας καθίσταται αναγκαία, η εύρεση κάποιας διαδικασίας που θα έχει αυτόν τον σκοπό αποτελεί κύριο μέλημα των ανθρώπων.

Στον τομέα της πληροφορικής, η προσπάθεια για αυτοματοποίηση πραγματοποιείται με τα συστήματα λογισμικού (software systems). Τον όρο *σύστημα λογισμικού* τον χρησιμοποιούμε για να περιγράψουμε ένα σύνολο διαδικασιών που έχουν σχεδιαστεί για να λειτουργούν και να ελέγχουν το υλικό του υπολογιστή (computer hardware) με σκοπό να παρέχουν μία πλατφόρμα για την εκτέλεση άλλων εφαρμογών.

Στη σημερινή εποχή λοιπόν πλήθος προγραμματιστών εργάζονται στον τομέα της ανάπτυξης τέτοιων συστημάτων, με αποτέλεσμα οι βιομηχανίες λογισμικού να ανταγωνίζονται μεταξύ τους ως προς την ποιότητα και την αξιοπιστία των προϊόντων τους. Ως *αξιοπιστία* ενός συστήματος, ορίζουμε την πιθανότητα P(t) το σύστημα να μην εμφανίσει κάποια αστοχία (bug) στο διάστημα συγκεκριμένου χρόνου λειτουργίας του t, δεδομένου ότι εκτελείται σε ένα συγκεκριμένο περιβάλλον. Κατά συνέπεια, η ανάγκη δημιουργίας ενός αξιόπιστου συστήματος που στερείται αστοχιών ολοένα και αυξάνεται, αφού ένα τέτοιο σύστημα δίνει στον χρήστη τη δυνατότητα να πραγματοποιήσει την εργασία του δίχως να πρέπει να έρθει αντιμέτωπος με ανεπιθύμητες δυσχέρειες.

Συνεπώς, τόσο η *αξιολόγηση* όσο και η συντήρηση ενός προγράμματος λογισμικού αποτελούν διαδικασίες οι οποίες συνεισφέρουν στη βελτίωση της ποιότητας του συστήματος, αλλά και στην εξέλιξή του. Κάθε λογισμικό αναμένεται να ανταποκριθεί σε συγκεκριμένες ανάγκες. Έτσι όταν αυτό έχει αναπτυχθεί, απαραίτητη προϋπόθεση είναι να ελεγχθεί αν πληρούνται οι προδιαγραφές που έχει θέσει ο χρήστης. Αυτό πραγματοποιείται μέσω της αξιολόγησής του, δηλαδή της διαδικασίας επικύρωσης και επαλήθευσης των λειτουργιών του λογισμικού που κύριο στόχο έχει τον εντοπισμό σφαλμάτων πριν το χρησιμοποιήσει ο χρήστης.

Σε αρκετά από τα συστήματα λογισμικού είναι πιθανό να εντοπιστούν αστοχίες στο στάδιο της αξιολόγησης. Ως *αστοχία* μπορεί να θεωρηθεί οποιαδήποτε διαφοροποίηση της παρατηρούμενης συμπεριφοράς του συστήματος σε σχέση με αυτή που έχει προδιαγραφεί και είναι δυνατό να οφείλεται σε κάποιο *ελάττωμα* (fault) ή *σφάλμα* (error). Το ελάττωμα αναφέρεται κυρίως στον πηγαίο κώδικα του προγράμματος και μπορεί να είναι επικεντρωμένο σε ένα σημείο ή να είναι διάσπαρτο σε πολλά μέρη του κώδικα εξαιτίας της αλληλεπίδρασης που έχει ο κώδικας αυτός με άλλους παράγοντες. Ορισμένες φορές, κάποιο ελάττωμα είναι δυνατό να οδηγήσει στη διαφοροποίηση της εσωτερικής κατάστασης (state) του συστήματος – καθώς αυτό λειτουργεί – από την προδιαγεγραμμένη κατάσταση στην οποία θα έπρεπε να βρίσκεται το σύστημα κατά την λειτουργία του (σφάλμα).

Ο *εντοπισμός αστοχιών* στον πηγαίο κώδικα (bug localization) είναι μία διαδικασία που συμβάλλει και εκείνη με τη σειρά της στην ανάπτυξη και τη συντήρηση λογισμικού. Με τη διαδικασία αυτή, προσδιορίζονται σημεία του κώδικα, που προκαλούν σφάλματα στο πρόγραμμα, τα οποία ο προγραμματιστής θα πρέπει να διορθώσει ώστε να εξαλείψει τις αστοχίες. Άλλωστε το εμπόδιο των αστοχιών είναι ένα μείζων θέμα που καλείται να αντιμετωπίσει ο προγραμματιστής σε όλη την πορεία της εργασίας του, καθώς θα πρέπει να είναι σε θέση να εντοπίζει άμεσα και έγκαιρα τον «ελαττωματικό» κώδικα, να παρεμβαίνει εξίσου άμεσα σε αυτόν και να τον διορθώνει.

Όλα τα παραπάνω, κάνουν επιτακτική την ανάγκη δημιουργίας ενός άλλου, νέου λογισμικού, το οποίο θα μπορεί να προσδιορίζει με αυτόματο τρόπο τα σημεία του πηγαίου κώδικα (source code) που προκαλούν αστοχίες. Ένα τέτοιο σύστημα θα προσφέρει στον προγραμματιστή τη δυνατότητα να επέμβει ακριβώς στο σημείο του κώδικα που χρειάζεται, χωρίς να ξοδέψει ιδιαίτερο χρόνο αναζητώντας σε όλη την έκταση του κώδικα τις εντολές που οδήγησαν σε αστοχία.

Ο χώρος του προγραμματισμού και της αυτοματοποίησης έρχεται και θα έρχεται πάντα αντιμέτωπος με το θέμα του χρόνου και των αποδοτικών σε αυτό τον τομέα συστημάτων. Γι’ αυτό άλλωστε πραγματοποιείται συστηματικά εκτεταμένη έρευνα για την ανάπτυξη αλγόριθμων σχετικών με τον εντοπισμό σφαλμάτων του πηγαίου κώδικα που στόχο θα έχει την διευκόλυνση του προγραμματιστή, αλλά και την σμίκρυνση του χρονικού βεληνεκούς αξιολόγησης και ελέγχου ενός συστήματος.

1.2 Αντικείμενο διπλωματικής

Σκοπός της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός, η ανάλυση και η υλοποίηση ενός συστήματος που θα μπορεί να εντοπίζει τα ελαττώματα του πηγαίου κώδικα στα οποία οφείλονται αστοχίες ενός προγράμματος λογισμικού. Θα πραγματεύεται αυτό που με την αγγλική ορολογία ονομάζουμε bug localization.

Η λειτουργία του συστήματος, το οποίο θα παρουσιαστεί λεπτομερώς σε επόμενα κεφάλαια, στηρίζεται στις αναφορές σφαλμάτων (bug reports). Τα περισσότερα από τα ήδη υλοποιημένα και ευρέως διαδεδομένα προγράμματα διαθέτουν μία πλατφόρμα στην οποία απλοί χρήστες (end-users) ή και άλλοι προγραμματιστές (developers) έχουν τη δυνατότητα να καταχωρήσουν καθώς και να σχολιάσουν σφάλματα που παρατηρούν στο συγκεκριμένο λογισμικό. Ο προγραμματιστής, στον οποίο ανατίθεται κάθε φορά ένα bug report, αναλύει το σφάλμα που προέκυψε και ψάχνοντας στον κώδικα του προγράμματος, βρίσκει και διορθώνει τα ελαττωματικά αρχεία ή τμήματα κώδικα.

Αυτές οι αναφορές σφαλμάτων που συγκεντρώνουμε και επεξεργαζόμαστε είναι γραμμένες σε φυσική γλώσσα. Επομένως μία από τις δυσκολίες της εργασίας αυτής είναι η επεξεργασία της φυσικής γλώσσας (**N**atural **L**anguage **P**rocessing – NLP) που στην προκειμένη περίπτωση είναι τα Αγγλικά.

Για το σκοπό αυτό έχουν δημιουργηθεί πολλά εργαλεία που στόχο έχουν την επεξεργασία της φυσικής γλώσσας όπως είναι το OpenNLP ή το Stanford NLP. Τα εργαλεία αυτά, παρέχουν μεθόδους οι οποίες υποστηρίζουν τις πιο συνηθισμένες διαδικασίες επεξεργασίας γλώσσας, όπως είναι ο χωρισμός κειμένου σε λέξεις (tokenization), σε προτάσεις (sentence segmentation), η αναγνώριση του μέρους του λόγου που ανήκει κάθε όρος που συναντάται στο κείμενο (part of speech tagging) καθώς και η δημιουργία του συντακτικού δέντρου του κειμένου.

Χρησιμοποιώντας τα, το πρόβλημα της επεξεργασίας της φυσικής γλώσσας, θα μπορούσε να ξεπεραστεί σε μεγάλο βαθμό, αλλά όχι να εξαλειφθεί, διότι όπως είναι αναμενόμενο η φυσική γλώσσα περιέχει όχι μόνο κυριολεκτικές, αλλά και μεταφορικές έννοιες ή άλλες ιδιομορφίες που μπορεί να «μπερδεύουν» ένα τέτοιο εργαλείο. Παρ' όλα αυτά όμως, τέτοια εργαλεία δίνουν στον προγραμματιστή τη δυνατότητα να κατανοήσει σε μεγάλο βαθμό το νόημα ενός κειμένου, αλλά και να κρατήσει έννοιες που μπορεί να του φανούν περισσότερο χρήσιμες για την περεταίρω ανάλυση και εργασία του.

Στο παρόν πόνημα όμως δεν χρησιμοποιήθηκε κάποιο από αυτά τα εργαλεία. Παραμένοντας σε ένα αρχικό στάδιο ανάλυσης, οι αναφορές σφαλμάτων και τα σχόλια αυτών υπέστησαν επεξεργασία από την οποία τελικά κρατήσαμε μόνο τις λέξεις που αναφέρονται στον πηγαίο κώδικα (source code entities). Έτσι χρησιμοποιήθηκαν για την εργασία αυτή bug reports τα οποία έκαναν αναφορά σε συγκεκριμένα αντικείμενα του πηγαίου κώδικα του συστήματος που μελετήσαμε.

Το περιβάλλον, που δημιουργήθηκε, στο στάδιο επεξεργασίας των αναφορών σφάλματος, δέχεται ως είσοδο μία αναφορά σφάλματος και στη συνέχεια δημιουργεί μία ταξινομημένη λίστα με τις υπόλοιπες αναφορές του υπό μελέτη συστήματος ώστε να δώσει στον προγραμματιστή μία εικόνα σχετικά με την ομοιότητα του

συγκεκριμένου σφάλματος με προηγούμενα ήδη καταχωρημένα σφάλματα. Θέτοντας ένα όριο στον βαθμό ομοιότητας, η λίστα αυτή μπορεί να περιέχει τα bug reports που εμφανίζονται πιο σχετικά με εκείνο της εισόδου.

Το κατώφλι (threshold) λοιπόν στο βαθμό ομοιότητας είναι ένα άλλο θέμα το οποίο πρέπει να επιλυθεί. Ποιο όριο θεωρείται ικανοποιητικό; Φυσικά αυτό είναι μία μεταβλητή που εξαρτάται από κάθε σύστημα και δε μπορεί να είναι μία τιμή που θα ισχύει σε όλα. Τι συμβαίνει όμως στην περίπτωση που δεν υπάρχει κανένα σχετικό σφάλμα για το όριο που έχει τεθεί; Ή ακόμη και αν ο αριθμός των αναφορών στην ταξινομημένη λίστα είναι αρκετά μικρός; Για να αποφύγουμε τέτοιου είδους περιπτώσεις θεωρήσαμε το κατώφλι του βαθμού ομοιότητας μεταβλητό για κάθε σύστημα, το οποίο μπορεί να οριστεί από τον προγραμματιστή αναλόγως με το αποτέλεσμα που θέλει να πάρει ή ακόμη και για να βελτιώσει κάποιο αποτέλεσμα που ήδη έχει (αυξάνοντας ή μειώνοντας το κατώφλι).

Συγκεντρώνοντας όλες τις λέξεις του κώδικα που υπάρχουν στις σχετικές αναφορές σφαλμάτων και χρησιμοποιώντας τις σχέσεις που τις συνδέουν στον κώδικα, προσδίδουμε σε κάθε μία ένα βάρος, το οποίο προσδιορίζει πόσο συναφές είναι το σημείο αυτό του κώδικα για την επίλυση του σφάλματος. Ποιες σχέσεις όμως είναι οι ιδανικότερες για αυτό το σκοπό; Οι σχέσεις μεταξύ αντικειμένων (συναρτήσεων, μεθόδων, μεταβλητών κλπ.) που ορίζονται από τον κώδικα του προγράμματος, εντοπίζονται με τη βοήθεια ενός εργαλείου επεξεργασίας πηγαίου κώδικα, η λειτουργία του οποίου αναλύεται εκτενώς σε επόμενο κεφάλαιο. Ορισμένες από αυτές είναι ιδιαιτέρως χρήσιμες για την ανάλυσή μας και άλλες δεν προσδίδουν κάποια επιπλέον πληροφορία. Ο διαχωρισμός τους αποτέλεσε ένα σημαντικό θέμα επειδή η χρήση πλεονάζουσας πληροφορίας είναι κάτι που πιθανόν να προσθέτει «θόρυβο» στα δεδομένα μας και να παρεμβαίνει την ακρίβεια του αποτελέσματος.

Και πώς θα υπολογίσουμε το κατάλληλο βάρος για κάθε αντικείμενο του πηγαίου κώδικα; Σε κάθε λέξη του κώδικα που εμφανίζεται στο σύνολο μας και προέρχεται τόσο από τα bug reports όσο και από τις σχέσεις με άλλα αντικείμενα του κώδικα προσδίδεται ένα βάρος, το οποίο σχετίζεται με το κατά πόσον ο όρος αυτός είναι σχετικός με την αναφορά σφάλματος που βάλαμε ως είσοδο στο σύστημά μας. Η ομοιότητα υπολογίζεται ανάμεσα σε κάποια σύνολα που δημιουργούνται, ένα για την αναφορά της εισόδου, και ένα για κάθε όρο που συναντάμε και για να βρούμε το βαθμό της χρησιμοποιήθηκε η μετρική ομοιότητας Jaccard.

Τα παραπάνω είναι και τα κυριότερα από τα προβλήματα που αντιμετωπίστηκαν κατά την εκπόνηση της εργασίας αυτής. Μερικές διαφοροποιήσεις στον τρόπο επίλυσής τους (όπως για παράδειγμα η χρήση των εργαλείων επεξεργασίας της φυσικής γλώσσας) θα μπορούσαν να αποτελέσουν αντικείμενο μελλοντικής εργασίας και έρευνας.

1.3 Οργάνωση κειμένου

Η υπόλοιπη εργασία οργανώνεται στα έξι κεφάλαια που ακολουθούν.

Στο δεύτερο κεφάλαιο παρουσιάζονται προηγούμενες έρευνες και εργασίες σχετικές με το αντικείμενο της παρούσας εργασίας.

Το Κεφάλαιο 3 περιλαμβάνει τεχνικές, μεθοδολογίες και μοντέλα στα οποία βασίστηκε η σχεδίαση του περιβάλλοντος ανάλυσης των αναφορών αστοχιών το οποίο παρουσιάζεται στην εργασία αυτή.

Στο Κεφάλαιο 4 αναπτύσσεται και επεξηγείται η έννοια «αναφορά αστοχιών» καθώς και η πλατφόρμα από την οποία συλλέξαμε τις αναφορές αυτές (Bugzilla). Επιπλέον παρουσιάζεται και αναλύεται το μοντέλο μίας τέτοιας αναφοράς αστοχιών.

Συνεχίζοντας, στο επόμενο κεφάλαιο (Κεφάλαιο 5) περιγράφεται αναλυτικά και με λεπτομέρεια το περιβάλλον που δημιουργήθηκε. Αναφέρονται όλα τα στάδια του και εξηγείται η διαδικασία που ακολουθείται από την είσοδό του, μέχρι και την παρουσίαση των αποτελεσμάτων.

Στο Κεφάλαιο 6 παρουσιάζονται τα πειραματικά αποτελέσματα. Το περιβάλλον, το οποίο δημιουργήθηκε, χρησιμοποιήθηκε για τον εντοπισμό μεθόδων που είναι πιθανό να χρειάζονται τροποποίηση προκειμένου να επιλυθεί κάποια αστοχία για έξι εφαρμογές λογισμικού. Στο κεφάλαιο αυτό αναλύονται τα αποτελέσματα που πήραμε και αξιολογείται το σύστημα το οποίο δημιουργήθηκε.

Τέλος, στο Κεφάλαιο 7, θα αναφερθούμε σε συμπεράσματα στα οποία καταλήξαμε κατά τη διάρκεια της εργασίας αυτής καθώς και σε ζητήματα τα οποία αποτελέσουν αντικείμενο περαιτέρω διερεύνησης.

2

Σχετικές εργασίες

Στο κεφάλαιο αυτό θα αναφερθούμε στις θεματικές ενότητες στις οποίες υπάρχουν εργασίες σχετικές με το αντικείμενο της παρούσας διπλωματικής εργασίας. Αρκετοί ερευνητές έχουν αφιερώσει ώρες εργασίας στον τομέα εντοπισμού όχι μόνο αστοχιών, αλλά και γενικότερα εννοιών στον πηγαίο κώδικα. Οι παράγραφοι που ακολουθούν περιγράφουν τον τρόπο εργασίας τους χρησιμοποιώντας διάφορες τεχνικές.

2.1 Concept Location με Latent Semantic Indexing

Η διαδικασία της εξέλιξης αλλά και συντήρησης ενός προγράμματος λογισμικού, απαιτεί επαναλαμβανόμενες αλλαγές στον κώδικά του, οι οποίες είτε είναι αποτέλεσμα προδιαγραφών που έχει θέσει ο χρήστης, είτε πραγματοποιούνται για την βελτίωση και τη διόρθωση τυχόν αστοχιών που παρατηρούνται στο σύστημα. Ο προγραμματιστής που θα κληθεί να τροποποιήσει τον κώδικα όμως, και στις δύο περιπτώσεις, θα πρέπει να εντοπίσει το σημείο εκείνο στο οποίο αναφέρεται η προδιαγραφή ή η αναφορά σφάλματος ώστε να κάνει τις απαραίτητες αλλαγές. Ο χρόνος που χρειάζεται μία τέτοια διαδικασία εξαρτάται από αρκετές παραμέτρους, όπως είναι η γνώση και η κατανόηση που έχει προγραμματιστής για το συγκεκριμένο λογισμικό καθώς και η πολυπλοκότητα του συστήματος.

Έτσι καταλήγουμε στον όρο «εντοπισμός έννοιας» ή όπως λέγεται στα Αγγλικά *concept location* με τον οποίο περιγράφουμε τον εντοπισμό σημείων του κώδικα στα οποία αναφέρεται μία έννοια ή μία περιγραφή.

Παρ' όλο που εκ πρώτης όψεως μία μελέτη σαν και αυτή δεν φαίνεται σχετική με τον εντοπισμό αστοχιών, τον οποίο πραγματεύεται το παρόν πόνημα, στην πραγματικότητα στηρίζεται στην ίδια λογική. Το πρόβλημα του bug localization είναι ο εντοπισμός τμημάτων κώδικα τα οποία προκαλούν σφάλματα στο λογισμικό. Για να πραγματοποιηθεί κάτι τέτοιο με αυτοματοποιημένο τρόπο, στηριζόμαστε στις αναφορές αστοχιών, οι οποίες περιγράφουν στη φυσική γλώσσα τη δυσλειτουργία που παρουσιάστηκε. Παρόμοια διαδικασία πραγματοποιείται και για το πρόβλημα του concept location.

Οι A. Marcus, A. Sergeyev, V. Rajlich και J. Maletic πραγματοποίησαν μία έρευνα στον τομέα του concept location χρησιμοποιώντας μία τεχνική ανάκτησης πληροφορίας, το Latent Semantic Indexing (LSI). Οι ίδιοι επιστήμονες σε προηγούμενη δουλειά τους, προσπάθησαν με τη βοήθεια του LSI να εντοπίσουν συνδέσεις μεταξύ κειμένων τεκμηρίωσης κώδικα (documentation) και αρχείων του πηγαίου κώδικα. Η σημαντική διαφορά σε αυτή την έρευνα είναι ότι το LSI θα χρησιμοποιηθεί για την αντιστοίχιση ερωτημάτων του χρήστη με στοιχεία του κώδικα, δηλαδή ο χρήστης θα κατέχει τον κυρίαρχο ρόλο στην μελέτη, αφού θα είναι εκείνος που θα δημιουργεί τα ερωτήματα και θα αξιολογεί τα αποτελέσματά τους. Τέλος, τα αποτελέσματα αυτά θα συγκριθούν και με στατικές μεθόδους όπως η παραδοσιακή μέθοδος αναζήτησης *grep* καθώς και η αναζήτηση στον γράφο εξαρτήσεων του κώδικα (dependency graph).

Όπως είναι φυσικό, οι έννοιες υπάρχουν «κρυμμένες» στον πηγαίο κώδικα ενός προγράμματος μέσω των ονομάτων των αναγνωριστικών (identifiers) και των σχολίων του κώδικα (comments). Τα στοιχεία αυτά θα χρησιμοποιηθούν στην ανάλυση με το LSI, αφού προηγουμένως ο κώδικας υποστεί μία προ-επεξεργασία. Επομένως, τρία είναι τα βασικά βήματα που πραγματοποιήθηκαν σε αυτή την έρευνα:

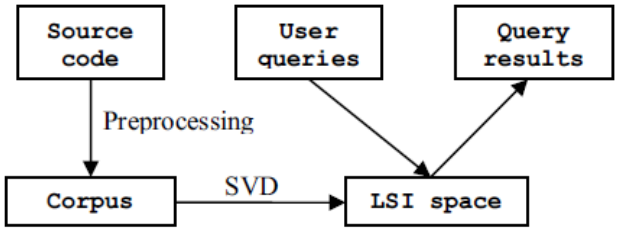
1) *Εξαγωγή αναγνωριστικών και σχολίων από τον πηγαίο κώδικα:*
   Η διαδικασία αυτή απαιτεί πολύ προσεκτική ανάλυση του κώδικα, αλλά εξαιτίας των ομοιοτήτων που εμφανίζουν πολλές από τις γλώσσες προγραμματισμού, οι ερευνητές έφτιαξαν ένα πρόγραμμα που αυτοματοποιεί τη διαδικασία αυτή και είναι εφαρμόσιμο σε πηγαίο κώδικα που είναι γραμμένος σε γλώσσες όπως C, C++ και Java.

2) *Διαχωρισμός των αναγνωριστικών σε όρους με νόημα:*
   Το βήμα αυτό συνεισφέρει στην ενίσχυση των εννοιών που θα υπάρχουν στη συλλογή εγγράφων και ταυτόχρονα βελτιώνει τα τελικά αποτελέσματα. Παρατηρήθηκε ότι οι συνηθέστεροι κανόνες για την ονομασία ενός αναγνωριστικού, όταν στο όνομα περιέχονται περισσότερες από μία σημασιολογικές λέξεις, είναι δύο: διαχωρίζοντας τις λέξεις με κάτω παύλα (underscore "_") όπως για παράδειγμα concept_location ή χρησιμοποιώντας κεφαλαία γράμματα (πχ ConceptLocation ή CONCEPTLocation). Με βάση τα παραπάνω τα αναγνωριστικά χωρίστηκαν στις λέξεις που τα αποτελούν και τόσο αυτές όσο και το αρχικό όνομα του αναγνωριστικού προστέθηκαν στο σώμα των όρων τους οποίους θα επεξεργαστεί το LSI στη συνέχεια.

3) *Δημιουργία συλλογής εγγράφων*
   Στο στάδιο αυτό, που είναι και το τελευταίο στάδιο της προ-επεξεργασίας του πηγαίου κώδικα, δημιουργούνται τα έγγραφα (documents) που είναι

απαραίτητα για την εφαρμογή του LSI. Κάθε συνάρτηση επιλέγεται να είναι ξεχωριστό έγγραφο όπως επίσης και το μπλοκ με τις δηλώσεις μεταβλητών έξω από αυτήν.

Στο σημείο αυτό λοιπόν ολόκληρο το σύστημα λογισμικού, το οποίο αναλύεται, έχει αποσυντεθεί σε ένα σύνολο από έγγραφα. Τυπικά λοιπόν κάθε έγγραφο είναι μία συνάρτηση ή μία κλάση του συστήματος. Έτσι, ο πηγαίος κώδικας (source code) μετατρέπεται στο σώμα (corpus) στο οποίο θα εφαρμοστεί το LSI και θα προκύψει το LSI space. Σε αυτό τον χώρο, κάθε έγγραφο αντιπροσωπεύεται από ένα διάνυσμα που θα χρησιμοποιηθεί για τον υπολογισμό του βαθμού ομοιότητας μεταξύ τους. Όλα τα παραπάνω φαίνονται στην εικόνα που ακολουθεί:



Εικόνα 2.1: Η διαδικασία του concept location χρησιμοποιώντας το LSI

4) *Δημιουργία ερωτημάτων αναζήτησης*

Μέχρι στιγμής δεν έχουμε αναφερθεί καθόλου στα ερωτήματα αναζήτησης (User queries). Αυτά μπορούν να πραγματοποιηθούν με δύο τρόπους. Ο ένας είναι να θέσει ο χρήστης απευθείας το ερώτημα στο σύστημα και ο άλλος είναι να δημιουργηθούν αυτοματοποιημένα ερωτήματα αναζήτησης. Τα ερωτήματα μετατρέπονται και αυτά στα αντιπροσωπευτικά τους διανύσματα, τα οποία συγκρίνονται με τα έγγραφα του σώματος (corpus) και έτσι εντοπίζονται αυτά που σχετίζονται με το ερώτημα. Το σύστημα επιστρέφει ένα σύνολο από όλα τα έγγραφα της συλλογής, ταξινομημένα με βάση το βαθμό ομοιότητάς τους με το ερώτημα. Έτσι ο χρήστης μπορεί να διαπιστώσει ποια από αυτά ταιριάζουν στην έννοια που αποδόθηκε στο ερώτημα αναζήτησης.

Το πρόγραμμα λογισμικού στο οποίο εφαρμόστηκε η μελέτη είναι ο NCSA Mosaic browser και τα συμπεράσματα που προέκυψαν είναι τα εξής:

- Η μέθοδος LSI παρουσιάζει πολλά προτερήματα στην εν λόγω μελέτη. Συγκριτικά με το grep είναι το ίδιο απλή και ευέλικτη, ενώ τα αποτελέσματά της είναι αρκετά καλύτερα.
- Επιπλέον, με το LSI εντοπίστηκαν και τμήματα του κώδικα, τα οποία δεν ήταν δυνατό να τα εντοπίσουν με την προσέγγιση του γράφου εξαρτήσεων.
- Το σημαντικότερο προτέρημα του LSI είναι ότι είναι ανεξάρτητο από την γλώσσα προγραμματισμού και η επεξεργασία του πηγαίου κώδικα ήταν πολύ πιο απλή διαδικασία από τη δημιουργία ενός γράφου εξαρτήσεων.

- Τέλος, έγινε εμφανές ότι το LSI μπορούσε να αναγνωρίσει λέξεις και αναγνωριστικά από τον πηγαίο κώδικα που σχετίζονται με τους όρους και τις φράσεις που τέθηκαν από τον χρήστη ως ερωτήματα αναζήτησης.

Παρόμοια μελέτη έχει πραγματοποιηθεί από τους D. Poshyvanyk, Y.Gueheneuc, A. Marcus, G. Antoniol και V. Rajlich για τον εντοπισμό χαρακτηριστικών του κώδικα χρησιμοποιώντας ταξινόμηση με πιθανοτικό υπολογισμού του σκορ και μεθόδους ανάκτησης πληροφορίας.

2.2 Εντοπισμός ελαττωμάτων με Latent Diriclet Allocation

Στην ενότητα αυτή θα περιγράψουμε μία στατική τεχνική για την αυτοματοποίηση του εντοπισμού ελαττωμάτων πηγαίου κώδικα (bug localization) η οποία βασίζεται σε ένα άλλο μοντέλο ανάκτησης πληροφοριών, το Latent Dirichlet Allocation (LDA).

Η έρευνα αύτη πραγματοποιήθηκε από τους S.K. Lukins, N.A. Kraft και L.H. Etzkorn και στηρίζεται στην ιδέα ότι ακόμη και ο ίδιος ο κώδικας ενός προγράμματος περιλαμβάνει σημασιολογικές έννοιες οι οποίες βρίσκονται «κρυμμένες» τόσο στα σχόλια, όσο και στα ονόματα των μεταβλητών ή άλλων αναγνωριστικών του.

Το σύστημα που δημιουργήθηκε και έχει ως στόχο το bug localization αποτελείται από δύο κύρια μοντέλα: το μοντέλο LDA και το μοντέλο των Queries. Δύο στάδια είναι απαραίτητα για την κατασκευή του LDA μοντέλου του συστήματος λογισμικού που εξετάζεται:

1) *Κατασκευή μίας συλλογής εγγράφων* από τον πηγαίο κώδικα:
   Για την δημιουργία της συλλογής εγγράφων είναι σημαντικό να διαχωριστεί η σημασιολογική πληροφορία, όπως τα σχόλια και τα αναγνωριστικά του κώδικα στον επιθυμητό βαθμό (μέθοδος, κλάση κλπ.). Στη συγκεκριμένη μελέτη, κάθε έγγραφο της συλλογής απαρτίζεται από τα δεδομένα που εξάγονται από κάθε μέθοδο του πηγαίου κώδικα, αφού πρώτα έχουν υποστεί κάποια επεξεργασία (stemming, remove stop words). Τα αναγνωριστικά που περιέχουν περισσότερες από μία λέξεις σε έναν όρο, όπως για παράδειγμα printFile, διαχωρίζονται στις λέξεις που τα απαρτίζουν χρησιμοποιώντας εργαλεία που βασίζουν τη λειτουργία τους στις κλασσικές τεχνικές κωδικοποίησης (παραδείγματος χάριν κάθε λέξη ξεκινά με κεφαλαίο γράμμα, όπως το File στο printFile).

2) *Εφαρμογή της LDA ανάλυσης* στη συλλογή εγγράφων που δημιουργήθηκε:
   Στο στάδιο αυτό χρησιμοποιήθηκε ένα εργαλείο για LDA ανάλυση που χρησιμοποιεί τη δειγματοληψία Gibbs για να εκτιμήσει τα θέματα (topics) τα οποία προκύπτουν από όλα τα έγγραφα που δημιουργήθηκαν στο προηγούμενο στάδιο. Το εργαλείο αυτό ονομάζεται GibbsLDA++. Αφού οριστούν κάποιες παράμετροι, όπως είναι το πλήθος των θεμάτων που θα υπολογιστεί από το LDA, το πλήθος των επαναλήψεων του αλγόριθμου και δύο παράγοντες εξομάλυνσης, εφαρμόζεται η μέθοδος LDA στη συλλογή των εγγράφων. Το αποτέλεσμα της μεθόδου LDA είναι δύο κατανομές πιθανότητας: μία σχετική με το θέμα των όρων (word-topic probability distribution) και μία σχετική με το θέμα των εγγράφων (document-topic probability distribution). Επιπλέον, το εργαλείο GibbsLDA++ δίνει και μία λίστα από θέματα με τις n λέξεις που έχουν την μεγαλύτερη πιθανότητα να ανήκουν σε κάθε θέμα.

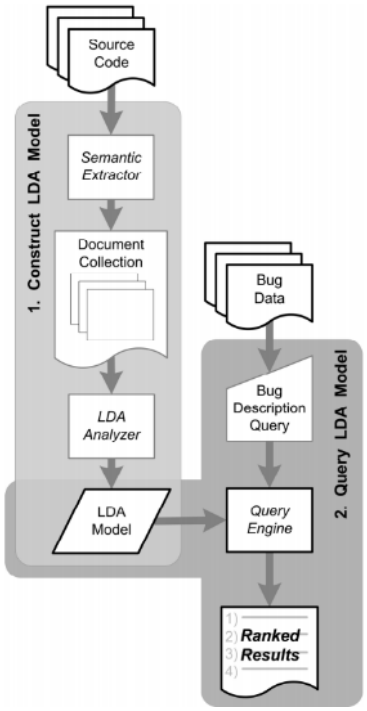
Έτσι στο σημείο αυτό έχει δημιουργηθεί το μοντέλο LDA του κώδικα, στο οποίο θα γίνονται τα ερωτήματα αναζήτησης (queries) ώστε να εντοπιστούν τα σφάλματα (bugs). Οι όροι του ερωτήματος προ-επεξεργάζονται με τις ίδιες μεθόδους που χρησιμοποιήθηκαν για την επεξεργασία του πηγαίου κώδικα. Κάθε ερώτημα

λοιπόν αντιπροσωπεύεται από μία λίστα με λέξεις του κώδικα οι οποίες είναι ταξινομημένες με βάση το βαθμό ομοιότητας με τα έγγραφα του LDA μοντέλου.

Όσο αφορά στο μοντέλο ερωτημάτων, τα queries δημιουργήθηκαν χειροκίνητα αξιοποιώντας τις πληροφορίες του τίτλου και της περιγραφής σφαλμάτων που υπάρχουν στις αναφορές σφαλμάτων (bug reports).

Το αποτέλεσμα είναι μία ακόμη ταξινομημένη λίστα με τα έγγραφα της συλλογής. Η θέση κάθε εγγράφου στη λίστα αυτή αντιστοιχεί στο βαθμό ομοιότητας του εγγράφου με το ερώτημα αναζήτησης που τέθηκε στο σύστημα.

Όσα περιγράψαμε παραπάνω, απεικονίζονται σχηματικά στην εικόνα που ακολουθεί:



Σχήμα 2.2: LDA προσέγγιση εντοπισμού σφαλμάτων πηγαίου κώδικα

Για να καθοριστεί η αποδοτικότητα αλλά και η ακρίβεια της LDA προσέγγισης για τον εντοπισμό των ελαττωμάτων του πηγαίου κώδικα που οδηγούν σε αστοχίες, οι ερευνητές πραγματοποίησαν τέσσερις μελέτες στα συστήματα: Mozilla, Rhino και Eclipse. Στην πρώτη μελέτη, εξέτασαν αν η ακρίβεια του LDA μοντέλου υπερτερεί σε σχέση με του LSI χρησιμοποιώντας τα ίδια δεδομένα (data set) που είχαν χρησιμοποιηθεί και σε προγενέστερες έρευνες σχετικές με το LSI για το Mozilla και το Eclipse. Ο στόχος σε αυτή την περίπτωση ήταν να μελετηθεί και να συγκριθεί το στατικό μοντέλο LDA μόνο με το στατικό μοντέλο LSI και όχι με δυναμικά μοντέλα ή με συνδυασμούς μοντέλων. Στο δεύτερο μέρος της μελέτης τους επικεντρώθηκαν στην εφαρμογή της LDA προσέγγισης σε όλα τα bugs ενός συστήματος και όχι σε ένα συγκεκριμένο σύνολό τους, όπως στην προηγούμενη περίπτωση. Με αυτό τον

τρόπο θα μπορούσαν να αξιολογήσουν την απόδοση και την ακρίβεια του συστήματος που δημιούργησαν όταν εκείνο αλληλεπιδρά με όλες τις αναφορές σφαλμάτων που διαθέτει το λογισμικό, το οποίο εξετάζεται. Η τρίτη περίπτωση που μελετήθηκε είναι η εφαρμογή της μεθόδου σε ακόμη μεγαλύτερο σετ δεδομένων (bugs) και για πολλές επαναλήψεις. Η μελέτη αυτή παρ' όλο που χρησιμοποιεί περισσότερα bug reports από την δεύτερη περίπτωση, δεν περιλαμβάνει όλα τα bugs κάθε έκδοσης του λογισμικού, αλλά μας δίνει μία αίσθηση της επεκτασιμότητας (scalability) της μεθόδου και μπορούμε να πούμε ότι είναι συμπληρωματική της δεύτερης περίπτωσης. Στο τελευταίο μέρος της έρευνας αυτής, εξετάστηκε ο εντοπισμός αστοχιών με διάφορες μεταβλητές όπως το μέγεθος του προγράμματος, η πολυπλοκότητα του πηγαίου κώδικα κλπ.

Τα συμπεράσματα στα οποία κατέληξε η έρευνα αυτή είναι ότι το LDA μοντέλο μπορεί να εφαρμοστεί για τον εντοπισμό αστοχιών και επιπλέον η μελέτη έδειξε ότι είναι τουλάχιστον ίδιας απόδοσης ή και πιο αποτελεσματικό σε ορισμένες περιπτώσεις από το αντίστοιχο LSI μοντέλο.

2.3 Εντοπισμός ελαττωμάτων συγκρίνοντας μεθόδους IR

Παρ' όλο που τα εργαλεία εντοπισμού αστοχιών, δυναμικά και μη, μπορούν να χρησιμοποιηθούν σαν διαγνωστικά εργαλεία για το λογισμικό που εξετάζεται, εκείνα που στηρίζονται σε μεθόδους ανάκτησης πληροφορίας (**I**nformation **R**etrieval –IR) αποτελούν και μέσο της διαδικασίας επιδιόρθωσης σφαλμάτων που παρουσιάζει το λογισμικό (debugging process). Στις μεθόδους αυτές, οι αναφορές σφαλμάτων (bug reports) σε συνδυασμό με τον τίτλο και την περιγραφή τους, αποτελούν οντότητες οι οποίες χρησιμοποιούνται ως ερώτημα με σκοπό την εύρεση άλλων σχετικών με αυτές αναφορών.

Στηριζόμενοι στην παραπάνω λογική, αρκετοί ερευνητές εκτός από την εφαρμογή μίας μόνο μεθόδου για τον εντοπισμό αστοχιών στον κώδικα, μελέτησαν και σύγκριναν την απόδοση πολλών μεθόδων ανάκτησης πληροφορίας. Μία από τις έρευνες αυτές, που περιλαμβάνει τη σύγκριση περισσότερων μοντέλων, παρουσιάζεται εκτενώς παρακάτω.

Οι S. Rao και A. Kak σύγκριναν πέντε βασικά IR μοντέλα για να αποφανθούν ποιο είναι το καταλληλότερο για το bug localization. Τα πέντε αυτά μοντέλα είναι το Vector Space Model, το Latent Semantic Indexing Model, το Unigram Model, το Latent Dirichlet Allocation και το Cluster-Based Document Model (CBDM). Από αυτά, τα δύο πρώτα (VSM, LSI) είναι καθαρά ντετερμινιστικά μοντέλα, ενώ τα επόμενα δύο (UM, LDA) είναι πιθανοτικά. Το τρίτο μοντέλο περιλαμβάνει τόσο ντετερμινιστική όσο και πιθανολογική θεώρηση. Για την επαλήθευση των μεθόδων τους χρησιμοποίησαν μία συλλογή από αναφορές σφαλμάτων –που ονομάζεται iBugs – η οποία μπορεί να χρησιμοποιηθεί και με τις δύο κατηγορίες μοντέλων της συγκεκριμένης έρευνας.

Η διαδικασία, που ακολουθήθηκε για την αξιολόγηση της μεθόδου εντοπισμού αστοχιών βασισμένης στην ανάκτηση πληροφοριών, απαιτεί οι αναφορές σφαλμάτων (bug reports) να περιέχουν περιγραφές κειμένου έτσι ώστε να μπορούν να χρησιμοποιηθούν σαν ερωτήσεις αναζήτησης (queries).

Η βασική ιδέα της μελέτης τους στηρίζεται στη δημιουργία μίας συλλογής εγγράφων που προκύπτει από τον πηγαίο κώδικα του συστήματος που αναλύεται (στη συγκεκριμένη περίπτωση το AspectJ). Έπειτα ως ερώτημα αναζήτησης θεωρείται μία αναφορά σφάλματος, η οποία αντιστοιχίζεται με τα παραπάνω έγγραφα αναλόγως με το βαθμό ομοιότητας που παρουσιάζει με τα αντικείμενα του πηγαίου κώδικα.

Προτού να εφαρμοστεί στο ερώτημα αναζήτησης κάθε μία από τις παραπάνω τεχνικές IR, ο πηγαίος κώδικας θα υποστεί μία επεξεργασία ώστε να μπορεί να χρησιμοποιηθεί στο περιβάλλον της ανάκτησης πληροφορίας. Οι πιο συχνές μέθοδοι επεξεργασίας είναι η εξάλειψη των λέξεων της γλώσσας προγραμματισμού (keywords) και ο διαχωρισμός των αναγνωριστικών που περιλαμβάνουν περισσότερες από μία λέξεις. Έπειτα, από τις λέξεις που μένουν και με βάση τα μοντέλα IR, δημιουργούνται τα έγγραφα της συλλογής.

Το αποτέλεσμα – η έξοδος – του συστήματος είναι η επιλογή των N_r εγγράφων της συλλογής, τα οποία εμφανίζουν μεγαλύτερη ομοιότητα με το ερώτημα.

Όπως αναφέρθηκε προηγουμένως, σε αυτή τη μελέτη συγκρίθηκαν τα πιο διαδεδομένα και χρησιμοποιημένα μοντέλα IR. Τα αποτελέσματα που προέκυψαν από τη σύγκριση αυτή, συνοψίζονται παρακάτω:

- Όσο αφορά σε μεγάλα συστήματα λογισμικού, τα μοντέλα όπως το LSI, LDA και CBDM δεν έχουν καλύτερες επιδόσεις από απλούστερα μοντέλα όπως το Unigram και το VSM.
- Η χρήση μετρικών όπως το tf-idf για την κατανομή των λέξεων του πηγαίου κώδικα, δίνει μία καλύτερη αίσθηση της χρησιμότητας των μοντέλων αυτών για το πρόβλημα του εντοπισμού αστοχιών.

Αντίστοιχη μελέτη πραγματοποιήθηκε από τους S. W. Thomas, M. Nagappan, D. Blostein και A. Hassan για τον εντοπισμό αστοχιών διάφορων συστημάτων λογισμικού (Mozilla, Rhino, Jazz, Eclipse, AspectJ και ArgoUML) χρησιμοποιώντας μεθόδους ανάκτησης πληροφορίας. Η διαφοροποίηση της μελέτης αυτής με προηγούμενες είναι ότι ορίζονται με ακρίβεια όλες οι παράμετροι που καθορίζουν τη συμπεριφορά των μοντέλων IR όπως για παράδειγμα ποια τμήματα του κώδικα προ-επεξεργάστηκαν, με ποιο τρόπο αποδίδεται βάρος στους όρους και τι είδους μετρική ομοιότητας χρησιμοποιήθηκε μεταξύ των αναφορών σφαλμάτων και του πηγαίου κώδικα.

Γνωρίζοντας τις αλληλοσυγκρουόμενες απόψεις σχετικά με τα μοντέλα IR και τις επιδόσεις τους, αποφάσισαν χρησιμοποιώντας ίδιο σύνολο δεδομένων και τις ίδιες παραμέτρους να μελετήσουν την επίδραση που έχει η αλλαγή κάθε παραμέτρου στην επίδοση των μεθόδων ξεχωριστά.

Τα μοντέλα IR που συγκρίθηκαν σε αυτή την έρευνα είναι τα VSM (Vector Space Model), LSI (Latent Semantic Indexing) και LDA (Latent Dirichlet Allocation). Τα βήματα που ακολουθήθηκαν σε γενικές γραμμές είναι:

1) Ο ορισμός όλων των παραμέτρων που χρησιμοποιήθηκαν
2) Εκτέλεση κάθε μιας περίπτωσης που προκύπτει και αξιολόγηση της επίδοσής της.
3) Ανάλυση κάθε απόδοσης χρησιμοποιώντας κάποια στατιστικά τεστ (Tukey’s HSD).

Τα συστήματα λογισμικού που εξετάστηκαν επιλέχθηκαν εξαιτίας του μεγάλου μεγέθους τους και του γεγονότος ότι ήταν πραγματικά συστήματα και επομένως επέτρεπαν μία ρεαλιστική σύγκριση των μεθόδων και αξιολόγηση των αποτελεσμάτων. Άλλος ένας σημαντικός λόγος για την επιλογή τους είναι ότι περιέχουν βάσεις δεδομένων από τις οποίες μπορούμε να πάρουμε τις αναφορές σφαλμάτων τους καθώς και ότι ο πηγαίος κώδικάς του είναι διαθέσιμος σε όλες τις εκδόσεις.

Το αρχικό βήμα για τη δημιουργία του πλαισίου εργασίας ήταν μία πρώτη σύνδεση των αναφορών αστοχιών (bug reports) με σημεία του κώδικα. Οι ερευνητές επεξεργάστηκαν και ανέλυσαν σχόλια τόσο του κώδικα όσο και εκείνα που υπάρχουν στο repository τη στιγμή κάποιας αλλαγής (commit logs). Σκοπός τους ήταν να εντοπίσουν μηνύματα όπως “Fixed Bug #4322” ή κάποια παρόμοια ώστε να δημιουργήσουν μία σύνδεση μεταξύ οντοτήτων του κώδικα και αναφοράς

σφάλματος. Το αποτέλεσμα ήταν ένα σύνολο με εξαρτήσεις ανάμεσα σε bug reports και πηγαίο κώδικα, το οποίο χρησιμοποίησαν για να αξιολογήσουν τους ταξινομητές που κατασκεύασαν.

Στη συνέχεια ήταν απαραίτητη η προ-επεξεργασία του κώδικα σε επίπεδο αρχείων. Η διαδικασία αυτή είναι παρόμοια με τις άλλες που περιγράφηκαν σε αυτό το κεφάλαιο και συνοπτικά περιλαμβάνει διαχωρισμό των λέξεων, εξάλειψη των keywords και stop word.

Τέλος, σειρά έχει η επεξεργασία των αναφορών αστοχιών. Αφού συγκεντρώθηκαν όλες οι αναφορές για καθένα από τα συστήματα λογισμικού που μελετήθηκαν, αφαιρέθηκαν εκείνες που:

I. Δεν είχαν κατάσταση (status) “FIXED”.
II. Η επίλυσή τους δεν απαιτούσε τροποποίηση τουλάχιστον ενός σημείου του πηγαίου κώδικα.
III. Δεν είχαν κάποιο τίτλο.
IV. Συνδέονταν με configuration αρχεία (διότι επιθυμούσαν μόνο σύνδεση με οντότητες του κώδικα και όχι με κάτι άλλο)

Όσες παρέμειναν μετά την εφαρμογή των παραπάνω κανόνων τις επεξεργάστηκαν χρησιμοποιώντας τις παραμέτρους κάθε ταξινομητή. Η ίδια επεξεργασία πραγματοποιήθηκε και στον πηγαίο κώδικα.

Εκτός από την σύγκριση των ταξινομητών (classifiers) που δημιούργησαν, μελέτησαν και την επίδοση του συνδυασμού αυτών στο πρόβλημα του bug localization. Τελικά, κατέληξαν στα επόμενα:

- Ο ακριβής καθορισμός των παραμέτρων των ταξινομητών είναι πολύ σημαντικός.
- Καλύτερη μέθοδος αποδείχτηκε εκείνη του μοντέλου VSM, χρησιμοποιώντας για απόδοση βαρών στους όρους το tf-idf, που εφαρμόστηκε σε όλα τα τμήματα του κώδικα (σχόλια, αναγνωριστικά), τα οποία είχαν υποστεί μία προεπεξεργασία παρόμοια με εκείνη των άλλων ερευνών και υπολογίζοντας το βαθμό ομοιότητας με την μετρική του συνημίτονου.
- Ο συνδυασμός των classifiers σε αρκετές περιπτώσεις συντέλεσε στη βελτίωση των αποτελεσμάτων ανεξάρτητα από τις τεχνικές που συνδυάστηκαν.

3

Θεωρητικό υπόβαθρο

Σε αυτή την ενότητα περιλαμβάνονται τεχνικές, μεθοδολογίες αλλά και μοντέλα που χρησιμοποιήθηκαν για την εκπόνηση της διπλωματικής εργασίας και τα οποία χρειάζεται να αναλυθούν πριν τη λεπτομερή παρουσίαση του συστήματος. Τα βασικότερα από αυτά είναι το Fact Extraction Chain Tool, η μέθοδος LSI, το Bugzilla Metamodel, το RDF Schema και κάποιες μαθηματικές έννοιες απαραίτητες για την κατανόηση των παραπάνω.

3.1 Ανάλυση Πηγαίου Κώδικα

3.1.1 Ορισμός του Fact Extraction

Τα συστήματα λογισμικού (software systems) στην σύγχρονη εποχή, συνεχώς αναπτύσσονται και διευρύνονται με ταχείς ρυθμούς. Αυτό έχει ως αποτέλεσμα ο πηγαίος κώδικάς (source code) τους να μεταβάλλεται και πιθανόν να γίνεται πολυπλοκότερος. Έτσι λοιπόν, δημιουργείται η ανάγκη κατανόησης των σχέσεων που συνδέουν όλα τα διαφορετικά μέρη ενός μεγάλου συστήματος.

Χρησιμοποιούμε τον όρο Fact Extraction για να περιγράψουμε τη διαδικασία ανάλυσης κώδικα από την οποία εξάγονται πληροφορίες στοιχείων που ανήκουν σε αντικείμενα λογισμικού. Η διαδικασία αυτή είναι αυτοματοποιημένη και με χρήση εργαλείων ανάλυσης (analyzer tools) πραγματοποιείται επεξεργασία κάθε αρχείου του υπό ανάλυση συστήματος έτσι ώστε να αναγνωριστούν διάφορα χαρακτηριστικά του κώδικα καθώς και οι σχέσεις που συνδέουν τα στοιχεία του συστήματος μεταξύ τους.

Στον πίνακα που ακολουθεί παρουσιάζονται και συγκρίνονται ορισμένα εργαλεία ανάλυσης πηγαίου κώδικα.

Εργαλεία Ανάλυσης Πηγαίου Κώδικα	Metric Calculation	Pattern Detection	Dependency Analysis	Visualization	Open Source
Open Visualization Toolkit	✓	×	×	✓	×
TkSee/SN	✓	×	✓	×	✓
SN/Rigi	✓	×	✓	✓	Μερικώς
CPPX	✓	✓	✓	✓	✓
FETCH	✓	✓	✓	✓	✓

Πίνακας 3.1: Σύγκριση των εργαλείων ανάλυσης πηγαίου κώδικα

Το *Open Visualization Toolkit* συνδυάζει τον Source Navigator (SN) με το Open Inventor C++ toolkit και χρησιμοποιείται κυρίως για αντίστροφη αρχιτεκτονική (reverse architecture) μεγάλων συστημάτων. Επόμενο στη σειρά είναι το *TKSee/SN* το οποίο επίσης χρησιμοποιεί τον Source Navigator (SN) για να δημιουργήσει μια GXL αναπαράσταση του Dagstuhl Middle Model (DMM). Το TKSee/SN υποστηρίζει τόσο πλοήγηση όσο και ερωτήματα αναζήτησης. Το *SN/Rigi* είναι ένα εργαλείο ανάλυσης C++ κώδικα που χρησιμοποιεί τον Source Navigator και το μοντέλο εξόδου παρουσιάζεται σε Rigi Standard Format (RSF). Τέλος, το εργαλείο CPPX εξάγει και εκείνο δεδομένα από C++ κώδικα από το υψηλότερο επίπεδο (classes, functions) μέχρι και το χαμηλότερο (ανεξάρτητες δηλώσεις και εκφράσεις). Ως έξοδο δίνει μία αναπαράσταση του Datrix σε GXL format.

Από τα παραπάνω, στη συγκεκριμένη εργασία χρησιμοποιήθηκε το **F**etch **E**xtraction **T**ool **C**hain (FETCH) του οποίου η λειτουργία παρουσιάζεται αναλυτικά στην παράγραφο που ακολουθεί.

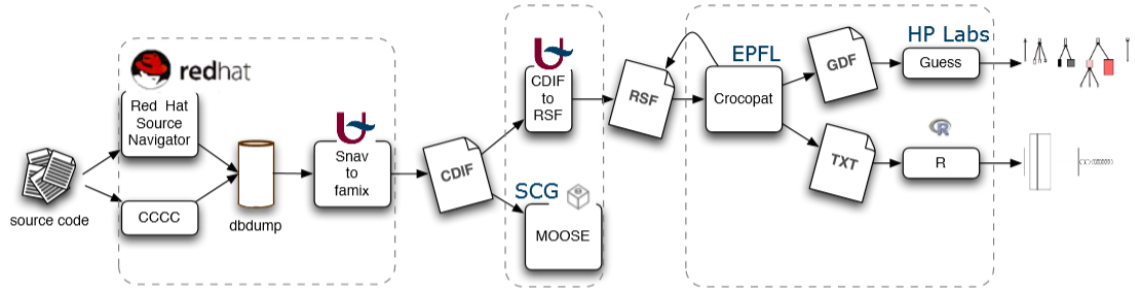
3.1.2 Fact Extraction Tool Chain

Το εργαλείο, που χρησιμοποιήθηκε για την ανάλυση του πηγαίου κώδικα των συστημάτων που χρησιμοποιήθηκαν για την εργασία αυτή, ονομάζεται *Fact Extraction Tool Chain (FETCH)*. Το FETCH περιλαμβάνει κάποια εργαλεία ανοιχτού κώδικα που υποστηρίζουν βασικά θέματα της αντίστροφης μηχανικής (reverse engineering) και στόχο έχει την εξερεύνηση μεγάλων συστημάτων λογισμικού γραμμένα σε C, C++ ή Java για:

(i) την ανάλυση εξαρτήσεων,
(ii) την ανίχνευση μοτίβων (pattern detection),
(iii) την οπτικοποίηση,
(iv) τον υπολογισμό μετρικών και άλλων παρόμοιων ειδών στατικής ανάλυσης

3.1.2.1 Εργαλεία του Fact Extraction Tool Chain

Στην εικόνα που ακολουθεί φαίνονται τα εργαλεία που χρησιμοποιεί ο FETCH, καθώς και το συνδυασμό αυτών προκειμένου να επιτευχθεί η ανάλυση του πηγαίου κώδικα.



Εικόνα 3.1: Η αλυσίδα των εργαλείων που χρησιμοποιούνται από το FETCH

Source Navigator (SN): Εργαλείο λεκτικής ανάλυσης πηγαίου κώδικα το οποίο ξεκίνησε από τη Red Hat. Το σημαντικό του πλεονέκτημα είναι τόσο η ταχύτητα επεξεργασίας που προσφέρει, όσο και η δυνατότητα ανάλυσης σε πολλές γλώσσες. Επιπλέον τα αποτελέσματα της ανάλυσης αποθηκεύονται σε βάση δεδομένων μαζί με πληροφορίες για την τοποθεσία αλλά και την εμβέλεια τους.

C and C++ Code Counter (CCCC): Εργαλείο μέτρησης που αναπτύχθηκε από την Littlefair και εκτελεί μετρήσεις σχετικά με χαρακτηριστικά που δεν συνάγονται από τη δομή του πηγαίου κώδικα, όπως η μέτρηση γραμμών κώδικα, η κυκλωματική πολυπλοκότητα (McCabe's Cyclomatic Complexity) και ο αριθμός των γραμμών που καταλαμβάνουν τα σχόλια του κώδικα.

Source Navigator to FAMIX (snavtofamix): Το εργαλείο αυτό μετατρέπει στοιχεία του πηγαίου κώδικα σε C++ και Java από τον SN σε ένα FAMIX **C**ase **D**ata **I**nterchange **F**ormat (CDIF) μοντέλο, το οποίο θα χρησιμοποιηθεί σε περιβάλλοντα αντίστροφης μηχανικής. Αυτό γίνεται μέσω των ερωτημάτων (queries) στη βάση δεδομένων του Source Navigator και της επίλυσης συσχετίσεων όπως κλήση (invocation), πρόσβαση (accesses), κληρονομικότητα (inheritance) και άλλων που θα αναλυθούν σε ακόλουθη παράγραφο.

CDIF2RSF: Μέσω κάποιων scripts το CDIF μοντέλο μετατρέπεται σε RSF (**R**igi **S**tandard **F**ormat) και έτσι δημιουργείται το Abstract Syntax Graph.

Crocopat: Μηχανή ερωτήσεων σε γράφημα (graph query engine) που δημιουργήθηκε από τον Dirk Beyer και Andreas Noack. Υποστηρίζει ερωτήματα μεγάλων μοντέλων λογισμικού σε RSF και ν-αδικών σχέσεων χρησιμοποιώντας μία γλώσσα παρόμοια με την Prolog (Prolog-alike **R**elation **M**anipulation **L**anguage - RML) αλλά αρκετά γρηγορότερη.

Guess: Σύστημα εξερεύνησης και απεικόνισης γραφημάτων του Eytan Adar. Τα γραφήματα εισόδου είναι σε ASCII format (**G**uess **D**ata **F**ormat) και προσδιορίζουν ιδιότητες των ακμών και των κόμβων ενώ το σύστημα παρέχει διάφορους αλγόριθμους διάταξης.

R project: Εργαλείο που χρησιμοποιείται για στατιστικούς υπολογισμούς και παρέχει boxplots, histograms και regression models.

Η ροή χρήσης των προαναφερθέντων εργαλείων φαίνεται στο προηγούμενο διάγραμμα (Εικόνα 1). Ο πηγαίος κώδικας αναλύεται λεκτικά από τον Source Navigator και γίνονται οι απαραίτητοι υπολογισμοί από τον CCCC. Τα δεδομένα που προκύπτουν από τα εργαλεία αυτά, συνδυάζονται και χρησιμοποιούνται ως είσοδοι

για το Snavtofamix, με αποτέλεσμα τη δημιουργία μιας CDIF αναπαράστασης του FAMIX μοντέλου. Στο επόμενο στάδιο της επεξεργασίας, για να υπάρχει η δυνατότητα να πραγματοποιηθούν ερωτήματα στο μοντέλο, θα πρέπει αυτό να μετατραπεί σε RSF γράφημα. Όταν δημιουργηθεί το γράφημα (RSF αρχείο), με το Crocopat γίνονται ερωτήματα στο γράφημα σε RML, τα αποτελέσματα των οποίων είναι σε GDF αρχείο και μπορούν να οπτικοποιηθούν με τη χρήση του εργαλείου Guess ή αν είναι επιθυμητή η περεταίρω στατιστική ανάλυσή τους, τα δεδομένα (σε TXT αρχείο) εισάγονται στο R project.

3.1.2.2 Οντότητες του κώδικα με βάση την ανάλυση του FETCH

Ο πηγαίος κώδικας αναλύεται με τη διαδικασία που παρουσιάστηκε και από την ανάλυσή του προκύπτουν οι βασικές του οντότητες, οι οποίες χαρακτηρίζονται ως εξής:

Οντότητα	**Περιγραφή**
Module	Αναφέρεται σε ένα directory
File	Αναφέρεται σε ένα αρχείο
Class	Αναφέρεται σε μία κλάση
Macro	Αναφέρεται σε μία μακροεντολή
Struct	Αναφέρεται σε μία δομή
TypeDef	Αναφέρεται σε κάποιον τύπο
Method	Αναφέρεται σε μία μέθοδο
Function	Αναφέρεται σε μία συνάρτηση
Attribute	Αναφέρεται σε μία μεταβλητή
GlobalVar	Αναφέρεται σε καθολικές μεταβλητές

Πίνακας 3.2: Οντότητες που προκύπτουν από τις αναλύσεις του FETCH

3.1.2.3 Σχέσεις μεταξύ οντοτήτων με βάση την ανάλυση του FETCH

Οι οντότητες που αναφέρθηκαν στην προηγούμενη παράγραφο, συνδέονται μεταξύ τους με σχέσεις οι οποίες φαίνονται στο RSF αρχείο που δημιουργείται. Μπορούμε να διαχωρίσουμε τις σχέσεις αυτές σε κατηγορίες όπως φαίνεται στους πίνακες που ακολουθούν:

Κατηγορία	Σχέση	Περιγραφή
Δομή οντοτήτων πηγαίου κώδικα	ModuleBelongsToModule	Δηλώνει ότι η μία οντότητα (directory) βρίσκεται μέσα σε μία άλλη.
	FileBelongsToModule	Δηλώνει σε ποιο directory ανήκει ένα αρχείο.
	ClassBelongsToFile	Δηλώνει σε ποιο αρχείο ανήκει μία κλάση.
	InvocableEntityBelongsToFile	Δηλώνει σε ποιο αρχείο ανήκει μία global μεταβλητή.
	MethodBelongsToClass	Δηλώνει ποια μέθοδος ανήκει σε μία κλάση.
	AttributeBelongsToMethod	Δηλώνει τις μεταβλητές μίας μεθόδου ή δομής (struct).
Μακροεντολές	MacroDefinition	Αναφέρεται στον ορισμό τις μακροεντολής καθώς και στο αρχείο στο οποίο υπάρχει.
	MacroUse	Αναφέρεται στο αρχείο στο οποίο χρησιμοποιείται η μακροεντολή.
Δηλώσεις	DefinedIn	Δηλώνει σε ποιο αρχείο ορίζεται μία μέθοδος ή συνάρτηση. Για κάθε μέθοδο ή συνάρτηση υπάρχει μόνο μία σχέση DefinedIn, αφού μόνο ένα είναι και το αρχείο που περιέχει το σώμα της.
	DeclaredIn	Η σχέση αυτή δηλώνει σε ποιο αρχείο υπάρχουν δηλώσεις τις μεθόδου ή της συνάρτησης. Επομένως σε αντίθεση με τη σχέση DefinedIn, είναι δυνατό να υπάρχει περισσότερες από μία φορές η σχέση αυτή για μία συνάρτηση ή μέθοδο.
Κλήσεις	Calls	Η σχέση αυτή δηλώνει την κλήση μεταξύ συναρτήσεων ή μεθόδων.

Τοποθεσία	EntityLocation	Δηλώνει σε ποιο αρχείο βρίσκεται μία οντότητα (entity).
	EntityBelongsToBlock	Δηλώνει σε ποιο μπλοκ του πηγαίου κώδικα ανήκει η οντότητα.
Συνθήκες	ConditionalCompilation	Δηλώνει κάτω από ποιες συνθήκες ένα μπλοκ κώδικα μεταγλωττίζεται.
Πληροφορίες	Signature	Δηλώνει την «υπογραφή» της μεθόδου. (Συνήθως είναι η τελευταία λέξη μετά την τελεία. Για παράδειγμα για την extract.foo(), η υπογραφή είναι foo().)
	Visibility	Δηλώνει από ποια σημεία είναι ορατή μία μεταβλητή, μία κλάση ή μία μέθοδος.
Τύποι	TypeDefinition	Η σχέση αυτή αναφέρεται σε έναν τύπο με τον οποίο σχετίζεται μία κλάση.
	HasType	Δηλώνει τον τύπου μίας συνάρτησης, μεθόδου, μεταβλητής (global ή όχι).
	HasTypeDefinition	Αυτή η σχέση είναι ένας συνδυασμός των σχέσεων τις κατηγορίας αυτής και συνδέει μία συνάρτηση, μεταβλητή ή μέθοδο με ένα τύπο.
	UsesType	Δηλώνει ποια κλάση χρησιμοποιείται σε μία μέθοδο ή αντίστοιχα ποια δομή χρησιμοποιείται σε μία συνάρτηση.
Γενικές Σχέσεις	InheritsFrom	Σε αντικειμενοστραφή προγραμματισμό, αναπαριστά την κληρονομικότητα μεταξύ κλάσεων.
	Set Variable	Δηλώνει σε ποια συνάρτηση ή μέθοδο υπάρχει καταχώρηση τιμής μίας μεταβλητής.
	Include	Δηλώνει ποιο αρχείο περιλαμβάνεται σε κάποιο άλλο.
	Accesses	Δηλώνει ποια μεταβλητή χρησιμοποιείται από κάποια μέθοδο ή συνάρτηση

Πίνακας 3.3: Σχέσεις που προκύπτουν από το FETCH

3.2 Τεχνικές Ανάκτησης Πληροφορίας

3.2.1 Εισαγωγή

Η αλματώδης αύξηση νέων πληροφοριών που πρέπει να αποθηκεύονται και να ανακτώνται ανά πάσα στιγμή είναι ένα από τα κύρια χαρακτηριστικά της σύγχρονης εποχής και οφείλεται στον υπεράριθμο όγκο δεδομένων που είναι διαθέσιμος. Παρ' όλο οι τρόποι αποθήκευσης όλων αυτών των δεδομένων γίνονται ευκολότεροι, η αναζήτησή, η ανάλυση καθώς και η εξαγωγή χρήσιμων χαρακτηριστικών τους γίνεται όλο και δυσχερέστερη. Οι δυσκολίες αυτές οδήγησαν στην ανάπτυξη και εύρεση μεθόδων οι οποίες αποσκοπούν στο διαχωρισμό δεδομένων σε μικρότερα σύνολα, δηλαδή την ομαδοποίησή τους με βάση κάποιο κοινό τους χαρακτηριστικό (κυρίως σε νοηματικό επίπεδο), τα οποία ο άνθρωπος θα μπορεί να χειρίζεται με μεγαλύτερη ευκολία από ότι ολόκληρο των όγκο πληροφοριών που έχει στη διάθεσή του.

Η διαδικασία ανάκτησης πληροφορίας (information retrieval) στηρίζεται στο αντιστοίχιση των όρων ενός κειμένου (document) και των όρων ενός ερωτήματος αναζήτησης (query). Η δυσκολία που συναντάται στους αλγόριθμους ανάκτησης πληροφορίας έγκειται στην ύπαρξη δύο βασικών χαρακτηριστικών που έχουν οι περισσότερες φυσικές γλώσσες: την *πολυσημία* και τη *συνωνυμία*.

Επομένως είναι προφανές ότι χρειάζεται ένα σύστημα το οποίο θα καταφέρει να ξεπεράσει τα προαναφερθέντα προβλήματα χωρίς να είναι απαραίτητη η επέμβαση του ανθρώπου. Προς αυτή την κατεύθυνση προσανατολίζεται η τεχνική ***L**atent **S**emantic **I**ndexing* (Λανθάνουσα Σημασιολογική Δεικτοδότηση). Πρόκειται για έναν αλγόριθμο ανάκτησης πληροφοριών που στηρίζεται σε μαθηματικές διαδικασίες και με την εισαγωγή των εννοιολογικών δεικτών που δημιουργούνται μεταξύ των όρων ενός κειμένου ανακαλύπτει τις σημασιολογικές σχέσεις που συνδέουν όρους και κείμενα που πιθανά να μην είναι εμφανείς.

3.2.2 Αδυναμίες υπαρχόντων μεθόδων ανάκτησης πληροφορίας

Όπως αναφέρθηκε στην εισαγωγή το θεμελιώδες πρόβλημα που έχουν να αντιμετωπίσουν οι τεχνικές ανάκτησης πληροφορίας είναι δύο από τα χαρακτηριστικά της φυσικής γλώσσας: την πολυσημία και τη συνωνυμία.

Τον όρο *πολυσημία* τον χρησιμοποιούμε για να περιγράψουμε το φαινόμενο κατά το οποίο μία λέξη δεν έχει απαραίτητα μία μόνο εννοιολογική σημασία (concept) αλλά ότι πολλαπλές σημασίες είναι δυνατόν να αποδοθούν στην ίδια λέξη ανάλογα με τα συμφραζόμενα, ενώ με τον όρο *συνωνυμία* αναφερόμαστε στους πολλούς διαφορετικούς τρόπους με τους οποίους μπορεί να αποδοθεί μία έννοια.

Στην περίπτωση πολυσημίας ανακτάται πληροφορία άσχετη με το ερώτημα του χρήστη (για παράδειγμα κείμενα για Internet όταν το ερώτημα αναζήτησης είναι «σερφάρισμα») και κατά συνέπεια μειώνεται το ποσοστό της ακρίβειας (precision), ενώ στην περίπτωση της συνωνυμίας χάνεται η πληροφορία που περιέχει διαφορετικές εκφράσεις της ίδιας έννοιας (όπως για παράδειγμα πληροφορίες για την έννοια «όχημα» ενώ το ερώτημα αναζήτησης είναι «αυτοκίνητο»).

Σε αντίθεση όμως με τις άλλες υπάρχουσες μεθόδους ανάκτησης πληροφορίας, το LSI ξεπερνά τα προαναφερθέντα προβλήματα, καθώς με τη χρήση διανυσμάτων στοχεύει στην σημασιολογική συσχέτιση όρων και κειμένων που αρχικά μπορεί να μην είναι εμφανής.

3.2.3 Τεχνικές Ανάκτησης Πληροφορίας

Στην ενότητα που ακολουθεί θα αναλυθούν τρεις από τις βασικότερες και ευρέως διαδεδομένες τεχνικές ανάκτησης πληροφορίας ώστε να κατανοηθούν οι διάφορες μέθοδοι που χρησιμοποιούνται για το σκοπό αυτό. Οι δύο από τις τεχνικές αυτές είναι ντετερμινιστικές (Vector Space Model, Latent Semantic Indexing) ενώ η άλλη είναι πιθανοτική (Latent Diriclet Allocation).

3.2.3.1 Vector Space Model

3.2.3.1.1 Περιγραφή του Μοντέλου

Το Μοντέλο Διανυσματικού Χώρου (VSM) είναι ένα απλό αλγεβρικό μοντέλο για την αναπαράσταση ενός κειμένου ή όρου με το σκοπό την ανάκτησης πληροφορίας (IR).

Μία συλλογή ή βάση δεδομένων από D σε πλήθος έγγραφα (documents) που περιγράφονται από T όρους (terms) μπορεί να αναπαρασταθεί με έναν πίνακα συσχετίσεων *όρων-κειμένων* (term-document correlation matrix). Ο πίνακας αυτός έχει διαστάσεις T x D και κάθε γραμμή του αναπαριστά κάθε ξεχωριστό όρο που εμφανίζεται στη συλλογή κειμένων, ενώ κάθε στήλη του ένα κείμενο (έγγραφο) της συλλογής αυτής. Στους παραπάνω όρους ανατίθενται βάρη τα οποία αποτελούν στοιχεία του πίνακα (w_{ij}) και χρησιμεύουν για τον υπολογισμό ομοιότητας μεταξύ του ερωτήματος και των κειμένων. Η τιμή κάθε βάρους αντικατοπτρίζει τη σημασία του όρου μέσα στο κείμενο.

Όταν ο πίνακας συσχετίσεων συμπληρωθεί, τα κείμενα μπορούν να αναπαρασταθούν με το διάνυσμα στήλης τους, ένα διάνυσμα που περιλαμβάνει τα βάρη κάθε όρου που υπάρχουν στο κείμενο (ή μηδενικά για τους όρους που δεν εμφανίζονται καθόλου). Αντίστοιχα κάθε όρος μπορεί να αναπαρασταθεί από διάνυσμα γραμμής που του αντιστοιχεί, το οποίο παρουσιάζει πόσο σημαντικός είναι ο όρος αυτός σε κάθε κείμενο της συλλογής.

Το επόμενο βήμα είναι να υπολογιστεί η ομοιότητα μεταξύ των κειμένων. Αυτό πραγματοποιείται συγκρίνοντας τα διανύσματά που αντιπροσωπεύουν τα κείμενα αυτά. Στο Vector Space μοντέλο, δύο κείμενα θεωρούνται όμοια αν περιέχουν τουλάχιστον έναν ίδιο όρο. Κατά συνέπεια, όσο περισσότερους κοινούς όρους έχουν, τόσο υψηλότερο θα είναι το σκορ ομοιότητάς τους.

Στο τελικό στάδιο τα κείμενα διατάσσονται με φθίνουσα σειρά, με κριτήριο το βαθμό ομοιότητάς τους με το ερώτημα αναζήτησης που έχει θέσει ο χρήστης. Έτσι

λαμβάνονται υπόψη και κείμενα που ικανοποιούν μερικώς τις συνθήκες του ερωτήματος.

Η τεχνική αυτή χρησιμοποιεί τις εξής παραμέτρους:

- Βάρος όρων (term weighting): το βάρος των όρων σε ένα κείμενο. Συνήθεις μέθοδοι για τον υπολογισμό του βάρους είναι η συχνότητα με την οποία ο όρος αναφέρεται στο κείμενο (term frequency) ή η μετρική td-idf (term frequency, inverse document frequency) που θα αναλυθεί κατά την παρουσίαση του LSI σε επόμενη παράγραφο.

- Μετρική ομοιότητας (similarity metric): η ομοιότητα μεταξύ των διανυσμάτων δύο κειμένων. Κάποιες από τις γνωστότερες μετρικές είναι η ομοιότητα συνημίτονου (cosine similarity) και η Ευκλείδεια απόσταση.

Το κύριο πλεονέκτημα του Vector Space Model είναι η απλότητα στους υπολογισμούς που περιλαμβάνει το μοντέλο καθώς επίσης και η ευκολία με την οποία κάθε ερώτημα αναζήτησης μπορεί να συγκριθεί με τα υπάρχοντα κείμενα της συλλογής. Σε αντίθεση με τα προαναφερθέντα έρχεται το γεγονός ότι οι διαστάσεις του πίνακα συσχέτισης όρων-κειμένων είναι πιθανό να είναι πολύ μεγάλες (large dimensionality problem) καθώς και το ότι τα διανύσματα των κειμένων μπορεί να είναι αρκετά αραιά (sparseness problem). Επιπλέον η βελτίωση της απόδοσής του καθίσταται δύσκολη χωρίς επέκταση του ερωτήματος αναζήτησης ή ανάδραση από το χρήστη. Όμως το σημαντικότερο –ίσως– μειονέκτημα αυτού του μοντέλου είναι ότι αδυνατεί να αντιμετωπίσει επαρκώς τα προβλήματα της συνωνυμίας και πολυσημίας που αναφέρθηκαν προηγουμένως. Παρ' όλα αυτά, λόγω της εύκολης υλοποίησής του παραμένει αρκετά διαδεδομένο.

3.2.3.1.2 Ταίριασμα Ερωτήσεων Αναζήτησης

Το ταίριασμα ερωτήσεων αναζήτησης (query matching) είναι η διαδικασία που ακολουθείται όταν ο χρήστης εισάγει μία ερώτηση στο vector space model (δηλαδή ένα κείμενο) και περιμένει ως αποτέλεσμα από το μοντέλο άλλα κείμενα της συλλογής με το ίδιο σημασιολογικό περιεχόμενο.

Όπως αναφέρθηκε και στην περιγραφή του μοντέλου, η συλλογή εγγράφων μπορεί να αναπαρασταθεί με τον πίνακα συσχετίσεων όρων-κειμένων και έτσι τα διανύσματα στήλης του πίνακα αυτού να αντιπροσωπεύουν τα κείμενα της συλλογής. Αντίστοιχη διαδικασία πραγματοποιείται και για τα ερωτήματα αναζήτησης. Το ερώτημα που θέτει ο χρήστης αναπαριστάται από ένα διάνυσμα το οποίο θα χρησιμοποιηθεί για τον υπολογισμό της ομοιότητάς του με τα κείμενα της συλλογής.

Έστω λοιπόν ένα κείμενο της συλλογής με διάνυσμα d_j και ένα ερώτημα αναζήτησης με διάνυσμα q. Τα δύο διανύσματα έχουν ίδιες διαστάσεις (T x 1), αφού και τα δύο υπολογίζονται για τους ίδιους όρους. Ο βαθμός ομοιότητας του κειμένου και του ερωτήματος υπολογίζεται ως ο βαθμός συσχέτισης των διανυσμάτων d_j και q του οποίου μέτρο είναι το *συνημίτονο της γωνίας των δύο διανυσμάτων* που δίνεται από τη σχέση:

$$\text{similarity}(d_j, q) = \frac{\vec{d_j}\ x\ \vec{q}}{|\vec{d_j}|\ x\ |\vec{q}|} = \frac{\sum_{i=1}^{T}(w_{i,j}\ x\ w_{i,q})}{\sqrt{\sum_{i=1}^{T} {w_{i,j}}^2}\ \ x\ \sqrt{\sum_{i=1}^{T} {w_{i,q}}^2}}$$

όπου $|\vec{d_j}|$, $|\vec{q}|$ είναι τα μέτρα των διανυσμάτων και $w_{i,j}$, $w_{i,q}$ είναι τα βάρη κάθε όρου που υπάρχουν στο διάνυσμα του κειμένου και του ερωτήματος αντίστοιχα.

Οι τιμές που μπορεί να πάρει το similarity είναι από 0, το οποίο εκφράζει τη μηδενική ομοιότητα, έως το 1 που αντιστοιχεί στο τέλειο ταίριασμα. Έτσι τα κείμενα ταξινομούνται κατά σειρά ομοιότητας και θέτοντας ένα κατώφλι ελέγχου του similarity, θα επιστραφούν τελικά στο χρήστη τα κείμενα εκείνα που έχουν βαθμό ομοιότητας μεγαλύτερο από το κατώφλι και επομένως εκείνα θα είναι αυτά που ταιριάζουν περισσότερο με το ερώτημά του.

3.2.3.2 Latent Semantic Indexing

3.2.3.2.1 Περιγραφή του Latent Semantic Indexing

Το μοντέλο της *Λανθάνουσας Σημασιολογικής Δεικτοδότησης* (**L**atent **S**ematic **I**ndexing – LSI ή αλλιώς **L**atent **S**emantic **A**nalysis – LSA) είναι μία προέκταση του Vector Space Model (VSM) στην οποία παρουσιάζονται όλες οι αλληλεξαρτήσεις μεταξύ όρων και εγγράφων που δημιουργούνται λόγω σημασιολογίας και όχι στηριζόμενες στην επιφανειακή δομή των λέξεων. Ένα σημαντικό πλεονέκτημα της τεχνικής αυτής, όπως προκύπτει από το προαναφερθέν χαρακτηριστικό της, είναι ότι ένα ερώτημα αναζήτησης χρήστη μπορεί να επιστρέψει αποτελέσματα όμοια με αυτό ακόμη και αν δεν περιέχουν κοινές λέξεις.

Για το σκοπό αυτό χρησιμοποιείται η μέθοδος της αποσύνθεσης ιδιάζουσας τιμής (**S**ingular **V**alue **D**ecomposition – SVD) η οποία δέχεται ως είσοδο έναν πίνακα συσχετίσεων όρων-εγγράφων (term-document correlation matrix) και κατασκευάζει ένα σημασιολογικό χώρο στον οποίο οι όροι και τα κείμενα που σχετίζονται τοποθετούνται το ένα κοντά στο άλλο. Δεν είναι απίθανο λοιπόν να παρατηρηθούν όροι κοντά σε ένα κείμενο, παρ' όλο που δεν εμφανίζονται καθόλου σε αυτό, εξαιτίας της εννοιολογικής τους συσχέτισης με το περιεχόμενο του κειμένου. Για παράδειγμα, αν το "document1" περιέχει τη λέξη "mouse" και το "document2" τη λέξη "click", τα δύο κείμενα θα αποκτήσουν μεγάλο βαθμό ομοιότητας λόγω της αναφοράς τους σε κοινό θέμα. Έτσι η τεχνική αυτή, παρ' όλο που δε λαμβάνει καθόλου υπόψη της κανόνες συντακτικού της γλώσσας και στηρίζεται μόνο στον σημασιολογικό χάρτη που κατασκευάζεται, καταφέρνει να ξεπεράσει τα προβλήματα που δημιουργούν τα φαινόμενα της συνωνυμίας και της πολυσημίας.

Όταν ο χρήστης θέτει ένα ερώτημα αναζήτησης το αποτέλεσμα που του επιστρέφεται περιλαμβάνει όλα τα γειτονικά του έγγραφα στο σημασιολογικό χώρο που δημιούργησε η SVD μέθοδος.

Η τεχνική LSI χρησιμοποιεί τις εξής παραμέτρους:

- Βάρος όρων (term weighting): το βάρος των όρων σε ένα κείμενο με μεθόδους υπολογισμού όμοιες με το VSM.

- Πλήθος θεμάτων (number of topics): ο αριθμός των θεμάτων (topics) που θα διατηρηθούν μετά την εφαρμογή της SVD μεθόδου.

- Μετρική ομοιότητας (similarity metric): η ομοιότητα μεταξύ των διανυσμάτων δύο κειμένων όπως και στο VSM.

3.2.3.2.2 Μαθηματικό Υπόβαθρο για την κατανόηση της τεχνικής LSI

Παρακάτω περιγράφονται τα μαθηματικά μοντέλα που χρησιμοποιεί ο αλγόριθμος του LSI.

3.2.3.2.2.1 Term Frequency – Inverse Document Frequency (TF-IDF)

Η συχνότητα όρου και η αντίστροφη συχνότητα εγγράφου (**T**erm **F**requency – **I**nverse **D**ocument **F**requency ή αλλιώς TF-IDF) είναι μία στατιστική μετρική που συχνά χρησιμοποιείται στην ανάκτηση πληροφορίας (information retrieval και την εξόρυξη κειμένου (text mining) για να αξιολογηθεί η σημασία που έχει κάθε όρος (λέξη) σε ένα έγγραφο ή μία συλλογή από έγγραφα. Η σημασία κάθε όρου αυξάνεται αναλογικά με τον αριθμό των φορών που εμφανίζεται στο κείμενο, αλλά αντισταθμίζεται από τη συχνότητα της λέξης σε ολόκληρη τη συλλογή κειμένων. Παραλλαγές του tf-idf χρησιμοποιούνται συχνά από τις μηχανές αναζήτησης ως κεντρικό εργαλείο βαθμολόγησης και κατάταξης ομοιότητας ενός εγγράφου δεδομένου ενός ερωτήματος χρήστη.

Το tf-idf βάρος αποτελείται από δύο όρους: ο πρώτος είναι η συχνότητα όρου (TF) και ο δεύτερος είναι η αντίστροφη συχνότητα εγγράφου (IDF).

Η *Συχνότητα Όρου (Term Frequency)* μετράει πόσο συχνά εμφανίζεται ένας όρος σε ένα έγγραφο. Είναι λογικό ότι εξαιτίας του διαφορετικού μήκους κάθε εγγράφου, ένας όρος να εμφανίζεται πολύ περισσότερες φορές σε μεγάλα έγγραφα παρά σε μικρότερα. Καθίσταται λοιπόν απαραίτητη η κανονικοποίηση, η οποία επιτυγχάνεται με τη διαίρεση της συχνότητας όρου με το μήκος του εγγράφου (δηλαδή με τον συνολικό αριθμό των όρων που περιέχει το κείμενο):

$$tf(t) = \frac{Αριθμός\ φορών\ που\ ο\ όρος\ t\ εμφανίζεται\ σε\ ένα\ έγγραφο}{Συνολικός\ Αριθμός\ Όρων\ σε\ ένα\ έγγραφο}$$

Η *Αντίστροφη Συχνότητα Εγγράφου (Inverse Document Frequency)* μετράει πόσο σημαντικός είναι ένας όρος. Για τον υπολογισμό της συχνότητας όρου, όλες οι λέξεις ενός κειμένου θεωρούνται εξίσου σημαντικές. Παρ' όλα αυτά όμως, υπάρχουν κάποιοι όροι που εμφανίζονται συχνά και δεν έχουν μεγάλη σημασία (όπως για

παράδειγμα τα άρθρα, ορισμένα ρήματα, λέξεις που αποκαλούμε stop words). Έτσι αυτές οι λέξεις θα πρέπει να έχουν μικρότερο βάρος από τις υπόλοιπες, ενώ οι σπάνιες θα πρέπει να θεωρούνται πιο σημαντικές. Η Αντίστροφη Συχνότητα Εγγράφου υπολογίζεται ως εξής:

$$idf(t) = \log \frac{Συνολικός\ Αριθμός\ Εγγράφων}{Αριθμός\ Εγγράφων\ που\ περιέχουν\ τον\ όρο\ t}$$

Συνδυάζοντας αυτούς τους δύο όρους παίρνουμε το tf-idf το οποίο υπολογίζεται όπως φαίνεται στην ακόλουθη σχέση:

$$tf - idf(t) = tf(t)\ x\ idf(t)$$

Συνοψίζοντας, ο υπολογισμός βάρους με τη μέθοδο tf-idf, αποδίδει στον όρο t ένα βάρος στο κείμενο d, το οποίο έχει:

- υψηλότερη τιμή όταν ο όρος t εμφανίζεται πολλές φορές σε μικρό αριθμό κειμένων.
- μικρότερη τιμή όταν ο όρος t συναντάται λιγότερες φορές σε ένα έγγραφο ή όταν εμφανίζεται σε πολλά έγγραφα.
- την χαμηλότερη τιμή όταν ο όρος t εμφανίζεται σε όλα τα κείμενα.

3.2.3.2.2.2 Singular Value Decomposition (SVD)

Ιδιοτιμή ή *χαρακτηριστική τιμή* (eigenvalue) και *ιδιοδιάνυσμα* (eigenvector) ενός τετράγωνου πίνακα A είναι ένα μέγεθος λ και ένα μη μηδενικό διάνυσμα x τέτοια ώστε:

$$Ax = \lambda x$$

Ιδιάζουσα τιμή (singular value) και ένα ζεύγος *ιδιαζόντων διανυσμάτων* (singular vectors) ενός τετράγωνου ή ορθογώνιου πίνακα A είναι ένα μη μηδενικό μέγεθος σ και δύο μη μηδενικά διανύσματα u και v τέτοια ώστε*:*

$$Av = \sigma u \ \ και \ \ A^H u = \sigma v$$

Ο εκθέτης A^H δηλώνει τον Ερμιτιανό ανάστροφο (Hermitian transpose), δηλαδή τον συζυγή ανάστροφο ενός μιγαδικού πίνακα. Αν ο πίνακας είναι πραγματικός, τότε ο ανάστροφός του είναι ο A^T.

Οι ιδιάζουσες τιμές έχουν μεγάλη σημασία όταν ένας πίνακας περιγράφει μετασχηματισμό από ένα διανυσματικό χώρο σε έναν άλλο διανυσματικό χώρο διαφορετικών διαστάσεων.

Η αποσύνθεση ιδιάζουσας τιμής (**S**ingular **V**alue **D**ecomposition –SVD) είναι μία μέθοδος γραμμικής άλγεβρας που χρησιμοποιείται για την παραγοντοποίηση (αποσύνθεση) ορθογώνιου πίνακα A διαστάσεων m x n σε τρεις πίνακες.

Έστω λοιπόν A ένας οποιοσδήποτε πίνακας m x n. Ο πίνακας αυτός μπορεί να αποσυντεθεί στο γινόμενο τριών άλλων πινάκων, των U, Σ, V^T. Δηλαδή:

$$A = U\Sigma V^T$$

όπου U είναι ένας m x m ορθογώνιος πίνακας του οποίου οι στήλες αποτελούν τα ιδιοδιανύσματα του AA^T , V είναι n x n ορθογώνιος πίνακας που οι στήλες του είναι τα ιδιοδιανύσματα του A^TA πίνακα και Σ είναι ένας διαγώνιος πίνακας διαστάσεων m x n του οποίου οι τιμές είναι οι πραγματικές και μη αρνητικές ιδιάζουσες τιμές σ_1, σ_2, … , σ_r του A ταξινομημένες κατά φθίνουσα σειρά. Αν ο πίνακας A είναι πραγματικός, τότε πραγματικοί θα είναι και οι πίνακες U, V. Επιπλέον οι στήλες του U και V αποτελούν αντίστοιχα τα αριστερά και δεξιά ιδιάζοντα διανύσματα που αντιστοιχούν στα διαγώνια στοιχεία του πίνακα Σ.

Για τον υπολογισμό λύσης με μικρότερη διάσταση, αφαιρούμε κάποιους από τους συντελεστές του διαγώνιου πίνακα Σ αρχίζοντας συνήθως με τους μικρότερους. Οι πρώτες k στήλες των U και V πινάκων και οι πρώτες k (μεγαλύτερες) ιδιάζουσες τιμές του πίνακα A χρησιμοποιούνται για να κατασκευάσουν έναν πίνακα τάξης k (A_k) ο οποίος αποτελεί προσέγγιση του A σύμφωνα με τη σχέση:

$$A_k = U_k\Sigma_k {V_k}^T$$

Οι στήλες των U και V είναι ορθογώνιες έτσι ώστε να ισχύει ότι $U^TU = V^TV = I_r$, με r την τάξη του πίνακα A. Ο πίνακας A_k κατασκευάζεται από τις k μεγαλύτερες ιδιόμορφες τριπλέτες του A, δηλαδή μία ιδιάζουσα τιμή και το αριστερό και δεξί ιδιάζον διάνυσμα που της αντιστοιχεί, και είναι η πλησιέστερη προσέγγιση του πίνακα A.

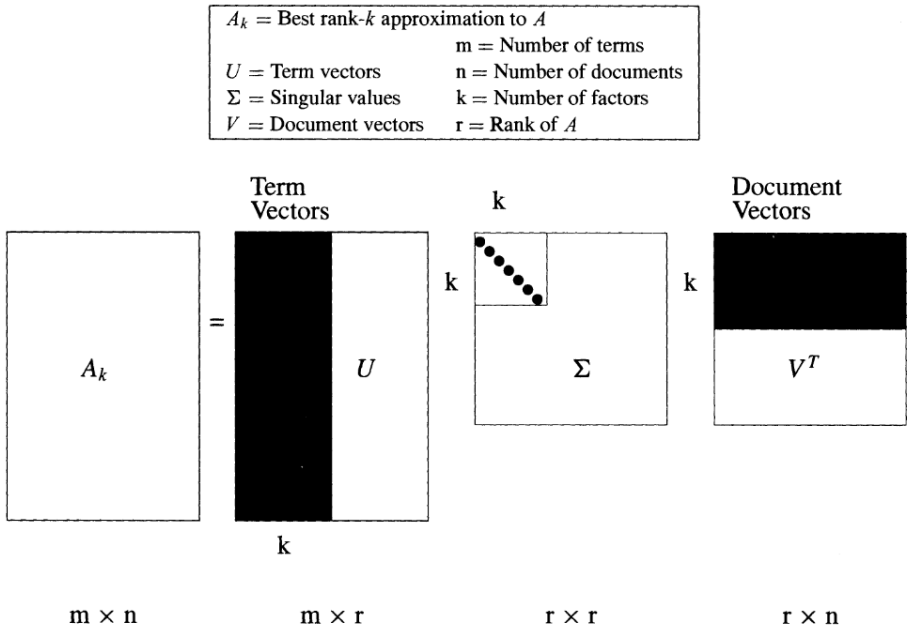
3.2.3.2.3 Τα στάδια της LSI μεθόδου

3.2.3.2.3.1 Δημιουργία πίνακα συσχετίσεων

Αρχικά όπως και στο μοντέλο Vector Space έτσι και στο LSI είναι απαραίτητη η κατασκευή ενός πίνακα συσχετίσεων όρων-κειμένων (term-document matrix). Ο πίνακας αυτός κατασκευάζεται με τον ίδιο τρόπο όπως και στο VSM.

Όπως έχει αναφερθεί σε προηγούμενη παράγραφο, ο πίνακας έχει διαστάσεις m x n, όπου m είναι το πλήθος των όρων που εμφανίζονται στα κείμενα που έχουμε στη διάθεσή μας και n το πλήθος των κειμένων αυτών. Τα στοιχεία του αποτελούν τις συχνότητες εμφάνισης κάθε όρου σε ένα συγκεκριμένο κείμενο, δηλαδή το a_{ij} δηλώνει τη συχνότητα εμφάνισης του όρου i στο κείμενο j. Επειδή ο αριθμός των όρων που εμφανίζονται σε ένα κείμενο είναι τυπικά πολύ μικρότερος από τον αριθμό των όρων ολόκληρης της συλλογής κειμένων, ο πίνακας συσχετίσεων προκύπτει συνήθως πολύ αραιός.

Στη συνέχεια, όταν ο πίνακας έχει συμπληρωθεί μπορεί είτε να χρησιμοποιηθεί στα επόμενα βήματα ως έχει, είτε να εφαρμοστεί η μέθοδος απόδοσης βάρους tf-idf (που αναλύθηκε παραπάνω) και αφού ολοκληρωθούν τα απαραίτητα στάδια, ο πίνακας είναι έτοιμος για το επόμενο βήμα που είναι η εφαρμογή της αποσύνθεσης ιδιάζουσας τιμής (SVD).



Σχήμα 3.2: Αναπαράσταση του SVD για την προσέγγιση A_k του πίνακα A

3.2.3.2.3.2 Υπολογισμός του SVD

Το Latent Semantic Indexing χρησιμοποιεί την αποσύνθεση ιδιάζουσας τιμής προκειμένου να μοντελοποιήσει τις σχέσεις μεταξύ όρων και κειμένων σε έναν ενιαίο χώρο. Με αυτό τον τρόπο είναι δυνατόν να υπολογιστούν ομοιότητες μεταξύ εγγράφων, όρων αλλά και μεταξύ εγγράφων ή ερωτημάτων αναζήτησης και όρων.

Ο πίνακας όρων-εγγράφων (A) που έχει προκύψει από το προηγούμενο βήμα αναλύεται με την SVD σε τρεις πίνακες U, Σ, V^T από τους οποίους ο αρχικός πίνακας μπορεί να προσδιοριστεί με γραμμικό συνδυασμό. Αναλυτικά, ο πίνακας Σ (singular values) μας δίνει ένα στοιχείο για τις διαστάσεις του σημασιολογικού χώρου που προκύπτει, ο πίνακας U αφορά στις συντεταγμένες κάθε όρου μέσα στον εννοιολογικό χώρο και τέλος, ο πίνακας V^T αποδίδει τις συντεταγμένες κάθε εγγράφου στον εννοιολογικό χώρο.

Για να είναι πιο εύκολη η υλοποίηση της διαδικασίας, υπολογίζεται μία προσέγγισή τάξης k του πίνακα συσχετίσεων A (A_k).

Η επιλογή του πλήθους των k ιδιόμορφων τιμών που θα επιλέξουμε για να υπολογίσουμε την προσέγγιση A_k αποτελεί κρίσιμο σημείο για την εφαρμογή του LSI, αφού θέλουμε τέτοιο k ώστε να μην χάνεται η απαραίτητη πληροφορία αλλά και να μπορεί να ανακατασκευαστεί ο πίνακας συσχετίσεων όρων-εγγράφων.

3.2.3.2.3.3 Ταίριασμα ερωτήσεων αναζήτησης

Το ερώτημα αναζήτησης (query) που θέτει ο χρήστης μπορεί να αναπαρασταθεί σαν ένα διάνυσμα στον k-διάστατο χώρο, το οποίο θα συγκριθεί με το αντίστοιχο διάνυσμα κάθε εγγράφου. Η σχέση που ισχύει για το διάνυσμα του ερωτήματος είναι:

$$\hat{q} = q^T U_k {\Sigma_k}^{-1}$$

όπου q είναι το διάνυσμα λέξεων που χρησιμοποιεί ο χρήστης για την ερώτησή του πολλαπλασιασμένο με τα βάρη των όρων.

Επομένως το ερώτημα αναζήτησης αποτελείται από το άθροισμα των διανυσμάτων των όρων ($q^T U_k$) που αντιστοιχούν στους όρους του ερωτήματος επί τον αντίστροφο πίνακα ιδιάζουσων τιμών ${\Sigma_k}^{-1}$.

Χρησιμοποιώντας ως συνάρτηση σχετικότητας τη συνάρτηση συνημίτονου συγκρίνεται το ερώτημα, δηλαδή το διάνυσμα που το αναπαριστά, με καθένα από τα έγγραφα της συλλογής μας (δηλαδή με τα διανύσματά τους). Για τη συνάρτηση συνημίτονου χρησιμοποιούμε το διάνυσμα του ερωτήματος αναζήτησης καθώς επίσης και το διάνυσμα που αντιστοιχεί σε κάθε έγγραφο και προκύπτει από τις στήλες του πίνακα V^T.

Μόλις γίνει ο υπολογισμός της ομοιότητας τα κείμενα ταξινομούνται σε μία λίστα ανάλογα με το βαθμό ομοιότητάς τους. Θέτοντας κάποιο προκαθορισμένο κατώφλι, όπως και στην περίπτωση του VSM, διαπιστώνεται ποια από τα κείμενα που διαθέτουμε είναι περισσότερο σχετικά με το ερώτημα που τέθηκε, δηλαδή πιο ψηλά στη λίστα. Αυτά επιστρέφονται στο χρήστη ως αποτέλεσμα της διαδικασίας.

3.2.3.3 Latent Diriclet Allocation

3.2.3.3.1 Περιγραφή του Latent Dirichlet Allocation

Η *Λανθάνουσα Κατανομή Dirichlet* (**L**atent **D**irichlet **A**llocation –LDA) είναι ένα πιθανοτικό μοντέλο, το οποίο παρέχει ένα μέσο για την αυτόματη δεικτοδότηση, αναζήτηση και ομαδοποίηση των εγγράφων. Όπως το LSI (Latent Semantic Indexing), έτσι και αυτό επιτυγχάνει τα παραπάνω, αφού αρχικά «ανακαλύψει» ένα σύνολο από θέματα στα οποία αναφέρονται τα κείμενα και αναπαραστήσει κάθε κείμενο ως ένα μίγμα θεμάτων.

Η σημαντική διαφορά μεταξύ του LDA και του LSI είναι η μέθοδος που χρησιμοποιείται για να δημιουργηθεί το εννοιολογικό περιεχόμενο των κειμένων. Στο LSI ο σημασιολογικός χώρος είναι αποτέλεσμα της εφαρμογής της μεθόδου αποσύνθεσης ιδιάζουσων τιμών (SVD) στον πίνακα συσχετίσεων όρων-εγγράφων, ενώ στο μοντέλο LDA δημιουργείται χρησιμοποιώντας machine learning αλγορίθμους (όπως η δειγματοληψία Gibbs). Έτσι, μέσω επαναλήψεων, ο αλγόριθμος συμπεραίνει ποιες λέξεις έχουν παρόμοια έννοια –και επομένως σχετίζονται με το ίδιο θέμα – και ποιες από τις έννοιες που «ανακαλύφθηκαν» συμπεριλαμβάνονται στα κείμενα της συλλογής. Κάθε έννοια λοιπόν είναι μία κατανομή πιθανότητας στο σύνολο των όρων που περιέχουν συνολικά όλα τα έγγραφα.

Το μοντέλο LDA αντιμετωπίζει το φαινόμενο της συνωνυμίας και επιπλέον υπερτερεί σε σχέση με το LSI στην αντιμετώπιση της πολυσημίας. Παρ' όλο που η διαδικασία της LDA παρουσιάζει αρκετά θεωρητικά πλεονεκτήματα σε σχέση με το VSM (Vector Space Model) και το LSI καθότι είναι παραγωγική, υπάρχουν ενθαρρυντικά και μη αποτελέσματα σε σχέση με την εφαρμογή της.

Συνοπτικά, ο αλγόριθμος περιλαμβάνει τις εξής παραμέτρους:

- Πλήθος εννοιολογικών θεμάτων (number of topics): ο αριθμός δηλαδή των θεμάτων που θα δημιουργηθούν.
- α : παράμετρος για την εξομάλυνση των θεμάτων των κειμένων (document-topic smoothing parameter)
- β : παράμετρος σχετικά με την εξομάλυνση των θεμάτων των όρων (word-topic smoothing parameter)
- Αριθμός των επαναλήψεων (number of iterations): πόσες φορές θα επαναληφθεί η διαδικασία της δειγματοληψίας
- Μετρική Ομοιότητας (Similarity Metric) : για τον υπολογισμό ομοιότητας μπορούν να χρησιμοποιηθούν μετρικές όμοιες με εκείνες του LSI και VSM όπως είναι η ομοιότητα συνημίτονου.

3.2.3.3.2 Ταίριασμα Ερωτήσεων Αναζήτησης

Στο LDA μοντέλο η ομοιότητα μεταξύ ενός κειμένου d_j και ενός ερωτήματος αναζήτησης Q (query) του χρήστη υπολογίζεται ως η δεσμευμένη πιθανότητα (conditional probability) του ερωτήματος, δεδομένου του κειμένου. Δηλαδή:

$$Similarity(Q, d_j) = P(Q|d_j) = \prod_{q_k \in Q} P(q_k|d_j)$$

όπου q_k είναι η k-οστή λέξη του ερωτήματος αναζήτησης.

Με τον τρόπο αυτό, ένα κείμενο εμφανίζεται σχετικό με το ερώτημα που τέθηκε εάν έχει μεγάλη πιθανότητα να περιέχει λέξεις που έχουν έννοιες οι οποίες αναφέρονται στο ερώτημα.

3.3 Μετρικές Ομοιότητας

Για να καθοριστεί αν κάποιο ερώτημα αναζήτησης (query) του χρήστη εμφανίζει ή όχι ομοιότητα με κάποιο από τα κείμενα της συλλογής, είναι απαραίτητο να χρησιμοποιηθούν μετρικές ομοιότητας που θα αποδίδουν στα υπάρχοντα έγγραφα μία τιμή, ανάλογη του βαθμού ομοιότητας που παρουσιάζουν με το ερώτημα που θέτει ο χρήστης. Μία τέτοια μετρική είναι η Ομοιότητα Συνημίτονου (Cosine Similarity).

Επιπλέον είναι δυνατό να υπολογιστεί η ομοιότητα όχι μόνο μεταξύ διανυσμάτων αλλά και μεταξύ συνόλων και αυτό επιτυγχάνεται με τη μετρική Jaccard που παρουσιάζεται παρακάτω.

3.3.1 Cosine Similarity

Με αυτή τη μετρική μπορεί να εκτιμηθεί η γωνία μεταξύ δύο διανυσμάτων η οποία αποτελεί ένα καλό μέτρο της ομοιότητάς τους. Όσο μικρότερη εμφανίζεται η γωνία, τόσο πιο κοντά είναι τα διανύσματα και επομένως τόσο μεγαλύτερο βαθμό ομοιότητας έχουν.

Έστω δύο διανύσματα $\vec{a}$, $\vec{b}$. Το εσωτερικό γινόμενο των δύο διανυσμάτων δίνεται από την ακόλουθη εξίσωση:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cdot \cos(\widehat{(a,b)})$$

όπου $|\vec{a}|, |\vec{b}|$ είναι τα μέτρα των δύο διανυσμάτων και $\widehat{(a,b)}$ η μεταξύ τους γωνία.

Επομένως μέσα από αυτή μπορούμε εύκολα να υπολογίσουμε τη γωνία μεταξύ των διανυσμάτων, μέσω της τιμή του συνημίτονου της γωνίας αυτής χρησιμοποιώντας τη σχέση:

$$\cos(\widehat{(a,b)}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

Η παραπάνω σχέση υποδεικνύει και την ομοιότητα των δύο διανυσμάτων. Συνεπώς, ισχύει ότι:

$$similarity(\vec{a}, \vec{b}) = \cos(\widehat{(a,b)}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

Οι τιμές που ενδέχεται να έχει το συνημίτονο της γωνίας κυμαίνονται από -1 έως 1. Το -1 αντιστοιχεί σε γωνία 180 μοιρών που σημαίνει ότι τα διανύσματα δεν εμφανίζουν καμία ομοιότητα (0% similarity), ενώ η τιμή 1 προκύπτει όταν η γωνία είναι 360 μοίρες που σημαίνει ότι τα διανύσματα είναι εντελώς όμοια (100% similarity). Κατά συνέπεια, όταν το συνημίτονο έχει τιμή 0, τότε τα διανύσματα είναι 50% όμοια που σημαίνει ότι είναι ορθογώνια μεταξύ τους (σχηματίζουν γωνία 90 μοιρών).

3.3.2 Jaccard Similarity Coefficient

Η μετρική ομοιότητας Jaccard (Jaccard Index ή αλλιώς Jaccard Similarity Coefficient) είναι ένα στατιστικό στοιχείο που χρησιμοποιείται για τη σύγκριση ομοιότητας και ποικιλομορφίας των συνόλων ενός δείγματος. Ο συντελεστής Jaccard μετρά την ομοιότητα μεταξύ πεπερασμένων συνόλων και ορίζεται ως το μέγεθος (cardinality) της τομής τους διαιρούμενο από το μέγεθος της ένωσής των συνόλων. Δηλαδή θεωρώντας δύο σύνολα Α, Β ισχύει ότι:

$$Jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

όπου $|A \cap B|$ είναι το μέγεθος της τομής των συνόλων Α και Β και $|A \cup B|$ το μέγεθος της ένωσής τους.

Γενικά οι τιμές που δίνει η μετρική Jaccard ανήκουν στο σύνολο [0, 1]. Αν $A = \emptyset$ και $B = \emptyset$ τότε $Jaccard(A,B) = 1$.

3.4 Αναπαράσταση Σημασιολογικής Πληροφορικής

3.4.1 Resource Description Framework

Το Πλαίσιο Περιγραφής Πόρων (**R**esource **D**escription **F**ramework –RDF) είναι ένα μοντέλο δεδομένων (data model) για την αναπαράσταση πληροφορίας και ιδιαίτερα μεταδεδομένων (metadata) σχετικά με την περιγραφή πόρων (resources) στον παγκόσμιο ιστό. Είναι ανεξάρτητο από το πεδίο εφαρμογής και μπορεί να απεικονιστεί ως ένας κατευθυνόμενος γράφος με ετικέτες.

Το μοντέλο RDF βασίζεται στα ακόλουθα ερευνητικά πεδία:

- Αναπαράσταση γνώσης (Knowledge Representation)
- Τεχνητή νοημοσύνη (Artificial Intelligence)
- Διαχείριση δεδομένων (data management)
- Εννοιολογικοί γράφοι (conceptual graphs)
- Σχεσιακές βάσεις (relational databases)
- Νοηματικός Ιστός (Sematic Web)

Το RDF μπορεί να χρησιμοποιηθεί επίσης για αντικείμενα τα οποία μπορούν να ταυτοποιηθούν στον παγκόσμιο ιστό, αλλά δεν είναι δυνατό να ανακτηθούν άμεσα από αυτόν, όπως για παράδειγμα ένα βιβλίο ή ένα πρόσωπο. Επιπλέον, προορίζεται για περιπτώσεις όπου οι καταγεγραμμένες πληροφορίες ενδεχομένως να τύχουν επεξεργασίας από κάποια άλλη εφαρμογή και όχι να διαβαστούν από ανθρώπους. Το πλαίσιο περιγραφής πληροφοριών που παρέχει, επιτρέπει την ανταλλαγή τους μεταξύ εφαρμογών χωρίς απώλεια του νοήματος.

Ένα σημαντικό χαρακτηριστικό του μοντέλου αυτού, είναι ότι παρέχει έναν συνεχή τρόπο αναπαράστασης δεδομένων, τέτοιο ώστε η πληροφορία από πολλαπλές πηγές να συνενώνεται και να χρησιμοποιείται σαν να προερχόταν από μία μόνο πηγή.

3.4.2 Θεμελιώδεις έννοιες του RDF

Το σχήμα του RDF βασίζεται στην ιδέα ότι οι πόροι που περιγράφονται έχουν ιδιότητες οι οποίες προσδιορίζονται από κάποιες τιμές. Ακόμη οι πόροι αυτοί μπορούν να περιγραφούν από προτάσεις οι οποίες περιλαμβάνουν τις ιδιότητες αλλά και τις τιμές τους.

3.4.2.1 Πόροι

Οι πόροι (resources) είναι αντικείμενα ή πράγματα (όπως για παράδειγμα συγγραφείς, βιβλία, πόλεις κλπ) καθένας από τους οποίους έχει μία διεύθυνση URI (**U**niform **R**esource **I**dentifier) ως μοναδικό αναγνωριστικό (συνήθως πρόκειται για διεύθυνση URL) και χρησιμοποιείται για την αναγνώριση ή ταυτοποίηση των πόρων.

3.4.2.2 Ιδιότητες και Τιμές

Η περιγραφή των πόρων επιτυγχάνεται με ιδιότητες αλλά και με τιμές ιδιοτήτων.

Οι *ιδιότητες* (properties) είναι ειδική περίπτωση πόρων που έχουν όνομα (παραδείγματος χάριν "book") ή περιγράφουν σχέσεις μεταξύ πόρων.

Οι *τιμές ιδιοτήτων* (property values) προδίδουν τιμές σε ιδιότητες.

3.4.2.3 RDF – Προτάσεις

Κάθε RDF πρόταση (statement) αποτελείται από μία τριπλέτα:

Υποκείμενο – Κατηγόρημα – Αντικείμενο

Το Υποκείμενο (Subject) είναι ένας πόρος στον οποίο αναφέρεται ολόκληρη η πρόταση, το Κατηγόρημα (predicate) είναι μία ιδιότητα ή ένα χαρακτηριστικό που έχει το υποκείμενο και ως Αντικείμενο (object) θεωρείται η τιμή της ιδιότητας αυτής.

3.4.2.4 RDF – Βασική Σύνταξη

Κάθε έγγραφο RDF χαρακτηρίζεται από ένα στοιχείο, τη ρίζα του, το οποίο περιέχει έναν αριθμό από περιγραφές. Το στοιχείο αυτό είναι το <*rdf:RDF*> και αρχικά κάνει αναφορές σε χώρους ονομάτων, με πρώτο τον χώρο ονομάτων RDF (xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#). Οι υπόλοιποι χώροι που αναφέρονται σε αυτό το τμήμα του εγγράφου είναι εξωτερικοί του RDF και περιλαμβάνουν άλλα έγγραφα RDF, τα οποία ορίζουν πόρους που χρησιμοποιούνται στο τρέχον έγγραφο. Το γεγονός ότι μπορεί να γίνει αναφορά σε άλλα RDF έγγραφα είναι εκείνο που επιτρέπει την επαναχρησιμοποίηση των πόρων.

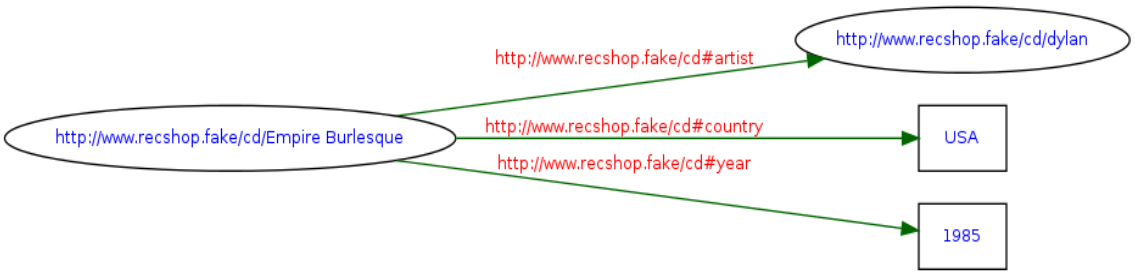
Κάθε μία από τις περιγραφές που περιλαμβάνονται στο έγγραφο αποτελεί ένα στοιχείο <*rdf:Description*> και περιέχει το χαρακτηριστικό *"rdf:about"* το οποίο με τη σειρά του αναφέρεται στον περιγραφόμενο πόρο καθώς και στοιχεία που τον περιγράφουν.

Παράδειγμα RDF σύνταξης παρουσιάζεται στην επόμενη σελίδα:

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:cd="http://www.recshop.fake/cd#">
<rdf:Description rdf:about="http://www.recshop.fake/cd/Empire Burlesque">
        <cd:artist rdf:resource="http://www.recshop.fake/cd/dylan"/>
        <cd:country>USA</cd:country>
        <cd:year>1985</cd:year>
</rdf:Description>
</rdf:RDF>
```

Το παραπάνω παράδειγμα αποτελεί ένα ενδεικτικό της στοιχειώδους σύνταξης ενός RDF εγγράφου. Στην αρχή του, αφού δηλωθεί η έκδοση της xml, φαίνεται η ρίζα του RDF όπως επίσης και το στοιχείο *<rdf:Description>* που περιλαμβάνει, στην προκειμένη περίπτωση, την περιγραφή ενός μόνο πόρου. Ο πόρος, που προσδιορίζεται από το *rdf:about*, είναι ο "*http://www.recshop.fake/cd/Empire Burlesque*". Οι ιδιότητές του φαίνονται αμέσως μετά και είναι οι εξής: *artist, country, year*. Ενώ οι δύο τελευταίες (country και year) έχουν τιμή (*USA* και *1985* αντίστοιχα), η ιδιότητα artist δεν έχει συγκεκριμένη τιμή, αλλά κάνει αναφορά σε ένα πόρο ("*http://www.recshop.fake/cd/dylan"*) που περιέχει τιμές σχετικές με artist.

Όπως είναι φυσικό οι πόροι που μπορούμε να περιγράψουμε με το πλαίσιο που μας παρέχει το RDF μπορεί να είναι αρκετά πιο πολύπλοκοι από το προηγούμενο παράδειγμα. Στις περιπτώσεις αυτές νέα στοιχεία περιλαμβάνονται στην περιγραφή τους ώστε να αποδοθούν οι ιδιότητες τους. Συνέπεια αυτού είναι μία επέκταση του RDF, το RDF Schema, το οποίο είναι μία «γλώσσα» που εμπλουτίζει το μοντέλο δεδομένων RDF με χαρακτηριστικά αντικειμενοστραφούς αναπαράστασης. Συγκεκριμένα, το RDF Schema ορίζει ένα λεξικό για να εκφράζονται οι πόροι, οι κλάσεις, οι ιδιότητές τους καθώς και οι μεταξύ τους σχέσεις.



Εικόνα 3.3: Αναπαράσταση RDF γράφου

3.5 Μετρικές Εκτίμησης Απόδοσης

Για να αξιολογήσουμε την απόδοση κάποιου συστήματος, υπάρχουν διάφορα κριτήρια τα οποία μπορούμε να ερευνήσουμε. Αυτά είναι τα εξής:

- Το στάδιο της *επεξεργασίας των δεδομένων* (Processing Stage): Ένα σύστημα θεωρείται αποδοτικό όταν ο χρόνος αλλά και η μνήμη που χρησιμοποιεί κατά την επεξεργασία των πληροφοριών κυμαίνονται σε λογικά πλαίσια.
- Το στάδιο της *αναζήτησης* (Search): Σε αυτό το στάδιο, ενδιαφερόμαστε κυρίως για τα αποτελέσματα του συστήματος, αξιολογώντας τη σχετικότητά τους με βάση το ερώτημα αναζήτησης που τέθηκε από τον χρήστη.
- Το *σύστημα* συνολικά όπου εξετάζεται το πόσο ικανοποιημένος είναι ο χρήστης από το αυτό.

Στην παράγραφο αυτή θα ασχοληθούμε με το δεύτερο στάδιο και τις μετρικές που αξιολογούν τα αποτελέσματα που προκύπτουν από το σύστημα. Για το σκοπό αυτό χρησιμοποιούμε το *precision* και το *recall*.

Έστω A, B, C, D δεδομένα από τα οποία μερικά έχουν προκύψει ως αποτελέσματα του συστήματος (Retrieved), ενώ άλλα όχι (Not Retrieved). Μερικά από αυτά είναι σχετικά με το ερώτημα αναζήτησης (Relevant) και άλλα δεν έχουν σχέση με αυτό (Irrelevant) σύμφωνα με τον ακόλουθο πίνακα:

	Relevant	Irrelevant
Retrieved	A	B
Not Retrieved	C	D

3.5.1 Precision

Ως precision (ακρίβεια) ορίζεται το κλάσμα των σχετικών αποτελεσμάτων του συστήματος ως προς όλα τα δεδομένα που συμπεριλαμβάνονται στα αποτελέσματα του συστήματος και εκφράζεται συνήθως ως ποσοστό.

Επομένως στην προκειμένη περίπτωση είναι:

$$Precision = \frac{number\ of\ relevant\ items\ retrieved}{total\ number\ of\ items\ retrieved} = \frac{A}{A \cup B}\ x\ 100\%$$

3.5.2 Recall

Recall (ανάκληση) ονομάζουμε το κλάσμα των σχετικών αποτελεσμάτων που προέκυψαν από το σύστημα ως προς όλα τα σχετικά δεδομένα που είτε αυτά παρουσιάστηκαν σαν αποτελέσματα είτε το σύστημα δεν κατάφερε να τα ανακτήσει.

Στην περίπτωση αυτή το recall έχει ποσοστό που υπολογίζεται από τη σχέση:

$$Recall = \frac{number\ of\ relevant\ items\ retrieved}{number\ of\ items\ in\ collection} = \frac{A}{A \cup C}\ x\ 100\%$$

3.5.3 Σχέση μεταξύ Precision και Recall

Το precision και το recall είναι δύο μετρικές *αντιστρόφως ανάλογες* μεταξύ τους. Αυτό σημαίνει ότι όσο αυξάνεται η μία, μειώνεται η άλλη. Πρακτικά, θα μπορούσε κάποιος να σκεφτεί ότι για να αυξήσει το recall του συστήματος θα έπρεπε να δημιουργήσει ένα σύστημα το οποίο να δίνει μεγαλύτερο πλήθος αποτελεσμάτων. Κάτι τέτοιο όμως είναι φανερό πώς θα μείωνε το precision.

3.6 Αναπαράσταση Αρχιτεκτονικής Συστήματος

Ως μοντέλο κάποιου συστήματος ορίζεται μία τυπική περιγραφή ή προδιαγραφή της λειτουργίας, της δομής και της συμπεριφοράς του συστήματος αλλά και του περιβάλλοντός του για κάποιο συγκεκριμένο σκοπό. Συχνά η περιγραφή αυτή αποτελείται τόσο από σχήματα όσο και από κείμενο, το οποίο είναι γραμμένο είτε σε κάποια γλώσσα μοντελισμού (modeling language), είτε σε φυσική γλώσσα. Τα παραπάνω ορίζονται από το **O**bject **M**anagement **G**roup (OMG) ώστε να καθιερωθεί μία ενιαία αναπαράσταση των μοντέλων με εκφραστικό και πλήρη τρόπο.

Για το σκοπό αυτό έχουν αναπτυχθεί και προταθεί διάφορες γλώσσες μοντελισμού, γραφικές αλλά και γραπτές (που περιέχουν κείμενο). Οι πρώτες περιλαμβάνουν διαγράμματα και σύμβολα τα οποία αναπαριστούν έννοιες, καθώς και γραμμές οι οποίες ενώνουν τα σύμβολα και αναπαριστούν τις σχέσεις μεταξύ των οντοτήτων που παρουσιάζονται. Οι δεύτερες αναφέρονται σε κείμενο, στο οποίο χρησιμοποιούνται καθορισμένες λέξεις (keywords) ή ακόμη και όροι φυσικής γλώσσας για την επεξήγηση στοιχείων του συστήματος που μελετάται.

Η πιο διαδεδομένη γλώσσα μοντελισμού είναι η **U**nified **M**odeling **L**anguage (UML) και έχει εδραιωθεί στον τομέα της Τεχνολογίας Λογισμικού τόσο για το σχεδιασμό και την οπτικοποίηση ενός συστήματος, όσο και για τον καθορισμό των προδιαγραφών του. Σύμφωνα λοιπόν με την UML, ένα μοντέλο είναι ένα στιγμιότυπο του UML μεταμοντέλου και με ένα διάγραμμα παρουσιάζεται η γραφική αναπαράσταση του μοντέλου αυτού. Επιπλέον, ένα στοιχείο του μοντέλου αναφέρεται σε ένα στιγμιότυπο κάποιας UML μετακλάσης και η κατάστασή του ορίζεται από τις τιμές των πεδίων του στοιχείου αυτού.

3.6.1 Το πρότυπο Meta Object Facility

Η **M**odel **D**riven **A**rchitecture (MDA) είναι μία προσέγγιση για το σχεδιασμό συστημάτων λογισμικού και ξεκίνησε από την OMG το 2001. Βασίζεται σε μία σειρά από υπάρχουσες τεχνολογίες όπως η UML, το **M**eta **O**bject **F**acility (MOF), το **X**ML **M**etadata **I**nterchange (XMI) και άλλες.

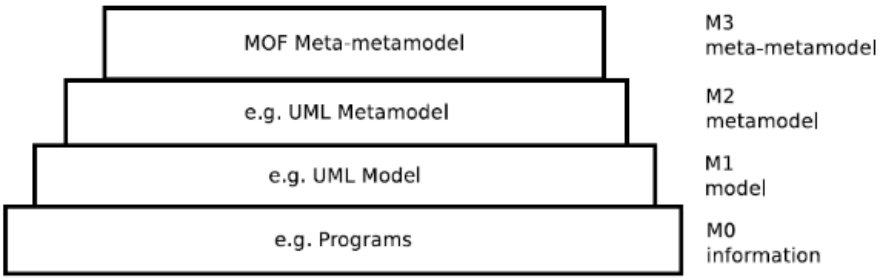
Το Meta Object Facility ορίζει μία αφηρημένη γλώσσα και ένα πλαίσιο για την προδιαγραφή και τον σχεδιασμό μεταμοντέλων, όπου ως μεταμοντέλο θεωρούμε μία αφηρημένη γλώσσα που χρησιμοποιείται για κάποιου είδους μεταδεδομένα. Παραδείγματα μεταμοντέλων είναι η UML, το **C**ommon **W**arehouse **M**etamodel (CWM) καθώς και το ίδιο το MOF. Επιπροσθέτως το MOF καθορίζει ένα πλαίσιο για τον ορισμό μοντέλων, τα οποία περιγράφονται από τα μεταμοντέλα.

Τα MOF μοντέλα σχεδιάζονται με βάση ορισμένους κανόνες έτσι ώστε να διατηρείται η συνέπεια και να μπορούν να χρησιμοποιηθούν από διαφορετικούς παράγοντες.

3.6.1.1 Αρχιτεκτονική Τεσσάρων Επιπέδων

Το κλασικό πλαίσιο για τα μεταμοντέλα βασίζεται σε μία αρχιτεκτονική τεσσάρων επιπέδων, όπου τα στοιχεία κάθε επιπέδου αποτελούν στιγμιότυπα στοιχείων του προηγούμενου επιπέδου.

Στην κορυφή της ιεραρχίας είναι το M3 επίπεδο. Σε αυτό το επίπεδο βρίσκονται τα μετα-μεταμοντέλα, όπως το MOF, τα οποία ορίζουν τη γλώσσα που χρησιμοποιείται για τον καθορισμό των προδιαγραφών ενός μεταμοντέλου. Σύμφωνα με τη βιβλιογραφία, ένα μετα-μεταμοντέλο είναι πιο συμπαγές από το μεταμοντέλο που το περιγράφει και που συχνά μπορεί περιγράφει και άλλα μεταμοντέλα. Επιπλέον, είναι επιθυμητό τα μεταμοντέλα και τα μετα-μεταμοντέλα που συσχετίζονται με αυτά να έχουν κοινή φιλοσοφία σχεδιασμού και να μοιράζονται τις ίδιες δομές. Παρ' όλα αυτά, είναι σημαντικό κάθε επίπεδο της ιεραρχίας να είναι ανεξάρτητο από τα υπόλοιπα και να διατηρεί την ακεραιότητα του σχεδιασμού του. Τα μετα-μεταμοντέλα ορίζονται αυτοαναφορικά. Αυτό σημαίνει ότι ο ορισμός του MOF γίνεται με χρήση του ίδιου του MOF.



Εικόνα 3.4: Αρχιτεκτονική Τεσσάρων Επιπέδων

Το επόμενο επίπεδο, το M2, συχνά καλείται και επίπεδο των μεταμοντέλων και όπως είπαμε και στην αρχή αποτελεί στιγμιότυπο του επιπέδου M3. Το επίπεδο αυτό περιλαμβάνει μεταμοντέλα, δηλαδή γλώσσες μοντελισμού οι οποίες χρησιμοποιούνται στον ορισμό των μοντέλων. Χαρακτηριστικά παραδείγματα μεταμοντέλων είναι η UML καθώς και το **C**ommon **W**arehouse **M**etamodel (CWM). Τα μεταμοντέλα είναι συνήθως πιο περίπλοκα από τα μετα-μεταμοντέλα ειδικά όταν υπάρχει η ανάγκη να περιγραφούν δυναμικές σημασιολογίες.

Το M1 είναι το επίπεδο στο οποίο ανήκουν τα μοντέλα (στιγμιότυπα των μεταμοντέλων του επιπέδου M2). Ο σκοπός των μοντέλων είναι η περιγραφή ενός συστήματος στον πραγματικό κόσμο διαμέσου κάποιων γλωσσών που περιγράφουν έννοιες. Στον τομέα του λογισμικού, το στοιχεία του στρώματος M1 είναι τα μοντέλα των συστημάτων λογισμικού, τα οποία ορίζονται με κάποια UML μοντέλα (διαγράμματα κλάσεων, τα διαγράμματα ακολουθίας κλπ).

Στη βάση της ιεραρχίας βρίσκεται το επίπεδο M0, στο οποίο ανήκουν τα δεδομένα που επιθυμούμε να μοντελοποιήσουμε και ανταποκρίνονται σε στοιχεία που περιγράφονται στο M1.

Στην Εικόνα 3.4 απεικονίζεται το σύστημα των τεσσάρων επιπέδων που περιγράφηκε. Είναι σημαντικό να σημειώσουμε ότι είναι δυνατό να υπάρξουν συστήματα μοντελισμού με διαφορετική αρχιτεκτονική από το παραπάνω. Αυτό σημαίνει ότι μπορούν να αποτελούνται από οποιονδήποτε αριθμό μετά-επιπέδων, αρκεί να είναι περισσότερα από δύο. Παρ' όλα αυτά, το κοινό χαρακτηριστικό όλων των συστημάτων αυτού του είδους είναι η ύπαρξη του ζεύγους κλάση-στιγμιότυπο (class-instance pair) παράλληλα με ένα μηχανισμό για τη μετάβαση από το στιγμιότυπο στην αντίστοιχη κλάση του.

3.6.1.2 Δομή και Αναπαράσταση του MOF

Η πιο πρόσφατη έκδοση του MOF έχει σχεδιαστεί με τέτοιο τρόπο ώστε να είναι στενά συνδεδεμένη με την UML, καθώς για τις προδιαγραφές της έχει χρησιμοποιηθεί τόσο η σύνταξη όσο και η σημασιολογία της UML. Στην πραγματικότητα για τις προδιαγραφές του MOF χρησιμοποιείται ένα υποσύνολο της UML (κυρίως διαγράμματα κλάσεων), μία γλώσσα προδιαγραφής περιορισμών (**O**bject **C**onstraint **L**anguage –OCL) και η φυσική γλώσσα.

Η προδιαγραφή του λοιπόν επιτρέπει δύο παραλλαγές: την βασική (**E**ssential MOF –EMOF) και την πλήρη (**C**omplete MOF –CMOF). Τόσο το EMOF όσο και το CMOF περιγράφονται χρησιμοποιώντας τον εαυτό τους και το καθένα προέρχεται ή επαναχρησιμοποιεί τμήμα της UML.

Ο σκοπός του πλήρους MOF (CMOF) είναι να παρέχει ένα γενικό πλαίσιο για μεταμοντελισμό, ενώ το βασικό MOF (EMOF), που στην πραγματικότητα είναι υποσύνολο του CMOF, περιγράφει χαρακτηριστικά και λειτουργίες των αντικειμενοστραφών γλωσσών προγραμματισμού και του XML.

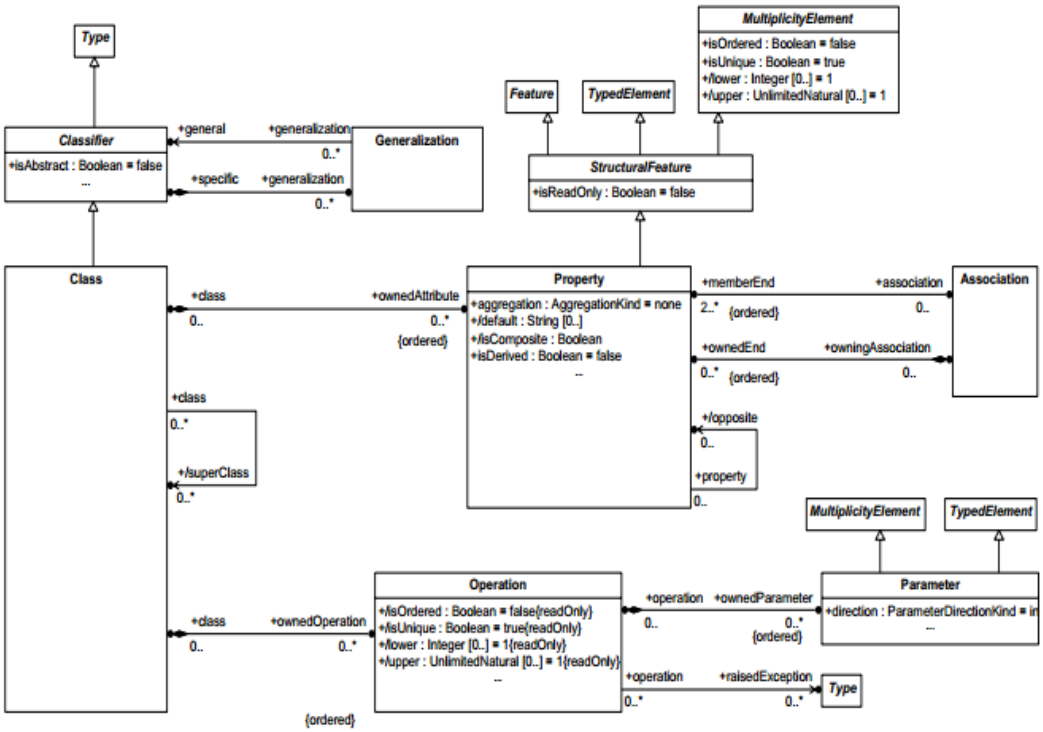
Στην εργασία αυτή χρησιμοποιήθηκε το βασικό MOF και για το λόγο αυτό τα σχήματα που ακολουθούν ορίζουν το EMOF όπως παρουσιάζεται στο έγγραφο προδιαγραφής της τελευταίας του έκδοσης (MOF 2.4.2). Όσο αφορά στον ορισμό του πλήρους MOF, μπορεί κανείς να τον αναζητήσει στο έγγραφο προδιαγραφών που αναφέρεται στη βιβλιογραφία.

3.6.1.3 Η XML Αναπαράσταση

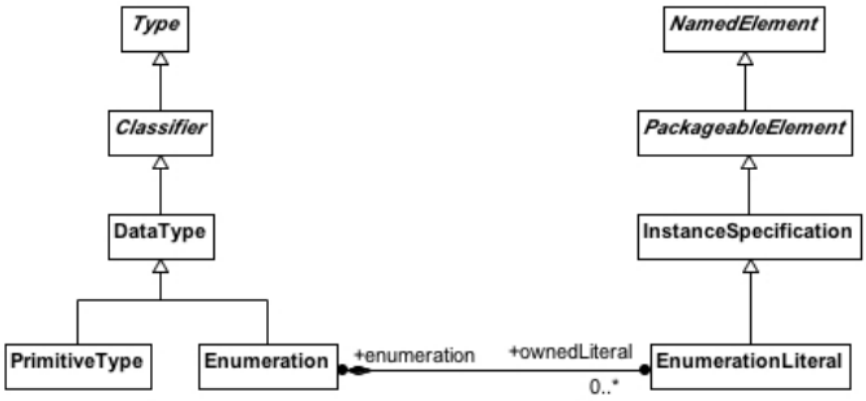
Το MOF αποτελεί ένα πρότυπο που συμβάλλει στη δημιουργία αλλά και στο χειρισμό μοντέλων και μεταμοντέλων. Ωστόσο, το MOF δεν είναι μία υλοποίηση και επειδή μπορεί να φανεί ιδιαίτερα χρήσιμο σε ένα προγραμματιστικά περιβάλλοντα όπου συνυπάρχουν διαφορετικές τεχνολογίες είναι απαραίτητο να βρεθεί ένας τρόπος διασύνδεσής του με άλλα πρότυπα. Για το λόγο αυτό, έχουν δημιουργηθεί προδιαγραφές που περιλαμβάνουν αντιστοιχίσεις από MOF σε άλλα πλαίσια και τεχνολογίες όπως είναι η αντιστοίχιση από MOF σε Java με το **J**ava **M**etadata **I**nterface (JMI) ή εκείνη από το MOF σε XML με το **X**ML **M**etadata **I**nterface (XMI).

Το XMI είναι ένα πρότυπο του OMG που σχετίζεται με την ανταλλαγή πληροφοριών μεταδεδομένων μέσω της E**xe**tensive **M**arkup **L**anguage (XML) και μπορεί να χρησιμοποιηθεί για οποιαδήποτε μεταδεδομένα των οποίων το

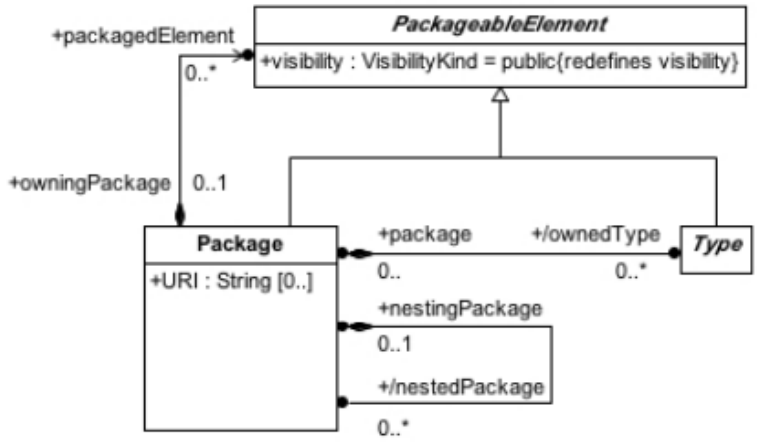
μεταμοντέλων μπορεί να εκφραστεί σύμφωνα με το πρότυπο MOF γεγονός που επιλύει το πρόβλημα της συμβατότητας που παρουσιάστηκε προηγουμένως. Χαρακτηριστικό παράδειγμα είναι η χρήση του XMI για την αναπαράσταση μοντέλων προδιαγεγραμμένων σε UML.



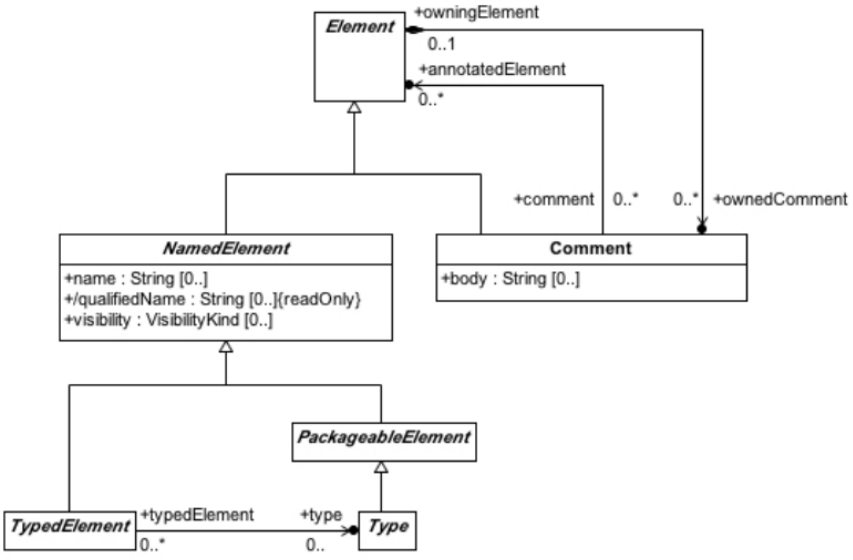
Σχήμα 3.5: Κλάσεις του EMOF (Essentials MOF)



Σχήμα 3.6: Οι τύποι δεδομένων του EMOF



Σχήμα 3.7: EMOF Package



Σχήμα 3.8: Τύποι του EMOF

4

Αναφορές και Αποθήκες Σφαλμάτων

Με τους όρους ανάπτυξη κώδικα ή προγραμματισμό στην επιστήμη των υπολογιστών συχνά αναφερόμαστε στη διαδικασία ανάπτυξής, ελέγχου και συντήρησης πηγαίου κώδικα προγραμμάτων για ηλεκτρονικούς υπολογιστές. Ήδη, σε προηγούμενο κεφάλαιο, έχει γίνει λόγος για τις πολλές και διαφορετικές δεξιότητες και γνώσεις που απαιτούνται από τον προγραμματιστή καθώς και για τις δυσκολίες που πρέπει εκείνος να αντιμετωπίσει στη διαδικασία αυτή. Για το σκοπό αυτό, έχουν αναπτυχθεί πολλά εργαλεία τα οποία διευκολύνουν αρκετά τη δουλειά του προγραμματιστή, προσπαθώντας παράλληλα να διασφαλίσουν την παραγωγή ποιοτικού κώδικα.

Οποιαδήποτε και αν είναι η προσέγγιση της ανάπτυξης λογισμικού και ο σκοπός του προγράμματος που δημιουργείται, θα πρέπει αυτό να ικανοποιεί κάποια βασικά κριτήρια. Αρχικά το πρόγραμμα θα πρέπει να είναι *αποδοτικό* (efficient) τόσο ως προς τη χρήση των πόρων (μνήμη RAM, σκληρό δίσκο, επεξεργαστή κ.α.) όσο και ως προς το χρόνο που χρειάζεται για την εκτέλεσή του. Επιπλέον είναι απαραίτητο ο παραγόμενος κώδικας να είναι *αξιόπιστος* (reliable). Αυτό σημαίνει ότι θα πρέπει να υλοποιεί οτιδήποτε έχει οριστεί στις προδιαγραφές του προγράμματος, προβλέποντας και αντιμετωπίζοντας προβλήματα που μπορεί να προκύψουν κατά την εκτέλεσή του, όπως για παράδειγμα η υπερχείλιση ή η έλλειψη μνήμης. Στη σύγχρονη εποχή, που χαρακτηρίζεται από ραγδαία ανάπτυξη της τεχνολογίας τόσο σε επίπεδο υλικού (hardware) όσο και σε επίπεδο λογισμικού (software), είναι πολύ χρήσιμο το πρόγραμμα που θα δημιουργηθεί να μπορεί να μεταφερθεί σε οποιοδήποτε περιβάλλον χωρίς επαναπρογραμματισμό. Η ιδιότητα αυτή ονομάζεται *μεταφερσιμότητα* (portability) και είναι ένα από τα χαρακτηριστικά του παραγόμενου κώδικα που διευκολύνει την εκτέλεσή του σε διαφορετικά συστήματα. Ο προγραμματιστής θα πρέπει ωστόσο να προβλέπει όσο το δυνατόν περισσότερες περιπτώσεις, στις οποίες υπάρχει πιθανότητα να προκύψουν σφάλματα κατά την εκτέλεση της εφαρμογής. Σε περίπτωση που συμβούν τέτοιου είδους σφάλματα,

κατάλληλα μηνύματα θα εμφανίζονται στο χρήστη με σκοπό να επιδεικνύουν τη σωστή χρήστη του προγράμματος (*ευρωστία* - robustness). Τέλος, ο προγραμματιστής λαμβάνοντας υπόψη του ότι τη συντήρηση και βελτίωση του κώδικα αναλαμβάνουν πολλές φορές ομάδες άλλων προγραμματιστών και όχι αποκλειστικά εκείνος που τον δημιούργησε, είναι απαραίτητο να παράγει ευανάγνωστο και κατανοητό κώδικα ώστε να μπορεί οποιοσδήποτε να διαβάσει κάθε μέρος της εφαρμογής χωρίς ιδιαίτερη δυσκολία (*αναγνωσιμότητα* – readability).

Όπως γίνεται αντιληπτό από τα παραπάνω, η διασφάλιση της ποιότητας μίας εφαρμογής λογισμικού είναι μία πολύπλοκη διαδικασία και εξαρτάται από αρκετούς παράγοντες. Τις περισσότερες φορές μάλιστα δεν είναι εύκολο για τον προγραμματιστή να ελέγξει την ποιότητα του κώδικα που παράγει, αφού αν μπορούσε να εντοπίσει τα σημεία στα οποία υστερεί, θα τα είχε διορθώσει εξαρχής. Παρ' όλα αυτά, έχουν αναπτυχθεί διάφορα εργαλεία που συμβάλλουν τόσο στον ποιοτικό όσο και στον έλεγχο της ορθότητας ενός συστήματος (στατική ανάλυση κώδικα – PMD για Java, test units – JUnit).

4.1 Αναφορές Αστοχιών

Οι αστοχίες είναι ένα φαινόμενο που παρατηρείται συχνά σε προγράμματα λογισμικού και συμβαίνει σε μεγάλο βαθμό εξαιτίας προβληματικών σημείων στον πηγαίο κώδικα τους. Τα σημεία αυτά, όπως έχει αναφερθεί και σε προηγούμενες ενότητες, είναι αρκετά δύσκολο να εντοπιστούν από τον ίδιο τον προγραμματιστή και να διορθωθούν τη στιγμή της δημιουργίας του λογισμικού. Έτσι λοιπόν, στην εξασφάλιση της ακεραιότητας και της ποιότητας του λογισμικού, συμβάλλουν οι αναφορές αστοχιών, οι οποίες περιγράφουν ελαττώματα ή πιθανές δυσλειτουργίες και είναι χρήσιμές τόσο για τη διόρθωση των σφαλμάτων, όσο και για την περεταίρω ανάπτυξη του προγράμματος.

Πιο αναλυτικά, μία αναφορά αστοχίας είναι ένα έγγραφο το οποίο δημιουργείται από κάποιον προγραμματιστή ή ακόμη και τελικό χρήστη ενός συστήματος και περιγράφει τις καταστάσεις στις οποίες το πρόγραμμα δεν ανταποκρίθηκε στις προδιαγραφές του και δεν είχε την αναμενόμενη συμπεριφορά. Επιπλέον περιλαμβάνει διάφορες πληροφορίες σχετικά με την περιγραφή του σφάλματος, βήματα για την αναπαραγωγή του καθώς και πηγαίο κώδικα όπου αυτό είναι αναγκαίο.

Συμπεραίνουμε λοιπόν ότι οι αναφορές αστοχιών αποτελούν ένα μέσο για την ενημέρωση των προγραμματιστών σχετικά με προβλήματα που συναντώνται στον κώδικα. Επειδή όμως συχνά συντάσσονται και από τελικούς χρήστες του προγράμματος λογισμικού, είναι πιθανό να ποικίλουν τόσο στην ποιότητα όσο και στο περιεχόμενο. Παρατηρείται δηλαδή το φαινόμενο οι χρήστες να μην είναι ιδιαίτερα προσεκτικοί κατά τη δημιουργία τους, με αποτέλεσμα να δίνουν λανθασμένες ή και ασαφείς πληροφορίες σχετικά με το σφάλμα που περιγράφουν (για παράδειγμα λανθασμένο λειτουργικό σύστημα ή μη επαρκείς πληροφορίες).

Η χρήση όμως των αναφορών αστοχιών, δημιούργησε την ανάγκη ύπαρξης συστημάτων στα οποία θα καταχωρούνται οι αναφορές αυτές και έτσι θα είναι πιο εύκολο για ένα προγραμματιστή να παρακολουθεί την κατάσταση και τα προβλήματα ενός προγράμματος λογισμικού. Δύο από τα πιο γνωστά συστήματα παρακολούθησης προβλημάτων, το Jira και το Bugzilla, παρουσιάζονται στην επόμενη παράγραφο. Επιπροσθέτως αναλύονται οι αναφορές αστοχιών του Bugzilla όπως επίσης και το μοντέλο τους.

4.2 Συστήματα Παρακολούθησης Προβλημάτων

Εκτός όμως από τον έλεγχο του κώδικα στα στάδια παραγωγής του, είναι πιθανό το λογισμικό που θα δημιουργηθεί, να παρουσιάσει σφάλματα και κατά την εκτέλεσή του τα οποία ο προγραμματιστής δεν κατάφερε να εντοπίσει νωρίτερα. Για την καταγραφή τέτοιου είδους δυσλειτουργιών, υπάρχουν τα συστήματα παρακολούθησης προβλημάτων (bug tracking systems) στα οποία καταχωρούνται, ενημερώνονται και επιλύονται προβλήματα σχετικά με κάποιο πρόγραμμα τόσο από χρήστες όσο και από άλλους προγραμματιστές.

Στην τεχνολογία λογισμικού, τα συστήματα αυτά είναι σχεδιασμένα έτσι ώστε να συνεισφέρουν στην εγγύηση ποιότητας του κώδικα καθώς και να βοηθούν τους προγραμματιστές να αναφέρουν αστοχίες που είναι πιθανό να προκύψουν. Η χρησιμότητα τέτοιων συστημάτων έγκειται στο γεγονός ότι παρέχουν μία γενική εποπτεία των αναφορών σφαλμάτων που δημιουργούνται κατά την ανάπτυξη κώδικα και της κατάστασή τους (status).

Δύο από τα πιο γνωστά συστήματα παρακολούθησης σφαλμάτων είναι το Bugzilla και το JIRA. Για το πρώτο θα γίνει εκτενής αναφορά σε επόμενη παράγραφο. Το JIRA είναι ένα εμπορικό πρόγραμμα και απευθύνεται κυρίως σε επιχειρήσεις. Συνήθως χρησιμοποιείται ως σύστημα παρακολούθησης σφαλμάτων αλλά και ως σύστημα διαχείρισης έργων. Συγκεκριμένα επιτρέπει την απεικόνιση του κύκλου ζωής ενός προβλήματος καθώς και αρκετούς τρόπους για την παροχή πληροφοριών σχετικών με προβλήματα ή ροές εργασίας σε πραγματικό χρόνο.

4.2.1 Bugzilla

Το Bugzilla είναι ίσως το πιο γνωστό από τα διαδικτυακά εργαλεία διαχείρισης προβλημάτων, το οποίο αναπτύχθηκε, χρησιμοποιήθηκε από το Mozilla και κυκλοφόρησε ως λογισμικό ανοιχτού κώδικα. Χρησιμοποιείται για την καταχώρηση και αποθήκευση αναφορών αστοχιών πολλών προγραμμάτων όπως το Mozilla και το Eclipse, αλλά και εφαρμογών λογισμικού των Linux.

Το Bugzilla προσφέρει τη δυνατότητα τόσο σε απλούς χρήστες όσο και σε προγραμματιστές όχι μόνο να αναζητήσουν ήδη υπάρχουσες αναφορές σφαλμάτων, αλλά και να δημιουργήσουν εκ νέου καινούριες. Οι χρήστες είναι σε θέση να αναζητήσουν κάποια αναφορά απλά πληκτρολογώντας στο πεδίο αναζήτησης (από απλό κείμενο μέχρι και το αναγνωριστικό της αναφοράς – bug id) το οποίο έχει προαιρετικό φιλτράρισμα για το προϊόν ή για την κατάσταση της αναφοράς (status) . Για εκείνους που επιθυμούν πιο εξελιγμένη αναζήτηση, υπάρχει η επιλογή της «Σύνθετης Αναζήτησης» (Advanced Search) όπου μπορούν να φιλτράρουν σχεδόν οποιοδήποτε πεδίο της αναφοράς σφάλματος που αποθηκεύεται στη βάση δεδομένων του Bugzilla.

Το Bugzilla παρέχει ένα πολύ εξελιγμένο σύστημα αναφοράς, το οποίο αρχικά φαντάζει περίπλοκο, λόγω της διεπαφής χρήστη (user interface) που δεν είναι ιδιαίτερα φιλική. Κατά τη διαδικασία της δημιουργίας μίας αναφοράς σφάλματος (bug report) ο χρήστης καλείται να συμπληρώσει μία φόρμα, η οποία περιλαμβάνει κάποια πεδία απαραίτητα για την καταχώρηση της αναφοράς στη βάση δεδομένων.

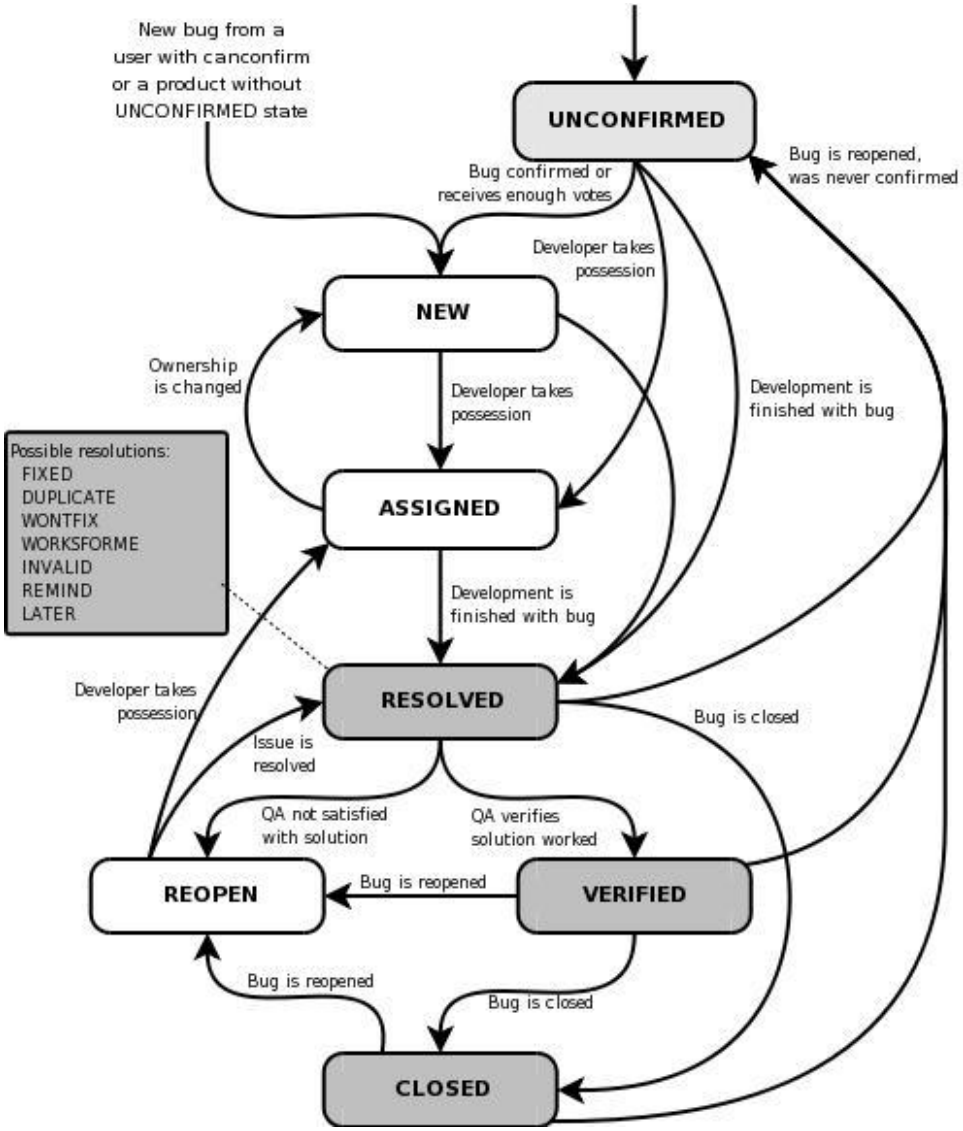
Οι πληροφορίες αυτές αφορούν λεπτομέρειες της αστοχίας που παρατηρήθηκε και περιλαμβάνουν μία σχετική περίληψη του σφάλματος, το προϊόν στο οποίο αναφέρονται και τη συγκεκριμένη έκδοσή του, την πλατφόρμα στην οποία είναι εγκατεστημένο το προϊόν, την προτεραιότητά του και άλλα. Επιπλέον παρέχεται στο χρήστη η δυνατότητα να προσθέσει σχόλια, στα οποία συνήθως επεξηγεί λεπτομερώς το σφάλμα που παρατήρησε. Διάφορες ενημερώσεις της κατάστασης του σφάλματος επιτρέπονται, μαζί με σημειώσεις από τον χρήστη που εντόπισε το σφάλμα καθώς και παραδείγματα στα οποία εμφανίζεται.

Όταν η αναφορά ολοκληρωθεί, το Bugzilla ειδοποιεί με μήνυμα ηλεκτρονικού ταχυδρομείου τους χρήστες για την νέα καταχώρηση. Τα σφάλματα στον κώδικα μπορούν να σταλούν από οποιονδήποτε και να ανατεθούν σε έναν συγκεκριμένο προγραμματιστή. Στην πραγματικότητα, το Bugzilla επιτρέπει την δημόσια καταχώρηση σφαλμάτων, τα οποία ανατίθενται σε κάποιον υπεύθυνο για την περεταίρω ανάθεσή τους στους αρμόδιους προγραμματιστές καθώς και τον ορισμό του επιπέδου προτεραιότητας.

Παρ’ όλο που η χρήση του φαίνεται ιδιαίτερα πολύπλοκή, οι δυνατότητες που προσφέρει είναι αξιόλογες και επαρκούν για τους περισσότερους οργανισμούς που το χρησιμοποιούν. Ένα από τα χαρακτηριστικά του Bugzilla είναι η ταχύτητά του και η μη επιβαρυμένη υλοποίησή του, ελαχιστοποιώντας όσο είναι δυνατό τις κλήσεις προς τη βάση δεδομένων. Επιπλέον, οι διαχειριστές έχουν τη δυνατότητα να δημιουργήσουν ακόμη και γραφήματα (πίνακες, γράφημα πίτας) ώστε να απεικονίσουν καλύτερα τα αποθηκευμένα δεδομένα που σχετίζονται με το εκάστοτε πρόγραμμα λογισμικού όπως επίσης και να παρατηρήσουν τις εξαρτήσεις που αναπτύσσονται μεταξύ των σφαλμάτων μέσα από τους γράφους εξαρτήσεων (dependency graphs).

Μία αναφορά αστοχίας στο Bugzilla ακολουθεί μία σταθερή «διαδρομή» στο σύστημα που ονομάζεται *κύκλος ζωής* της αναφοράς και παρουσιάζεται στο παραπάνω διάγραμμα καταστάσεων. Όταν η αναφορά καταχωρηθεί για πρώτη φορά, βρίσκεται στην κατάσταση (state) “New” (δηλαδή καινούρια) και είναι είτε “Confirmed” (επιβεβαιωμένη) είτε “Unconfirmed”. Έπειτα η αναφορά ανατίθεται σε κάποιον προγραμματιστή (οπότε μεταβαίνει στην κατάσταση “Assigned”) ο οποίος είναι υπεύθυνος να ερευνήσει την προέλευση του προβλήματος και να διορθώσει τον κώδικα. Όταν επιλυθεί το πρόβλημα, η αναφορά χαρακτηρίζεται ως “Resolved” (δηλαδή επιλυμένη) και υπάρχει πιθανότητα να γίνει “Reopened” αν η επίλυσή της χαρακτηριστεί μη επαρκής. Διαφορετικά η κατάστασή της θα είναι “Verified” (επαληθευμένη). Οι verified αναφορές αποκτούν κατάσταση “Closed” όταν η αστοχία

έχει επιλυθεί αποτελεσματικά, ενώ μπορεί και να μεταβούν σε κατάσταση “Reopened”.



Εικόνα 4.1: Κύκλος ζωής μίας αναφοράς αστοχίας (bug report)

4.3 Μοντέλο Αναφοράς Αστοχιών

Σε αυτή την παράγραφο παρουσιάζεται το μεταμοντέλο των αναφορών αστοχιών στο Bugzilla. Στη συνέχεια θα αναλυθούν οι σημαντικότερες οντότητες του μοντέλου καθώς και οι σχέσεις που υπάρχουν μεταξύ τους.

4.3.1 Οντότητες Μοντέλου

Η *αναφορά σφάλματος* (BugReport) αποτελεί την κυριότερη οντότητα σε ένα σύστημα παρακολούθησης αστοχιών όπως το Bugzilla και περιέχει πληροφορίες που είναι παρόμοιες για τα περισσότερα συστήματα. Στο συγκεκριμένο μοντέλο, οι πληροφορίες αυτές είναι άμεσα συνδεδεμένες με την αναφορά, ενώ εκείνες που σχετίζονται έμμεσα αναπαρίστανται με ξεχωριστές οντότητες. Πιο συγκεκριμένα υπάρχουν τα εξής:

- *BugID*: Στο Bugzilla όλες οι αναφορές αστοχιών έχουν έναν αναγνωριστικό αριθμό, μοναδικό για καθεμία. Ο αριθμός αυτός είναι ακέραιος και αυξάνεται κατά την προσθήκη ενός νέου bug report στο σύστημα.
- *Date Submitted*: Η ημερομηνία στην οποία καταχωρήθηκε η συγκεκριμένη αναφορά στο σύστημα.
- *Status*: Η κατάσταση μίας αναφοράς αστοχίας αφορά στα στάδια που περνάει κατά τον κύκλο ζωής της, όπως περιγράφεται στην προηγούμενη παράγραφο. Παραδείγματα καταστάσεων είναι η “new”, “resolved”, “fixed” και άλλα.
- *Priority*: Τροποποιείται από τον προγραμματιστή που είναι υπεύθυνος για την επίλυση του σφάλματος που περιγράφεται, προκειμένου να θέσει προτεραιότητα στις αναφορές με τις οποίες πρέπει να ασχοληθεί.
- *Resolution*: Όταν μία αναφορά σφάλματος βρεθεί στην κατάσταση “resolved”, είναι πιθανό να υπάρχουν περισσότερες από μία αναλύσεις, καθεμία εκ των οποίων να προτείνει διαφορετικό τρόπο για την επίλυση της αστοχίας.
- *Commit Link*: Τις περισσότερες φορές η επίλυση ενός σφάλματος απαιτεί τροποποιήσεις σε τμήματα του πηγαίου κώδικα του προγράμματος λογισμικού. Ο κώδικας αυτός συνήθως είναι αποθηκευμένος σε ένα σύστημα διαχείρισης εκδόσεων πηγαίου κώδικα (όπως για παράδειγμα είναι το Git) και ανανεώνεται όποτε γίνονται αλλαγές σε αυτόν. Το συγκεκριμένο link αφορά στην τοποθεσία του κώδικά μέσα στο εν λόγω σύστημα.

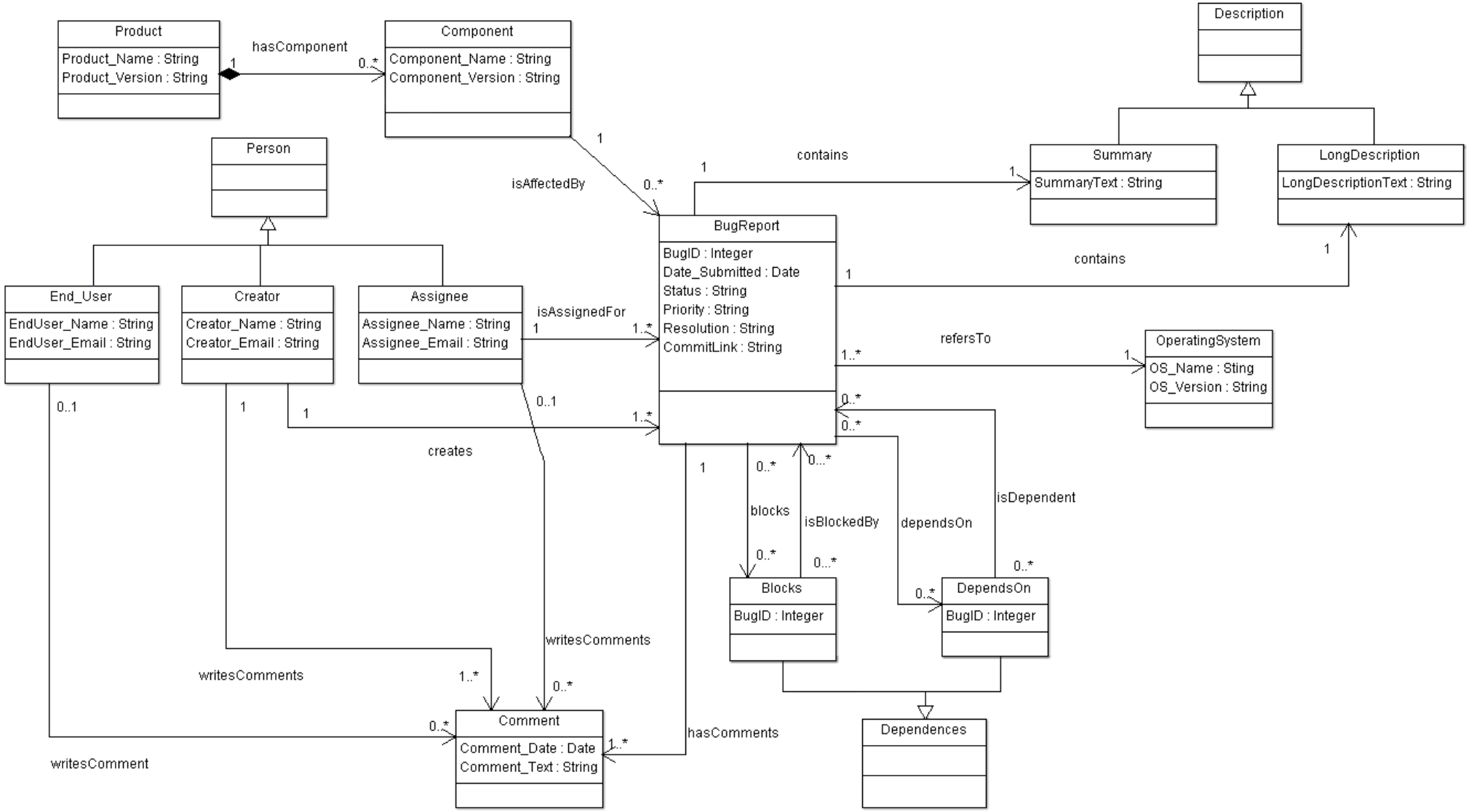
Επόμενη σημαντική οντότητα είναι ο *Άνθρωπος* (Person), η οποία αποτελείται από τρείς επιμέρους οντότητες: το Δημιουργό (Creator) της αναφοράς αστοχίας, τον Τελικό χρήστη (end user) του προγράμματος λογισμικού και τον προγραμματιστή στον οποίο ανατίθεται η επίλυση του σφάλματος (Assignee). Εκτός από τον βασικό ρόλο κάθε κατηγορίας, ο καθένας από τους παραπάνω μπορεί να καταχωρήσει σχόλια σχετικά με τις αναφορές καθώς και να δημιουργήσει εκ νέου κάποιο bug report, γεγονός που του προσδίδει τις ιδιότητες ενός Creator. Οι πληροφορίες που

αποθηκεύονται για κάθε άτομο που αλληλεπιδρά με μία αναφορά σφάλματος είναι το όνομα και το email του και γι' αυτό το λόγο η συγχώνευσή τους σε μία οντότητα δε θα δημιουργούσε κάποιο πρόβλημα ανακρίβειας ή απώλειας δεδομένων. Παρ' όλα αυτά για να γίνει περισσότερο αντιληπτός ο διαχωρισμός των ρόλων, στο μεταμοντέλο που παρουσιάζεται στην ενότητα αυτή, εμφανίζονται ως ξεχωριστές οντότητες που ανήκουν στην γενικότερη οντότητα Person.

Μία ακόμη οντότητα για το μεταμοντέλο είναι τα *σχόλια* (Comment), στα οποία κυρίως περιγράφονται τρόποι για την αναπαραγωγή του σφάλματος ή ακόμα και διάφοροι προβληματισμοί που προκύπτουν κατά τη διάρκεια επίλυσης του. Όπως αναφέρθηκε στην προηγούμενη παράγραφο, ένα σχόλιο μπορεί να δημιουργηθεί από οποιονδήποτε άνθρωπο είτε έχει το ρόλο Creator, είτε End User, είτε Assignee και για κάθε σχόλιο αποθηκεύονται πληροφορίες όπως το κείμενό του (Comment Text), δηλαδή την περιγραφή του, καθώς επίσης και η ημερομηνία δημιουργίας του (Comment Data).

Μερικά από τα *προϊόντα* (Product) λογισμικού είναι πιθανό να αποτελούνται από πολλά *επιμέρους στοιχεία* (Components). Για κάθε προϊόν στο Bugzilla, όπως και για κάθε component σημαντικές πληροφορίες είναι το όνομά του (Product/Component Name), αλλά και η έκδοσή του (Product/Component Version). Σε μία αναφορά αστοχίας του Bugzilla είναι επίσης απαραίτητο να καταχωρηθεί και το *λειτουργικό σύστημα* (Operating System) στο οποίο παρατηρήθηκε το σφάλμα αναφέροντας το όνομα και την έκδοσή του (OS Name/Version).

Τέλος, υπάρχουν οι εξαρτήσεις (Dependences) οι οποίες θα γίνουν περισσότερο κατανοητές στην επόμενη παράγραφο όπου γίνεται λόγος για τις σχέσεις των οντοτήτων και οι *περιγραφές* (Descriptions), στις οποίες ανήκουν η περίληψη (Summary), δηλαδή μία σύντομη περιγραφή του σφάλματος που παρατηρήθηκε και μία εκτενέστερη περιγραφή (Long Description).



Εικόνα 4.2: Μεταμοντέλο Αναφορών Σφαλμάτων στο Bugzilla

4.3.2 Σχέσεις μεταξύ Οντοτήτων

Ξεκινώντας από το Product, η πρώτη σχέση που συναντάμε είναι η *σχέση Product-Component*, η οποία δηλώνεται στο μοντέλο ως *hasComponent*. Κάθε προϊόν λοιπόν είναι δυνατόν να έχει περισσότερα από ένα στοιχεία και γι' αυτό το λόγο η σχέση hasComponent είναι 1-προς-N.

Τα επιμέρους στοιχεία ενός προϊόντος (components) είναι εκείνα στα οποία παρουσιάζονται και καταγράφονται σφάλματα σε συστήματα όπως το Bugzilla. Αυτή η 1-προς-N σχέση *Component-Bug Report* αναφέρεται στο μοντέλο ως *isAffectedBy*, αφού είναι προφανές ότι ένα στοιχείο του προγράμματος λογισμικού, μπορεί να έχει περισσότερες από μία αναφορές αστοχιών που σχετίζονται με αυτό, ενώ αντίθετα μία αναφορά αστοχίας σχετίζεται με ένα μόνο συγκεκριμένο component.

Στην προηγούμενη παράγραφο, περιγράφηκε λεπτομερώς ο ρόλος κάθε οντότητας που ανήκει στη γενική κατηγορία Person (Creator, Assignee, End User). Κάθε μία οντότητα έχει διαφορετικών ειδών αλληλεπιδράσεις με τις υπόλοιπες που παρουσιάζονται στο μοντέλο. Βασική και ταυτόχρονα απαραίτητη είναι η σχέση μεταξύ *Creator-Bug Report*, η οποία ονομάζεται *creates* και είναι 1-προς-N, αφού δεν υπάρχει περιορισμός ως προς το πλήθος των αναφορών αστοχιών που μπορεί να δημιουργήσει ένας άνθρωπος. Έχει ήδη αναφερθεί ότι μετά τη δημιουργία μίας αναφοράς, αναλαμβάνει κάποιος προγραμματιστής να επιδιορθώσει το πρόβλημα. Αυτή είναι η σχέση *Assignee-Bug Report* και είναι 1-προς-N (isAssignedFor). Οι οντότητες που ανήκουν στην κατηγορία Person έχουν τη δυνατότητα να γράψουν ένα σχόλιο, σχέση *writesComment* και είναι 1-προς-N, επειδή κάθε άνθρωπος έχει το δικαίωμα να γράψει περισσότερα από ένα σχόλια, αλλά κάθε σχόλιο συντάσσεται και καταχωρείται μόνο από ένα συγκεκριμένο άτομο.

Σχέση 1-προς-N είναι και η σχέση *Bug Report-Comment*. Μία αναφορά αστοχίας έχει ένα ή και περισσότερα σχόλια. Όμως ένα σχόλιο αναφέρεται σε μία μόνο αναφορά και δεν μπορεί να αντιστοιχεί σε περισσότερες. Η σχέση αυτή έχει το όνομα hasComments.

Κατά τη δημιουργία μιας αναφοράς σφάλματος στο Bugzilla, συμπληρώνεται το πεδίο Summary και το αντίστοιχο Long Description. Έτσι έχουμε τη σχέση contains μεταξύ *Bug Report-Summary* και *Bug Report-Long Description*. Επειδή για κάθε μία αναφορά υπάρχει μόνο ένα πεδίο Summary και Long Description, η σχέση αυτή είναι 1-προς-1.

Επιπλέον ένα bug report σχετίζεται εκτός από το component και με το λειτουργικό σύστημα στο οποίο εντοπίστηκε το σφάλμα. Έτσι προκύπτει και η σχέση N-προς-1 *Bug Report-Operating System* (refersTo), εφόσον πολλές αναφορές σφαλμάτων είναι δυνατό να σχετίζονται με ένα Λειτουργικό Σύστημα.

Τέλος μεταξύ της αναφοράς αστοχίας και των εξαρτήσεων (*Bug Report – Dependencies*) υπάρχουν οι σχέσεις *blocks* και *dependsOn*. Επειδή η επεξήγηση αυτών των σχέσεων είναι περίπλοκη, θα περιγραφούν χρησιμοποιώντας ένα παράδειγμα. Έστω ότι έχουμε τις αναφορές σφαλμάτων BR1, BR2, BR3 και θεωρούμε ότι για την επίλυση της BR1 πρέπει πρώτα να προηγηθεί η επίλυση της BR2 και της BR3. Αυτό σημαίνει ότι η BR1 εξαρτάται από τις άλλες δύο (σχέση dependsOn) και παράλληλα ότι οι BR2 και BR3 μπλοκάρουν την επίλυση της BR1 (σχέση blocks). Παρατηρούμε λοιπόν ότι για κάθε σχέση dependsOn δημιουργείται και η αντίστοιχη blocks όπως επίσης και ότι μία αναφορά αστοχίας είναι δυνατό να

εξαρτάται από περισσότερες αναφορές καθώς και να μπλοκάρει περισσότερες από μία αναφορές.

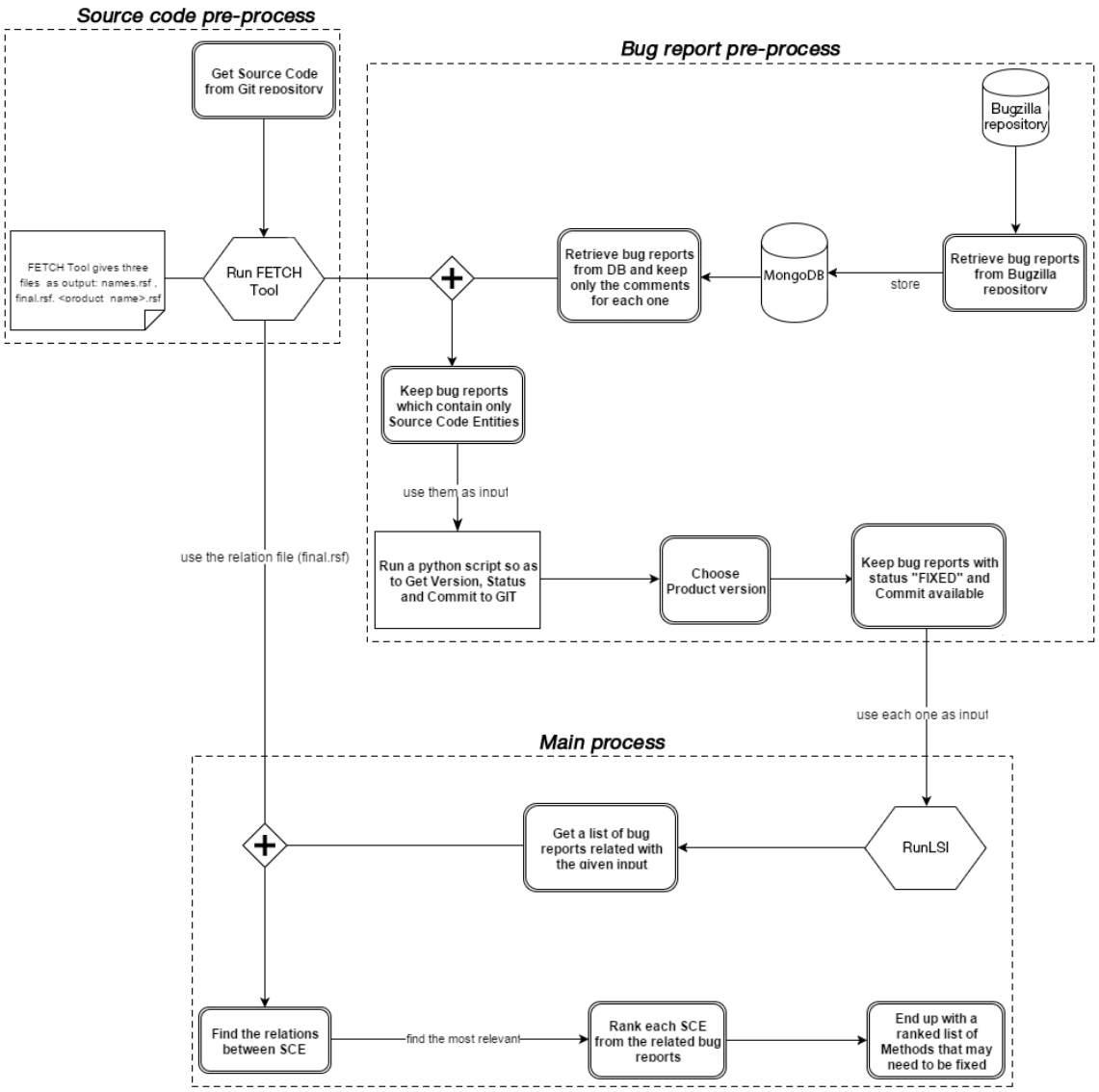
5

Αρχιτεκτονική του Συστήματος

Στην ενότητα αυτή θα περιγραφούν διεξοδικά όλα τα βήματα που πραγματοποιήθηκαν για το σχεδιασμό και την ανάπτυξη του τελικού συστήματος. Το σύστημα που υλοποιήθηκε παρέχει στο χρήστη τη δυνατότητα να εντοπίζει τα «ελαττωματικά» σημεία του πηγαίου κώδικα, τα οποία είναι υπεύθυνα για σφάλματα που εντοπίζονται κατά την εκτέλεση ενός προγράμματος λογισμικού. Για να λειτουργήσει επιτυχώς το εν λόγω σύστημα, χρειάζονται οι αναφορές αστοχιών (bug reports) στις οποίες στηρίζεται το μεγαλύτερο ποσοστό του καθώς και ο πηγαίος κώδικας.

Η αρχιτεκτονική του bug localization συστήματος, το οποίο δημιουργήθηκε στην παρούσα διπλωματική εργασία, χωρίζεται σε τρία βασικά στάδια, που παρουσιάζονται ξεκάθαρα στην διαγραμματική αναπαράσταση που ακολουθεί. Καθένα από τα στάδια αυτά είναι εξίσου απαραίτητο και αποτελεί αναπόσπαστο κομμάτι για τον υπολογισμό του τελικού αποτελέσματος.

Αρχικά ο κώδικας περνάει από το στάδιο *προ-επεξεργασίας του πηγαίου κώδικα* (Source Code pre-Process), όπου λαμβάνει χώρα ο διαχωρισμός των keywords της γλώσσας προγραμματισμού από τα αναγνωριστικά (tokens). Επιπλέον, αναλύονται και παρουσιάζονται οι σχέσεις μεταξύ των οντοτήτων του κώδικα. Το αποτέλεσμα αυτού του σταδίου, κυρίως οι λέξεις που αντιστοιχούν σε οντότητες του κώδικα, είναι απαραίτητο για το επόμενο που σχετίζεται με την επεξεργασία των αναφορών αστοχιών.



Εικόνα 5.1: Σχηματική αναπαράσταση της αρχιτεκτονικής του συστήματος

Οι αναφορές σφαλμάτων είναι αδύνατο να εισαχθούν στο σύστημα με τη μορφή που έχουν όταν τις ανακτούμε από το Bugzilla. Για το λόγο αυτό, στο στάδιο *προ-επεξεργασίας των αναφορών* (Bug Report pre-Process) πραγματοποιείται μία αρχική επεξεργασία των bug reports προκειμένου να καταστούν κατάλληλες για μετέπειτα χρήση στο σύστημα.

Όταν ολοκληρωθούν τα δύο αυτά στάδια, σειρά έχει η *κυρίως επεξεργασία* των δεδομένων (Main Process). Σε αυτό το στάδιο είναι σημαντικό να έχει ήδη δημιουργηθεί το αρχείο με τις σχέσεις των οντοτήτων του πηγαίου κώδικά καθώς και να έχουν επιλεγεί οι αναφορές αστοχιών που θα χρησιμοποιηθούν σαν είσοδο στο κυρίως σύστημα. Η έξοδος αυτού του συστήματος, που είναι και το τελικό του αποτέλεσμα, είναι ένα αρχείο, το οποίο παρουσιάζει τις μεθόδους του προγράμματος λογισμικού που εξετάζουμε, στις οποίες έχει αποδοθεί ένα σκορ, ανάλογα με την πιθανότητα που έχουν να ευθύνονται για την αστοχία.

Στις επόμενες παραγράφους αναλύονται λεπτομερώς όλα τα στάδια και γίνεται εκτενής παρουσίαση για καθένα από τα επιμέρους βήματά τους.

5.1 Στάδιο προ-επεξεργασίας του πηγαίου κώδικα

Το στάδιο προ-επεξεργασίας πηγαίου κώδικα (*Source Code pre-Process*) αποτελείται από δύο βασικά επιμέρους βήματα: την ανάκτηση του κώδικα του προγράμματος λογισμικού από το Git repository και την εκτέλεση του FETCH Tool.

Το Git είναι ένα κατανεμημένο σύστημα ελέγχου εκδόσεων ανοιχτού κώδικα (distributed version control system – DVCS) που σημαίνει ότι επιτρέπει την παράλληλη εργασία πολλαπλών ατόμων σε ένα έργο, ακόμη και χωρίς την ύπαρξη σύνδεσης σε κεντρικό δίκτυο αφού η εργασία τους μπορεί να καταχωρηθεί (push) όταν είναι έτοιμη. Επειδή τα προγράμματα που έχουν επιλεγεί για αυτή την εργασία είναι όλα ανοιχτού κώδικα (open source) και ο πηγαίος κώδικάς τους υπάρχει στο Git, το πρώτο βήμα είναι να αποκτηθεί πρόσβαση στον κώδικα και στα αρχεία του.

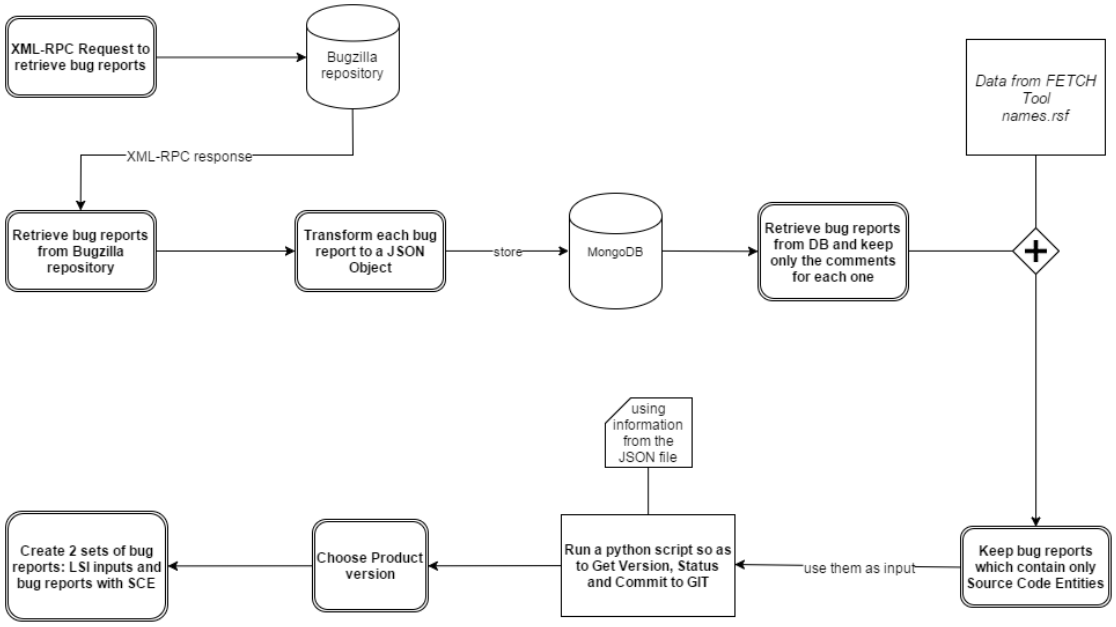
Το FETCH Tool είναι ένα εργαλείο που αναλύθηκε σε προηγούμενο κεφάλαιο, το οποίο δέχεται ως είσοδο τον πηγαίο κώδικα κάποιου προγράμματος και δίνει ως έξοδο τρία αρχεία τύπου rsf. Στο ένα αρχείο, που ονομάζεται “names.rsf”, υπάρχουν όλες οι οντότητες του κώδικα (**S**ource **C**ode **E**ntities – SCE) και επιπλέον αναγράφεται και ο τύπος τους. Το αρχείο με το όνομα του προγράμματος περιέχει όλες τις σχέσεις μεταξύ των οντοτήτων αυτών και τέλος το αρχείο με το όνομα “final.rsf” το οποίο είναι συνδυασμός των δύο προηγούμενων.

Συνεπώς, κύριος σκοπός του σταδίου προ-επεξεργασίας του πηγαίου κώδικα είναι η δημιουργία των τριών αυτών αρχείων από τα οποία παίρνουμε πληροφορίες τόσο για τις SCE όσο και για τις σχέσεις που τις συνδέουν. Τα tokens που αντιστοιχούν σε οντότητες χρησιμοποιούνται στο επόμενο στάδιο, ενώ αντίστοιχα οι σχέσεις μεταξύ αυτών είναι απαραίτητες για το στάδιο της επεξεργασίας των δεδομένων.

5.2 Στάδιο προ-επεξεργασίας αναφορών αστοχιών

Επόμενο στάδιο αποτελεί η επεξεργασία των αναφορών αστοχιών (*Bug Report pre-Process)*. Για να πραγματοποιηθεί αυτό, πρώτο και σημαντικό βήμα είναι η ανάκτηση των αναφορών αστοχιών από το Bugzilla repository, το οποίο είναι μία βάση δεδομένων του Bugzilla που αναλαμβάνει την αποθήκευση των αναφορών που καταχωρούνται από διάφορους χρήστες.

Αντίγραφα των αναφορών αστοχιών που βρίσκονται αποθηκευμένα στη βάση, χρειαζόμαστε τοπικά στο δικό μας μηχάνημα και ένας τρόπος για να γίνει αυτό είναι να τα «ζητήσουμε» από το Bugzilla με ένα XML-RPC request[1]. Το Bugzilla παρέχει ένα XML-RPC API το οποίο διευκολύνει τη διαδικασία αυτή και χρησιμοποιήθηκε για την ανάκτηση των αναφορών αστοχιών καθώς είναι ανεξάρτητο της πλατφόρμας, γεγονός που επιτρέπει ξεχωριστές εφαρμογές να επικοινωνούν μεταξύ τους.



Εικόνα 5.2: Bug Report pre-Process

Το endpoint για τα WebServices του Bugzilla είναι ένα xmlrpc.cgi αρχείο που δημιουργείται κατά την εγκατάσταση του. Έτσι για παράδειγμα, όταν το Bugzilla βρίσκεται στο bugzilla.yourdomain.com, ο XML-RPC client θα πρέπει να έχει πρόσβαση στο API μέσω της URL http:// bugzilla.yourdomain.com/xmlrpc.cgi.

Η εφαρμογή που δημιουργήθηκε για να εξυπηρετήσει τη συγκεκριμένη ανάγκη προσφέρει δύο επιλογές: την αποστολή αιτήματος για συγκεκριμένα bug reports χρησιμοποιώντας το αναγνωριστικό τους (id) ή τον ορισμό του αρχικού και τελικού id προκειμένου η απάντηση του αιτήματος να περιλαμβάνει όλα τα ενδιάμεσα bug reports. Η πρώτη περίπτωση διευκολύνει τον χρήστη όταν γνωρίζει συγκεκριμένα ids

[1] Το XML-RPC είναι ένα πρωτόκολλο το οποίο χρησιμοποιεί XML μηνύματα για να εκτελέσει απομακρυσμένες κλήσεις ρουτινών (**R**emote **P**rocedure **C**alls – RPC). Πιο συγκεκριμένα χρησιμοποιεί XML για να κωδικοποιήσει τις αιτήσεις, οι οποίες στέλνονται με HTTP μηχανισμό.

για τις αναφορές που χρειάζεται ενώ η δεύτερη δίνει στο αίτημα ένα εύρος αναφορών.

Αφότου έρθει η απάντηση από το XML-RPC request, πρέπει να υποστεί την απαραίτητη επεξεργασία προκειμένου να μετατραπεί σε κατάλληλη μορφή, αποδεκτή για την αποθήκευση σε μία τοπική Mongo Database. Αυτό σημαίνει ότι πρέπει να διατηρηθούν όλα τα πεδία που είναι συμπληρωμένα στην αναφορά σφάλματος και ταυτόχρονα να συνδυαστούν για να δημιουργηθεί ένα JSON object που θα αποθηκευτεί στη MongoDB[2].

Η χρήση της τοπικής Mongo βάσης δεδομένων δεν ήταν απαραίτητη, παρ' όλα αυτά όμως διευκολύνει την αποθήκευση και ανάκτηση των αναφορών αστοχιών ξεχωριστά για καθένα από τα προγράμματα λογισμικού που αναλύθηκαν σε αυτό το σύστημα.

Σε όλες τις αναφορές σφαλμάτων είναι απαραίτητο να περιγράφεται με αναλυτικό τρόπο το πρόβλημα που παρουσιάστηκε, ώστε να είναι σε θέση κάποιος να το αναπαράγει και να ασχοληθεί με την επίλυσή του. Μία συνοπτική περιγραφή υπάρχει στο πεδίο Summary, ενώ εκτενέστερη συναντά κανείς στα σχόλια της αναφοράς. Γίνεται κατανοητό λοιπόν, ότι για το bug localization είναι απαραίτητο να επεξεργαστούμε τα κείμενα που καταχωρούνται ως σχόλια σε μία αναφορά αστοχίας, καθώς είναι εκείνα που μας οδηγούν στο σφάλμα που διαπιστώθηκε στο συγκεκριμένο πρόγραμμα λογισμικού. Οι υπόλοιπες πληροφορίες που υπάρχουν δεν επικεντρώνονται σε σημεία του κώδικα επομένως δεν διευκολύνουν την εύρεση του ελαττωματικού τμήματός του.

Σύμφωνα με τα προαναφερθέντα, για κάθε εφαρμογή λογισμικού που αναλύουμε, ανακτούμε από τη βάση όλες τις αναφορές αστοχιών που σχετίζονται με εκείνη και στη συνέχεια τις επεξεργαζόμαστε προκειμένου να δημιουργήσουμε αρχεία – ένα για κάθε αναφορά — τα οποία θα περιέχουν αυτή τη φορά μόνο τα σχόλια των χρηστών για το συγκεκριμένο πρόβλημα.

Τα κείμενα των χρηστών όπως είναι λογικό είναι γραμμένα σε φυσική γλώσσα, γεγονός που δημιουργεί μία δυσκολία στην επεξεργασία των δεδομένων. Ιδανικά θα έπρεπε να δημιουργηθεί ένα σύστημα, το οποίο θα είναι σε θέση να κατανοεί τη σημασία κάθε φράσης της φυσικής γλώσσας. Κάτι τέτοιο θα ήταν δυνατό να πραγματοποιηθεί με εργαλεία επεξεργασίας φυσικής γλώσσας, αλλά εξαιτίας της μεγάλης πολυπλοκότητάς του δεν αποτελεί μέρος του συστήματος που περιγράφεται.

Ως εναλλακτική λύση και δεδομένου ότι ο τελικός στόχος είναι ένα σύστημα το οποίο θα προσδιορίζει τα σημεία του πηγαίου κώδικα στα οποία οφείλονται οι αστοχίες, πρέπει να συσχετιστούν τα σχόλια των αναφορών που έχουμε στη διάθεσή μας, με τον κώδικα. Κάτι τέτοιο επιβάλλει ενός είδους διαλογή στο σύνολο των σχολίων προκειμένου να μείνουν εκείνα που περιέχουν source code entities ή αλλιώς λέξεις του πηγαίου κώδικα.

Στο σημείο αυτό αξίζει να σημειωθεί η διαδικασία με την οποία έγινε η αναζήτηση και η ταυτοποίηση των source code entities, καθώς πέρασε από πολλά

[2] Η MongoDB είναι ίσως η δημοφιλέστερη εγγραφοκεντρική βάση δεδομένων. Το αντίστοιχο record των σχεσιακών βάσεων, σε αυτή τη βάση είναι το έγγραφο (document) και περιέχει πεδία που απαρτίζονται από ζεύγη κλειδιών-τιμών (key-value pairs) παρόμοια με αντικείμενα της μορφής JSON.

στάδια υλοποίησης αλλά και βελτιστοποίησης. Το εργαλείο FETCH, όπως έχει ήδη αναφερθεί, μας παρέχει ένα αρχείο “names.rsf”, το οποίο περιέχει όλες τις λέξεις του πηγαίου κώδικα. Χρησιμοποιώντας το αρχείο αυτό υλοποιήθηκαν κάποιες μέθοδοι για την αναζήτηση των source code entities που υπάρχουν μέσα σε κάθε bug report, μερικές από τις οποίες ενώ έδιναν σωστά αποτελέσματα, υστερούσαν σε ταχύτητα, ενώ άλλες παρουσίαζαν κάποια σημαντικά μειονεκτήματα και κατά συνέπεια δεν ήταν δυνατό να εφαρμοστούν.

Αρχική ιδέα για τη διαδικασία ταυτοποίησης των λέξεων του κώδικα ήταν η υλοποίηση αναζήτησης με hash tree. Από το αρχείο που περιέχονται όλες οι λέξεις, δημιουργήθηκε ένα hash tree, κάθε κόμβος του οποίου είναι μία λέξη του κώδικα. Έπειτα για κάθε μία λέξη της αναφοράς αστοχίας, αναζητούμε στο δέντρο αν ταιριάζει με κάποια από τα source code entities. Αν και η αναζήτηση σε ένα hash tree πραγματοποιείται με μεγάλη ταχύτητα, η μέθοδος αυτή έχει ένα σημαντικό μειονέκτημα. Για να ταυτοποιηθεί μία λέξη ως λέξη κώδικα, πρέπει στη αναφορά σφάλματος να εμφανίζεται ακριβώς όπως στο αρχείο “names.rsf”, διαφορετικά η αναζήτηση αποτυγχάνει. Όμως είναι συχνό το φαινόμενο ένας χρήστης να αναφέρεται σε μία μέθοδο χωρίς να χρησιμοποιεί ολόκληρο το όνομά της, αλλά μόνο το Signature της. Παράδειγμα αυτού αποτελεί η αναφορά στη μέθοδο method.foo(), χρησιμοποιώντας για λόγους συντομίας το όνομα foo().

Αυτό το πρόβλημα μπορεί να επιλυθεί αναζητώντας τη λέξη ή substring αυτής απευθείας στο αρχείο του FETCH Tool, πράγμα το οποίο θα χρειαζόταν πολλές ώρες να ολοκληρωθεί για όλες τις αναφορές αστοχιών. Για την επιτάχυνση της διαδικασίας ενώθηκαν όλα τα source code entities σε ένα ενιαίο string που περιέχει το σύμβολο του δολαρίου ($) στα σημεία στα οποία διαχωρίζονται οι λέξεις. Η αναζήτηση λοιπόν πραγματοποιείται προσπαθώντας να αντιστοιχιστεί κάθε λέξη του bug report με κάποιο substring της συμβολοσειράς αυτής. Αν αυτό επιτευχθεί, γεγονός που υποδεικνύει ότι η λέξη είναι μία source code entity, στόχος είναι να εισάγουμε στο σύνολο που δημιουργούμε το ολοκληρωμένο όνομά της και όχι τη συντόμευσή του. Συνεπώς, το επόμενο βήμα είναι να εντοπιστεί το δολάριο πριν και μετά τη λέξη ώστε να κρατήσουμε ό,τι υπάρχει ανάμεσά τους και να το εισάγουμε στο σύνολό μας.

Παρ’ όλο που το παραπάνω αποτελεί σημαντική βελτίωση ως προς την αναζήτηση απευθείας στο αρχείο του FETCH Tool, επιπλέον τροποποιήσεις πραγματοποιήθηκαν ώστε η εκτέλεσή του να απαιτεί ακόμη λιγότερο χρόνο. Έτσι δημιουργήθηκε μία αντιστοίχιση σε μορφή json, την οποία θα μπορούσαμε να χαρακτηρίσουμε ως ένα είδους μνήμης και περιέχει τη λέξη του bug report ως κλειδί (key) και τις λέξεις του πηγαίου κώδικα που αντιστοιχούν σε αυτή ως τιμές (values). Για καθεμία λέξη της αναφοράς σφάλματος, εξετάζουμε πρώτα αν έχει καταχωρηθεί στο json αντιστοίχισης και εφόσον δεν υπάρχει, αναζητούμε τη λέξη στο ενιαίο string με τον τρόπο που περιγράφηκε προηγουμένως. Η διαδικασία αυτή εξοικονομεί χρόνο σε περιπτώσεις που αναζητούμε την ίδια λέξη περισσότερες από μία φορές, καθώς δεν χρειάζεται να πραγματοποιείται η ίδια αναζήτηση σε ολόκληρη τη συμβολοσειρά.

Κατά συνέπεια, έπειτα από το βήμα αυτό, έχουμε ένα σύνολο από αρχεία, καθένα από τα οποία περιέχει λέξεις του κώδικα, όχι keywords αλλά ονόματα μεταβλητών, αρχείων, μεθόδων, συναρτήσεων και άλλα. Κάθε αρχείο αντιστοιχεί σε ένα bug report του οποίου το id περιέχεται στο όνομα του.

Σημαντική προϋπόθεση για να δώσει το σύστημα ορθή έξοδο αποτελεί η δυνατότητα συσχέτισης, μέσω μίας διαδικασίας που θα περιγραφεί στο επόμενο στάδιο, της αναφοράς αστοχίας που θα θέσουμε ως είσοδο, με άλλες ήδη καταχωρημένες αναφορές. Για το σκοπό αυτό χρειαζόμαστε μία έκδοση του προγράμματος λογισμικού στην οποία να υπάρχει ικανοποιητικός αριθμός αναφορών σφαλμάτων προκειμένου να είναι εφικτή η συσχέτιση με μεγαλύτερη ακρίβεια. Στη λήψη απόφασης για το ποια έκδοση του κώδικα θα αναλυθεί συμβάλλει το επόμενο βήμα, στο οποίο χρησιμοποιώντας τα αρχεία που έχουν δημιουργηθεί, παίρνουμε πληροφορίες για το Status και τη Version κάθε αναφοράς, καθώς και για το αν υπάρχει ή όχι Commit στο Git που συνδέεται με το συγκεκριμένο bug id. Τα δύο πρώτα (status και version) είναι εύκολο να βρεθούν από το αρχείο που ήδη έχουμε. Για το commit στο Git όμως απαιτείται μία άλλη διαδικασία, η οποία περιλαμβάνει την αποστολή ενός request στο online Git του προϊόντος με τις κατάλληλες παραμέτρους (για παράδειγμα το bug id). Αν υπάρχει commit στο Git για το συγκεκριμένο bug, τότε στην απάντηση (response) εμφανίζεται ένα pattern το οποίο αναγνωρίζει ο αλγόριθμός μας και καταγράφει την ύπαρξη commit. Τα commits στο Git είναι ιδιαίτερα χρήσιμα για το στάδιο αξιολόγησης του συστήματος, αφού περιέχουν πληροφορίες για τα σημεία του κώδικα που έχουν τροποποιηθεί προκειμένου να διορθωθεί το bug.

Έτσι λοιπόν γνωρίζοντας για πόσες αναφορές αστοχιών με status “Fixed” υπάρχει commit και σε ποια έκδοση κώδικα αναφέρεται αυτό, μπορούμε να αποφασίσουμε ποιον πηγαίο κώδικα θα χρησιμοποιήσουμε. Σε περίπτωση που ο κώδικας που αναλύθηκε με το FETCH Tool δεν είναι ο επιθυμητός, τρέχουμε πάλι το FETCH με τη σωστή version κώδικα και επαναλαμβάνουμε τη διαδικασία. Αυτό είναι και το βήμα της επιλογής πηγαίου κώδικα του προγράμματος λογισμικού.

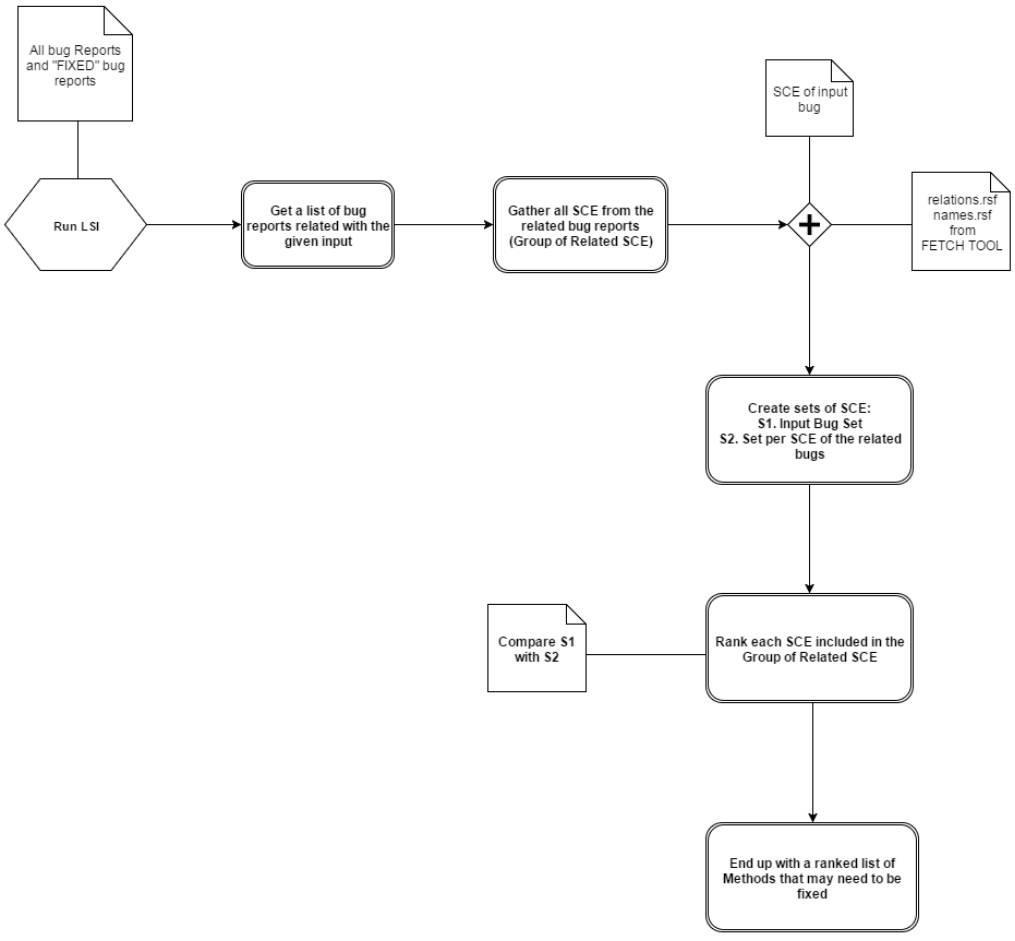
Από τα bug reports τα οποία περιέχουν λέξεις κώδικα (source code entities) δημιουργούμε δύο σύνολα. Το ένα αποτελείται από τις αναφορές που είναι “Fixed” και θα χρησιμοποιηθούν ως input για το επόμενο στάδιο και το άλλο από όλα τα bug reports στα σχόλια των οποίων εμφανίζονται source code entities. Το δεύτερο σύνολο αποτελείται από τις αναφορές αστοχιών με τις οποίες θα συσχετιστεί η αναφορά εισόδου του LSI.

5.3 *Στάδιο επεξεργασίας των δεδομένων*

Το στάδιο επεξεργασίας δεδομένων (Main Process) ξεκινά μόλις τελειώσει το στάδιο προ-επεξεργασίας των αναφορών αστοχιών στο οποίο δημιουργούνται τελικά τα δύο σύνολα που περιγράφηκαν προηγουμένως. Επόμενο βήμα λοιπόν είναι η εφαρμογή της τεχνικής LSI, η οποία θα δημιουργήσει συσχετίσεις μεταξύ όλων των αναφορών αστοχιών και της αναφοράς που θέτουμε ως είσοδό της.

Όπως έχει ήδη αναφερθεί στο κεφάλαιο που αναλύεται το θεωρητικό υπόβαθρο, η τεχνική LSI δημιουργεί clusters από αναφορές αστοχιών, τα οποία έχουν συγκεκριμένες συντεταγμένες. Για να συμβεί αυτό, σε καθένα bug report αντιστοιχίζεται ένα διάνυσμα, το οποίο αντιπροσωπεύει τις συντεταγμένες του στο χώρο εννοιών (concept space). Αναφορές αστοχιών με γειτονικές συντεταγμένες αποτελούν ένα cluster.

Το αποτέλεσμα του LSI είναι μία ταξινομημένη λίστα με τις υπόλοιπες αναφορές αστοχιών του υπό μελέτη συστήματος οι οποίες σχετίζονται εννοιολογικά με την αναφορά εισόδου (input bug report). Η λίστα αυτή δημιουργείται θέτοντας κάποιες παραμέτρους στον υπολογισμό ομοιότητας και δίνει μία εικόνα σχετικά με την ομοιότητα του σφάλματος εισόδου με προηγούμενα ήδη καταχωρημένα σφάλματα.



Εικόνα 5.3: Στάδιο επεξεργασίας των δεδομένων

Για τον διαχωρισμό των αναφορών αστοχιών σε clusters είναι αναγκαίο να χρησιμοποιηθεί μία μέθοδος σύγκρισης των συντεταγμένων των bug reports, δηλαδή των αντιπροσωπευτικών τους διανυσμάτων. Η μέθοδος αυτή είναι η ομοιότητα συνημίτονου (cosine similarity). Συνεπώς τίθεται μία τιμή ως κατώφλι (threshold) με την οποία συγκρίνεται ο βαθμός ομοιότητας κάθε αναφοράς έτσι ώστε να πραγματοποιηθεί μία διαλογή εκείνων που σχετίζονται περισσότερο με το περιεχόμενο της αναφοράς εισόδου.

Το κατώφλι αυτό είναι μία μεταβλητή που εξαρτάται από κάθε σύστημα και είναι σπάνιο η ίδια τιμή να ταιριάζει σε όλα. Αυτό σημαίνει ότι κάθε σύστημα το οποίο μελετάται, έχει διαφορετικό κατώφλι έτσι ώστε το αποτέλεσμα του LSI να περιλαμβάνει ικανοποιητικό αριθμό αναφορών, οι οποίες θα αναλυθούν σε επόμενο βήμα. Το πλεονέκτημα της μεταβλητότητας της τιμής αυτής είναι ότι ο προγραμματιστής θα μπορεί να θέτει όποια τιμή θεωρεί εκείνος καταλληλότερη αναλόγως με το αποτέλεσμα που θέλει να έχει ή ακόμη και για να βελτιώσει ήδη υπάρχον αποτέλεσμα.

Παρ' όλα αυτά όμως είναι πιθανό να μην υπάρχει κανένα σχετικό σφάλμα για το κατώφλι που έχει οριστεί ή ακόμη και ο αριθμός των αναφορών που συσχετίζονται με την αναφορά εισόδου να είναι αρκετά μικρός. Σε αυτή την περίπτωση στο περιβάλλον που έχει αναπτυχθεί υπάρχει η δυνατότητα να ορίσει κανείς το πλήθος, έστω x, των related bug reports που επιθυμεί να περιλαμβάνονται στη λίστα. Οι αναφορές αστοχιών του συστήματος κατατάσσονται με βάση το βαθμό ομοιότητάς τους και εισάγονται στη λίστα εκείνες που ο βαθμός τους ικανοποιεί τη σχέση:

$$bug\ report\ similarity\ > similarity\ x\text{–}\iota o\sigma\tau o\acute{\upsilon}\ bug\ report$$

Αφού έχει δημιουργηθεί η λίστα με τις αναφορές που σχετίζονται εννοιολογικά με το σφάλμα που περιγράφεται στην αναφορά εισόδου, σειρά έχει η επεξεργασία τους. Στο στάδιο προ-επεξεργασίας των αναφορών σφαλμάτων έχει πραγματοποιηθεί η διαλογή των λέξεων σε λέξεις φυσικής γλώσσας και λέξεις κώδικα (Source Code Entities). Έχοντας ήδη λοιπόν το σύνολο που περιλαμβάνει source code entities που υπάρχουν στην αναφορά σφάλματος εισόδου, δημιουργείται σε αυτό το στάδιο ένα άλλο σύνολο, το οποίο αποτελείται από τα source code entities εκείνα που συναντώνται σε όλα related bug reports.

Το σύνολο των λέξεων πηγαίου κώδικα της αναφοράς εισόδου είναι πιθανό να περιέχει κάποιες λέξεις κώδικα οι οποίες δεν είναι ιδιαίτερα σημαντικές με αποτέλεσμα να μην προσδίδουν χρήσιμη πληροφορία στην ανάλυσή μας, όπως για παράδειγμα η λέξη public ή το όνομα του υπό μελέτη συστήματος. Για να αποκλειστούν οι λέξεις αυτές χρησιμοποιήθηκε μία μετρική η οποία δίνει ένα σκορ σε καθεμία λέξη του συνόλου με βάση τις φορές που εκείνη εμφανίζεται στις αναφορές σφαλμάτων που έχουν προκύψει από το LSI και υπολογίζεται από τη σχέση:

$$\log \frac{\#\ related\ bug\ reports}{\#\ related\ bug\ reports\ with\ the\ SCE}$$

Έπειτα από την εφαρμογή της παραπάνω σχέσης σε όλες τις λέξεις του συνόλου, πρέπει να οριστεί ένα ακόμη κατώφλι σύμφωνα με το οποίο θα ελέγχεται αν η λέξη

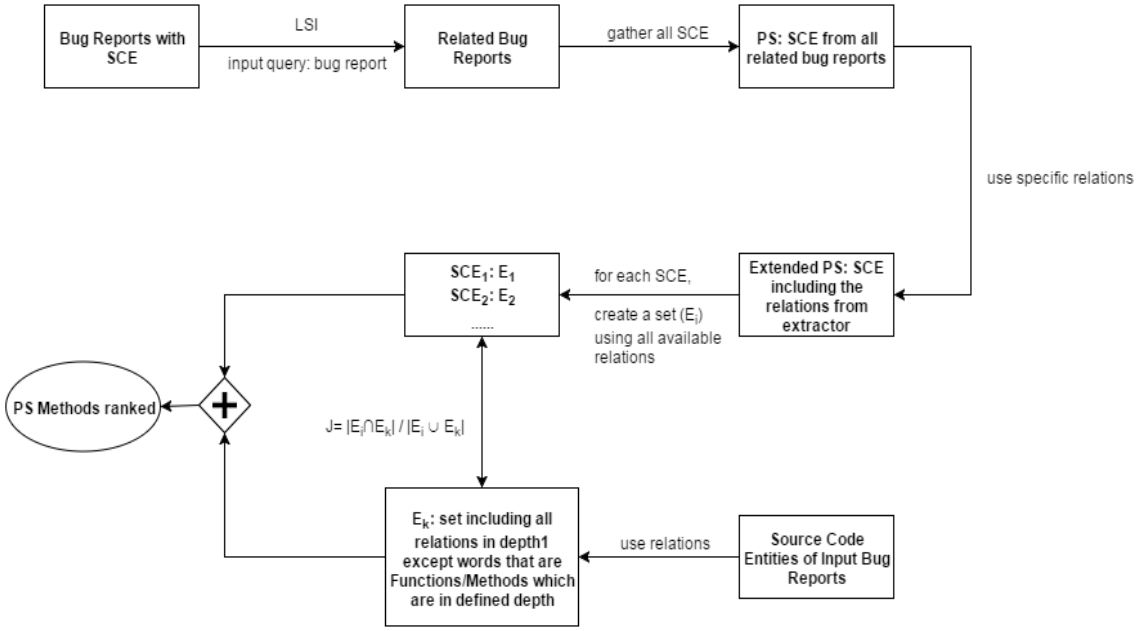
είναι σημαντική ή όχι. Αυτό το κατώφλι όπως και εκείνο του LSI εξαρτάται από το σύστημα το οποίο αναλύεται καθώς και από το πλήθος των λέξεων που επιθυμεί ο χρήστης να αποκλείσει. Επομένως διατηρούνται οι λέξεις εκείνες που πληρούν την παρακάτω προδιαγραφή:

$$importance\ threshold\ \leq SCE\ rank$$

Εξαίρεση αποτελούν οι λέξεις που δεν εμφανίζονται σε καμία από τις αναφορές που σχετίζονται με εκείνη της εισόδου. Για αυτές η τιμή #related bug reports with SCE ισούται με μηδέν (0) και δεν μπορεί να πραγματοποιηθεί η διαίρεση του λογάριθμου. Για το λόγο αυτό το σκορ τους τίθεται εξ' αρχής μηδέν (0) και επιπλέον φροντίζουμε ώστε να μην διαγραφούν από το τελικό σύνολο. Αν καμία λέξη δεν ικανοποιεί την παραπάνω συνθήκη, τότε ως κατώφλι ορίζεται ο μέσος όρος των λογαρίθμων των λέξεων και διατηρούνται πλέον εκείνες που ικανοποιούν τη σχέση:

$$average\ SCE\ log\ \leq SCE\ rank$$

Συνδυάζοντας τις πληροφορίες που βρίσκονται τόσο στο αρχείο "final.rsf", στο σύνολο SCE της αναφοράς εισόδου –μετά την εφαρμογή του λογάριθμου– όσο και στο σύνολο SCE όλων των related bug reports, δημιουργούνται μεγαλύτερα σύνολα, τα οποία περιέχουν πλέον και τις σχέσεις μεταξύ των SCE. Η διαδικασία αυτή παρουσιάζεται αναλυτικά στο διάγραμμα που ακολουθεί.



Εικόνα 5.4: Δημιουργία επιμέρους συνόλων για ranking των SCE

Έχει αναφερθεί ότι μόλις ολοκληρωθεί η εφαρμογή της τεχνικής LSI, συγκεντρώνονται σε ένα αρχείο όλες οι λέξεις πηγαίου κώδικα που συναντώνται στις αναφορές αστοχιών οι οποίες έχουν συσχετιστεί σημασιολογικά με την αναφορά εισόδου. Το σύνολο αυτών των SCE, ονομάζεται *Problem Set (**PS**)*, περιλαμβάνει όχι μόνο μεθόδους, αλλά και μεταβλητές, αρχεία ή ακόμη και κλάσεις. Στο σημείο αυτό θα χρησιμοποιηθούν οι σχέσεις που συνδέουν τα SCE μεταξύ τους και θα δημιουργηθεί μία επέκταση του PS ***(Extended PS)*** στην οποία θα εμπεριέχονται περισσότερα SCE σύμφωνα με τον πίνακα που ακολουθεί:

SCE Type	Relation used for PS
GlobalVar	Accesses
Attribute	Sets
File	DefinedIn, DeclaredIn
Class	MethodBelongsToClass
Method	Calls
Function	Calls

Πίνακας 5.1: Σχέσεις που χρησιμοποιούνται για την δημιουργία του Problem Set (PS)

Έτσι για κάθε τύπο Source Code Entity χρησιμοποιήθηκαν οι παραπάνω σχέσεις, σε επίπεδο in αλλά και out[3] και δεδομένου του ότι η προσέγγισή του συστήματος έχει ως τελικό σκοπό να παρουσιάσει τις μεθόδους του κώδικα που χρειάζονται τροποποίηση, όταν ο τύπος του SCE ή και της λέξης που προκύπτει μέσα από τις σχέσεις είναι Method ή Function, τη σχέση Call την παίρνουμε σε βάθος, το οποίο ορίζεται από τον προγραμματιστή. Όπως γίνεται αντιληπτό, πραγματοποιήθηκε μία διαλογή στις σχέσεις των SCE, η οποία αποτελεί σημαντικό θέμα αφού πλεονάζουσα πληροφορία πιθανόν να προσθέτει «θόρυβο» στα δεδομένα μας και να παρεμβαίνει την ακρίβεια του τελικού αποτελέσματος.

Επόμενο στάδιο είναι μία περαιτέρω ανάλυση καθεμίας λέξης που περιλαμβάνεται στο Extended PS, δημιουργώντας ένα σύνολο (E_i) που αντιστοιχεί σε αυτή. Εφόσον ο τύπος της SCE είναι Method ή Function, τότε αξιοποιείται η σχέση Calls προκειμένου να εισαχθούν στο σύνολο E_i και άλλες μέθοδοι ή συναρτήσεις που καλούνται από το συγκεκριμένο SCE (σχέση out), αλλά ακόμη και εκείνες που καλούν το SCE (σχέση in). Εκτός όμως από τη σχέση Calls, στα αρχεία που δημιουργεί extractor (FETCH Tool) δηλώνονται και άλλες σχέσεις, τις οποίες χρησιμοποιούμε σε βάθος ένα (1), εξαιρώντας τις σχέσεις Signature και Visibility διότι το αποτέλεσμά τους δεν προσδίδει περεταίρω πληροφορία στην ανάλυση.

[3] Του όρους in και out για τον προσδιορισμό των σχέσεων, τους δανειστήκαμε από τους ορισμούς του RDF. Για την καλύτερη απόδοσή τους χρησιμοποιούμε το σχήμα:

Ο κόμβος SCE 2 συνδέεται με τον SCE 1 με τη σχέση relation_one (σχέση IN), καθώς και με τον κόμβο SCE 3 με τη σχέση relation_two (σχέση OUT)

Επιπλέον εξαιρούνται από τα SCE που προκύπτουν στο E_i, όσα είναι Attributes ή επειδή η ανάλυση επικεντρώνεται στην εύρεση των μεθόδων του κώδικα που χρειάζονται τροποποίηση για να διορθωθεί το bug. Αντίστοιχο σύνολο (E_k), ακολουθώντας την ίδια διαδικασία, δημιουργείται συγκεντρωτικά και για όλες τις λέξεις πηγαίου κώδικα που υπάρχουν στην αναφορά σφάλματος εισόδου (input bug report).

Όπως αναφέρθηκε και στις προηγούμενες παραγράφους, όλα τα σύνολα που δημιουργήθηκαν (Extended PS, E_i και E_k) περιλαμβάνουν SCE τα οποία σχετίζονται με τα ήδη υπάρχοντα μέσα από τις σχέσεις του extractor. Έτσι λοιπόν οι λέξεις του πηγαίου κώδικα μπορούν να συνδυαστούν με τις σχέσεις αυτές και να δημιουργηθεί ένας κατευθυνόμενος γράφος, με κατευθύνσεις που ορίζονται από τις In και Out σχέσεις. Επομένως για να βρεθούν τα «συγγενικά» SCE ήταν αναγκαία η προσπέλαση του γράφου, κάτι το οποίο θα μπορούσε να πραγματοποιηθεί με δύο τρόπους: με την Cayley DB[4] ή με RDF γράφο. Η δημιουργία του κατευθυνόμενου γράφου και στις δύο περιπτώσεις πραγματοποιείται με τριπλέτες (για παράδειγμα SCE_1-relation-SCE_2), όπου relation είναι οι σχέσεις του extractor στις οποίες προστέθηκε και η σχέση "Is" για να δηλωθεί και ο τύπος κάθε SCE (SCE_1 Is Method). Έπειτα αναζητούμε κάθε SCE στο γράφο και ακολουθώντας τις κατευθύνσεις του, παίρνουμε τα υπόλοιπα SCE που εισάγονται στο σύνολο. Παρ' όλο που και οι δύο υλοποιήσεις εφαρμόστηκαν, τελικά επικράτησε εκείνη του RDF γράφου, λόγω αποδοτικότητας σε επίπεδο χρόνου.

Συνεπώς, για να αποδοθεί ένα βάρος σε κάθε SCE που θα αντικατοπτρίζει το πόσο πιθανό είναι να ευθύνεται εκείνο για το bug που περιγράφεται στην αναφορά αστοχίας, είναι απαραίτητο να συγκριθεί κάθε σύνολο E_i με το αντίστοιχο σύνολο του input bug report E_k. Για τη σύγκριση αυτή χρησιμοποιείται η μετρική ομοιότητας Jaccard:

$$J = \frac{|E_i \cap E_k|}{|E_i \cup E_k|}$$

Το αποτέλεσμά της αναγράφεται στο τελικό αρχείο, δίπλα στο όνομα της μεθόδου, ώστε ο χρήστης να μπορεί να κατανοήσει το βαθμό στον οποίο επηρεάζει η κάθε μέθοδος το σφάλμα που παρουσιάστηκε.

Σημείωση*: Για να διευκολύνουμε το κομμάτι της αξιολόγησης του συστήματός μας, δημιουργείται ένα ακόμη αρχείο με όνομα "Results-Methods" που περιέχει το όνομα της μεθόδου και δίπλα τη θέση της στο αρχείο "psRankMethods", στο οποίο περιλαμβάνεται η ταξινομημένη λίστα με τις μεθόδους που είναι πιθανό να χρειάζονται τροποποίηση για την επίλυση του σφάλματος.*

[4] Η Caley είναι μία open source βάση δεδομένων, η οποία δίνει τη δυνατότητα στο χρήστη όχι μόνο να αποθηκεύει γράφους, αλλά και να τους οπτικοποιεί όπως επίσης και να εκτελεί queries πάνω σε αυτούς. Χρησιμοποιείται κυρίως για Linked Data και υπάρχει ήδη υλοποιημένο API με το οποίο μπορεί ένα πρόγραμμα να εκτελεί queries στη βάση.

6

Εφαρμογή και Πειραματικά Αποτελέσματα

Στο κεφάλαιο αυτό θα παρουσιαστεί τόσο η εφαρμογή του συστήματος που δημιουργήθηκε στα πλαίσια της διπλωματικής εργασίας, όσο και τα αποτελέσματα που προέκυψαν από αυτή.

Το σύστημα εντοπισμού «ελαττωματικών» σημείων του κώδικα (bug localization) εφαρμόστηκε σε τέσσερα προϊόντα του KDE, για τα οποία υπάρχει η δυνατότητα ανάκτησης των αναφορών σφαλμάτων τους από το Bugzilla και ταυτόχρονα είναι προγράμματα ανοιχτού κώδικα (open source), γεγονός εξαιρετικά απαραίτητο, αφού όπως έχει ήδη αναφερθεί ο πηγαίος κώδικας αναλύεται και οι σχέσεις που προκύπτουν χρησιμοποιούνται για την εύρεση των «προβληματικών» μεθόδων. Τα προϊόντα αυτά επιλέχθηκαν μεταξύ άλλων εξαιτίας του πλήθους των bug reports που είχαν δημιουργηθεί στο Bugzilla. Αυτό σημαίνει ότι απορρίφθηκαν τα προϊόντα εκείνα τα οποία είτε είχαν μικρό αριθμό συνολικών bug reports, είτε τα πιο πρόσφατα bug reports τους ήταν αρκετά παλιά, πράγμα το οποίο υποδεικνύει ότι έχει περάσει αρκετός καιρός από την τελευταία φορά που συντηρήθηκαν. Τέλος, ένα βασικό χαρακτηριστικό τους είναι η γλώσσα προγραμματισμού. Το FETCH Tool έχει καλύτερα αποτελέσματα με γλώσσες όπως η C ή C++, γι' αυτό και στα προϊόντα που εφαρμόστηκε η τεχνική bug localization είναι γραμμένα στις γλώσσες αυτές.

6.1 Παρουσίαση Σταδίων του Συστήματος

Στο σημείο αυτό γίνεται μία περιληπτική παρουσίαση των σταδίων μέσα από τα οποία πραγματοποιήθηκε η επεξεργασία των δεδομένων και κατ' επέκταση η εφαρμογή του bug localization συστήματος. Εκτενέστερη ανάλυση προηγήθηκε στο κεφάλαιο 5, όπου παρουσιάστηκε λεπτομερώς η αρχιτεκτονική του εν λόγω συστήματος.

Αρχική και απαραίτητη διαδικασία είναι η ανάκτηση του πηγαίου κώδικα από το Git Repository, ώστε να εφαρμοστεί σε αυτόν το FETCH Tool και να δημιουργηθούν οι σχέσεις μεταξύ των οντοτήτων του κώδικα, που αναγράφονται στο αρχείο 'final.rsf'. Έπειτα σειρά έχει η εύρεση της κατάλληλης έκδοσης του προϊόντος που θα χρησιμοποιηθεί για την ανάλυση και πραγματοποιείται αφού έχουν ανακτηθεί όλα τα bug reports από το Bugzilla repository. Από αυτές τις αναφορές αστοχιών επιλέγονται μόνο εκείνες που περιέχουν σχόλια και από τις λέξεις που περιέχονται στα σχόλια γίνεται μία διαλογή προκειμένου να παραμείνουν εκείνες που είναι λέξεις κώδικα (Source Code Entities). Επομένως από τον συνολικό αριθμό αναφορών αστοχιών, τελικά κρατάμε εκείνες που στα σχόλιά τους περιέχουν SCE.

Σε αυτό το σημείο είναι σημαντικό να αναφερθεί ότι για αξιολογηθούν τα αποτελέσματα του συστήματος που δημιουργήθηκε, θα χρησιμοποιηθούν σαν είσοδοι σε αυτό bug reports για τα οποία γνωρίζουμε ήδη τις αλλαγές που πραγματοποιήθηκαν στον κώδικα (bug reports με status 'FIXED'). Κάθε αλλαγή συνοδεύεται και από ένα commit του κώδικα στο Git, με αποτέλεσμα να γνωρίζουμε ποια αναφορά σφαλμάτων επιλύει. Επειδή ακριβώς ο κώδικας έχει ήδη τροποποιηθεί, είναι πολύ πιθανό να υπάρχει κάποιο σχόλιο στην αναφορά, το οποίο θα ενημερώνει τους χρήστες για αυτές τις αλλαγές. Όμως το σχόλιο αυτό δε λαμβάνεται υπόψη μας στην ανάλυση και εξαιρείται από την αναφορά εισόδου, αφού αποκαλύπτει εκ των προτέρων τις μεθόδους ή συναρτήσεις που χρειάστηκαν τροποποίηση.

Έπειτα σε κάθε αναφορά εισόδου εφαρμόζουμε την τεχνική LSI, με την οποία παίρνουμε και ορισμένα διαγράμματα που αποκαλύπτουν τα clusters που δημιουργούνται μεταξύ των bug reports. Στη συνέχεια συγκεντρώνονται όλα τα SCE που έχουν προκύψει από τις αναφορές που συσχετίζονται με εκείνη της εισόδου και εφαρμόζοντας τις σχέσεις του FETCH Tool, κατασκευάζονται τα σύνολα που συγκρίνονται με σκοπό να προστεθεί σε κάθε μέθοδο/συνάρτηση ένα βάρος που εκφράζει το πόσο σημαντική θεωρείται για την επίλυση του προβλήματος που καταγράφεται στην αναφορά.

6.2 *Εφαρμογή και Παρουσίαση αποτελεσμάτων*

Στη συνέχεια ακολουθεί η παρουσίαση των αποτελεσμάτων που προέκυψαν από την εφαρμογή του συστήματος bug localization στα προϊόντα που επιλέχθηκαν, καθώς και ορισμένες γραφικές παραστάσεις ενδιάμεσων σταδίων.

6.2.1 *Εφαρμογή στο Amarok*

Το Amarok είναι μία εφαρμογή αναπαραγωγής μουσικής (music player software) ανοιχτού κώδικα (open source) και λειτουργεί εξίσου σε πλατφόρμες όπως Windows, Linux και UNIX. Παρ' όλο που αποτελεί μέρος του KDE project, υπάρχει η δυνατότητα αυτόνομης χρήσης του ανεξάρτητα από τα υπόλοιπα προγράμματα που περιλαμβάνονται στο KDE.

Μετά την ανάλυση του πηγαίου κώδικ από το FETCH Tool, συγκεντρώθηκαν όλες οι αναφορές σφαλμάτων του Amarok (number of bug reports retrieved from Bugzilla) και από εκείνες μόνον όσες είχαν σχόλια από χρήστες παρέμειναν στο σύνολο αναφορών που εξετάστηκαν (number of bug reports with comments). Μετά τον έλεγχο κάθε σχόλιου, διαχωρίζονται εκείνα που περιέχουν SCE (number of bug reports including SCE in comments) τα οποία χρησιμοποιούνται για τη δημιουργία clusters (number of bug reports that would be related with LSI input). Έτσι προέκυψε ο πίνακας που ακολουθεί για την έκδοση 2.8 του Amarok:

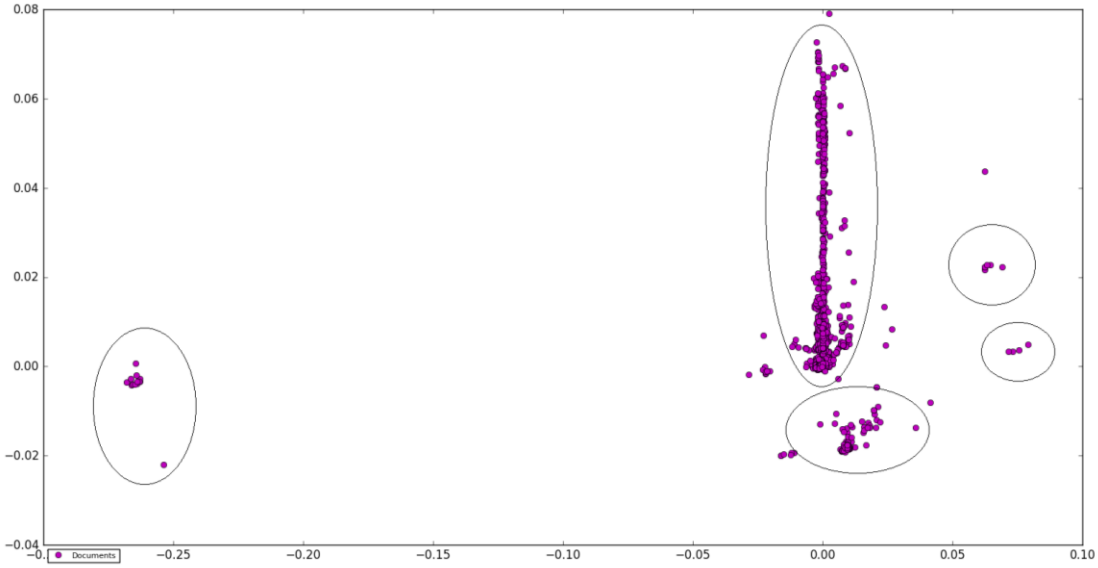
Number of bug reports retrieved from Bugzilla	17976
Number of bug reports with comments	17976
Number of bug reports including SCE in comments	7278
Number of bug reports that would be related with LSI input	7277

Πίνακας 6.1: Πλήθος των αναφορών αστοχιών που χρησιμοποιήθηκαν στα στάδια της προ-επεξεργασίας τους

Επόμενο βήμα είναι η εφαρμογή της τεχνικής LSI χρησιμοποιώντας ως είσοδο επτά (7) αναφορές σφαλμάτων για τις οποίες γνωρίζουμε τις μεθόδους που τροποποιήθηκαν στον κώδικα, προκειμένου να επιλυθεί το πρόβλημα που είχε εντοπιστεί στη λειτουργία του προϊόντος.

Με την τεχνική LSI επιτυγχάνεται ο διαχωρισμός των αναφορών αστοχιών σε clusters, δηλαδή ομάδες τέτοιες ώστε να περιλαμβάνουν τις αναφορές εκείνες οι οποίες σχετίζονται εννοιολογικά μεταξύ τους. Αυτή είναι εξάλλου και η επιπλέον πληροφορία που παρέχεται από την εφαρμογή του LSI. Το αποτέλεσμα είναι να μην συγκρίνονται απλά λέξεις με λέξεις, αλλά το σημασιολογικό περιεχόμενό τους. Τα clusters λοιπόν που δημιουργούνται προκύπτουν από ένα βάρος που αντιπροσωπεύει κάθε SCE, και κάποιες φορές είναι εύκολα διαχωρίσιμα ενώ κάποιες άλλες δεν είναι ευδιάκριτη η διαφοροποίησή τους.

Στην εικόνα φαίνονται τα clusters για τις αναφορές αστοχιών που περιείχαν SCE και χρησιμοποιήθηκαν στο LSI. Κάθε σημείο στο επίπεδο αντιπροσωπεύει ένα bug report και μπορεί κανείς εύκολα να παρατηρήσει ότι σχηματίζονται κάποιες ξεκάθαρες ομάδες, στις οποίες συγκεντρώνονται αρκετές αναφορές. Είναι αξιοσημείωτο όμως το γεγονός ότι το μεγαλύτερο μέρος των αναφορών σφαλμάτων φαίνεται να συγκεντρώνεται σε ένα σημείο του επιπέδου και να επεκτείνεται προς τα πάνω δημιουργώντας το μεγαλύτερο cluster. Φυσικά υπάρχουν και bug reports τα οποία «πλησιάζουν» κάποια clusters, χωρίς να ανήκουν μέσα σε αυτά.



Εικόνα 6.1: Clustering για τις αναφορές αστοχιών του Amarok

Στη συνέχεια και αφού δημιουργήθηκαν τα clusters, για κάθε αναφορά εισόδου έχουμε τα παρακάτω αποτελέσματα που αφορούν στο πλήθος των αναφορών σφαλμάτων που σχετίζονται με καθένα από τα bug reports (#related bug reports), στο πλήθος των SCE που σχετίζονται με το input (#related SCE) και τέλος στον αριθμό των SCE που προκύπτουν μετά την εφαρμογή των σχέσεων του .rsf αρχείου (#related SCE using code relations) :

Input Bug Reports (IDs)	**#related bug reports (LSI result)**	**#SCE in the input bug report**	**#related SCE (PS size)**	**#related SCE using code relations**
323156	352	32	1654	3594
323614	5196	21	5376	10497
323635	5190	10	5368	10463
325006	5199	2	5381	10501
328445	280	2	2585	5085
334479	5188	13	5317	10459
337725	323	4	3242	5797

Πίνακας 6.2: Στοιχεία μετά την εφαρμογή του LSI

Έπειτα δημιουργώντας τα σύνολα E_i για καθεμία λέξη που περιλαμβάνεται στο Extended Problem Set (Extended PS) και E_k για όλες τις λέξεις της αναφοράς εισόδου συνολικά, χρησιμοποιούμε την μετρική σύγκρισης συνόλων Jaccard ώστε να αποδοθεί ένα βάρος σε κάθε SCE. Η ταξινομημένη λίστα πού περιλαμβάνει όλα τα SCE περιέχεται στο αρχείο "*PSRankMethods.txt*" από το οποίο παίρνουμε τα εξής:

Input Bug Reports (IDs)	**Methods**	**Ranked position in the Sorted List**	**Methods in the Sorted List**
323156	SqlRegistry.getTrack(int)	86	3286
323614	MainWindow.slotShufflePlaylist()	6	8963
	Playlist::SortWidget.SortWidget(QWidget*)	5	
	Playlist::ViewUrlRunner.run(AmarokUrl)	7	
323635	Analyzer::Base.Base(QWidget)	118	8938
	BallsAnalyzer.BallsAnalyzer(QWidget*)	Not found	
	BlockAnalyzer.~BlockAnalyzer()	19	
	BlockAnalyzer.resizeEvent(QResizeEvent*)	8	
	BlockAnalyzer.paintEvent(QPaintEvent*)	1	
	BlockAnalyzer.drawBackground()	28	
325006	MainWindow.createMenus()	Not found	8968
	QScriptEngine::AmarokScript::ScriptImporter.loadQtBinding(QString)	1	
328445	ProgressWidget.updateTimeLabelTooltips()	14	4537
334479	ScriptableService.addTrack()	5	8926
337725	SearchQueryBias.toString()	1	5061

Πίνακας 6.3: Αποτελέσματα bug localization

Στον προηγούμενο πίνακα φαίνονται οι μέθοδοι του πηγαίου κώδικα του Amarok, οι οποίες τροποποιήθηκαν με σκοπό την επίλυση συγκεκριμένων αναφορών σφαλμάτων (input bug reports). Παρατηρώντας τα αποτελέσματα, διαπιστώνουμε ότι με την ανάλυση του bug localization συστήματος, η θέση των μεθόδων στην ταξινομημένη λίστα των SCE είναι αρκετά καλή, παρ' όλο που υπάρχουν ορισμένες μέθοδοι – δύο για την ακρίβεια – που δεν εντοπίστηκαν από το σύστημα. Το γεγονός αυτό οφείλεται είτε στο ότι δεν υπήρχε καμία σχέση (relation) από το FETCH Tool που να συσχέτιζε τις συγκεκριμένες μεθόδους με εκείνες που υπάρχουν τόσο στο

input bug report όσο και στα related bug reports (δηλαδή στο Extended PS), είτε στο ότι το βάθος που επιλέχθηκε για τη δημιουργία των συνόλων δεν ήταν ικανοποιητικό και επομένως χρειαζόταν ανάλυση σε μεγαλύτερο βάθος.

Πιο συγκεκριμένα για την αναφορά σφάλματος με id 325006, παρατηρούμε ότι οι δύο μέθοδοι, που τροποποιήθηκαν προκειμένου να επιλυθεί το σφάλμα, ανήκουν σε διαφορετικές κλάσεις και δεν προέκυψε κάποια σχέση από την ανάλυση του FETCH Tool σε αυτό το βάθος που να τις συνδέει μεταξύ τους. Συνεπώς, παρά το γεγονός ότι η μία από αυτές ανιχνεύτηκε από το σύστημά μας και μάλιστα το σκορ που της αποδόθηκε την κατέταξε στην υψηλότερη θέση, η δεύτερη δεν έχει εντοπιστεί ούτε και υπάρχει στην ταξινομημένη λίστα των μεθόδων.

Αντίστοιχο παράδειγμα αποτελεί για την αναφορά με id 323635 και η μέθοδος BallsAnalyzer. BallsAnalyzer (QWidget*). Για τη συγκεκριμένη όμως μέθοδο αξίζει να παρατηρηθεί ότι λεκτικά μοιάζει με την κλάση BlockAnalyzer, της οποίας αρκετές μέθοδοι έχουν υποστεί τροποποίηση. Επομένως θα μπορούσαμε να υποθέσουμε, επαληθεύοντας την υπόθεση αυτή και στα επόμενα προγράμματα λογισμικού, ότι σημαντικό ρόλο παίζει και η σημασιολογία του ονόματος του εκάστοτε SCE, καθώς SCE με παρόμοια λεκτική σημασιολογία εμφανίζουν αυξημένη πιθανότητα να χρειάζονται μαζί τροποποίηση προκειμένου να επιλυθεί κάποιο σφάλμα.

Τα δεδομένα που παρουσιάζονται στον επόμενο πίνακα (6.4) περιλαμβάνουν συμπληρωματικές πληροφορίες όπως το precision και το recall για κάθε bug report. Είναι αξιοσημείωτο το γεγονός ότι εκτός από τις δύο αναφορές αστοχιών που αναφέρθηκαν προηγουμένως, όλες οι υπόλοιπες μέθοδοι εντοπίστηκαν από το bug localization σύστημα, αλλά και κατατάχθηκαν σε υψηλές θέσεις στη λίστα των μεθόδων συγκριτικά με το μέγεθος της λίστας. Εξαίρεση αποτελούν οι μέθοδοι SqlRegistry.getTrack(int) – bug report 323156 και Analyzer::Base.Base(QWidget) – bug report 323635, οι οποίες εμφανίζονται σε χαμηλή θέση. Παρ’ όλα αυτά όμως βρίσκονται στο 2,63% και 1,32% αντίστοιχα των μεθόδων με το καλύτερο σκορ της λίστας με αποτέλεσμα το precision να έχει πολύ υψηλές τιμές.

Ο πίνακας στην επόμενης σελίδας παρουσιάζει τις μεθόδους που αποτελούν το resolution κάθε αναφοράς αστοχίας, ομαδοποιημένες με βάση τις αναφορές αστοχίας και ταυτόχρονα παρέχει και πληροφορία για τα ποσοστά του precision και recall καθενός bug report. Όσο αφορά τα ποσοστά του precision παρατηρούμε ότι σε πρώτη φάση είναι αρκετά χαμηλά. Αυτό ερμηνεύεται από τον μεγάλο αριθμό μεθόδων που ανασύρονται όταν λάβουμε υπόψη μας τις σχέσεις που τις συνδέουν μεταξύ τους (Methods in the Sorted List). Αντίθετα όμως το ποσοστό του recall είναι αρκετά υψηλό, με εξαίρεση την αναφορά 325006 για την οποία δεν εντοπίστηκε η μία εκ των δύο μεθόδων. Τα υψηλά αυτά νούμερα ήταν αναμενόμενα καθώς για όλες τις αναφορές αστοχιών, στο σύνολο των μεθόδων που ανακτήθηκαν περιέχονται και εκείνες που αποτελούν το resolution τους

Bug IDs	Methods	Ranked position in list / number of methods in list	Recall (%)	Precision (%)
323156	SqlRegistry.getTrack(int)	86/3286	100	0.0304
323614	MainWindow.slotShufflePlaylist()	6/8963	100	0.0335
	Playlist::SortWidget.SortWidget(QWidget*)	5/8963		
	Playlist::ViewUrlRunner.run(AmarokUrl)	7/8963		
323635	Analyzer::Base.Base(QWidget)	118/8938	83,33	0.0559
	BallsAnalyzer.BallsAnalyzer(QWidget*)	Not found		
	BlockAnalyzer.~BlockAnalyzer()	19/8938		
	BlockAnalyzer.resizeEvent(QResizeEvent*)	8/8938		
	BlockAnalyzer.paintEvent(QPaintEvent*)	1/8938		
	BlockAnalyzer.drawBackground()	28/8938		
325006	MainWindow.createMenus()	Not found	50	0.0112
	QScriptEngine::AmarokScript::ScriptImporter .loadQtBinding(QString)	1/8968		
328445	ProgressWidget.updateTimeLabelTooltips()	14/4537	100	0.022
334479	ScriptableService.addTrack()	5/8926	100	0.0112
337725	SearchQueryBias.toString()	1/5061	100	0.0198

Πίνακας 6.4: Precision και Recall για τα αποτελέσματα του bug localization

Μελετώντας τα αποτελέσματα, γίνεται αντιληπτό ότι τέτοιο πλήθος μεθόδων δεν είναι εύκολα διαχειρίσιμο, καθώς είναι δύσκολο για τον προγραμματιστή να εξετάσει περίπου 9000 μεθόδους, στη χειρότερη περίπτωση, προκειμένου να εντοπίσει εκείνη που ευθύνεται για το σφάλμα. Για το λόγο αυτό το bug localization σύστημά μας, επιστρέφει στο χρήστη μόνο ένα ποσοστό των μεθόδων, ώστε να είναι σε θέση να τις ελέγξει. Έπειτα και από αυτή τη διαλογή των μεθόδων, το πλήθος τους στο τελικό αρχείο που περιέχει την κατάταξή τους, ελαττώνεται σημαντικά. Η μείωση αυτή των δεδομένων έχει ως αποτέλεσμα το ποσοστό του precision να ανέβει αρκετά, όπως φαίνεται στους πίνακες που ακολουθούν Παρ' όλα αυτά όμως το σύνολο των μεθόδων που παραμένουν είναι μεγάλο και γι' αυτό το λόγο το ποσοστό του precision κυμαίνεται σε χαμηλά επίπεδα.

Στους πίνακες που ακολουθούν παρουσιάζονται τα ποσοστά του precision και recall για όλες τις αναφορές που χρησιμοποιήθηκαν ως είσοδοι στο 3%, 5% και 7% του συνολικού μεγέθους της ταξινομημένης λίστας των μεθόδων.

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
323156	1.02	100	98
323614	1.11	100	268
323635	1.865	83.33	268
325006	0.372	50	269
328445	0.735	100	136
334479	0.374	100	267
337725	0.66	100	151

Πίνακας 6.5: Precision και Recall για το 3% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
323156	0.61	100	163
323614	0.669	100	448
323635	1.121	83.33	446
325006	0.223	50	448
328445	0.442	100	226
334479	0.22	100	446
337725	0.395	100	253

Πίνακας 6.6: Precision και Recall για το 5% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
323156	0.435	100	230
323614	0.478	100	627
323635	0.8	83.33	625
325006	0.159	50	627
328445	0.315	100	317
334479	0.16	100	624
337725	0.282	100	354

Πίνακας 6.7: Precision και Recall για το 7% των μεθόδων της ταξινομημένης λίστας

Αρχική παρατήρηση είναι ότι το ποσοστό του recall δεν διαφοροποιήθηκε σε σχέση με το προηγούμενο, δηλαδή εκείνο που αφορούσε στο 100% των μεθόδων που ανακτήθηκαν από το bug localization σύστημα. Αυτό είναι αρκετά σημαντικό καθώς τόσο το 3% όσο το 5% και το 7% είναι αρκετά μικρά ποσοστά σε σχέση με το αρχικό και επομένως περιλαμβάνουν λιγότερες μεθόδους. Έτσι θα ήταν πολύ πιθανό κάποιες από τις μεθόδους που αποτελούν το resolution των αναφορών εισόδου, να μην εντοπίζονται σε τόσο μικρά ποσοστά. Το γεγονός όμως ότι ανακτήθηκαν και επιπλέον βρίσκονται και στο 3% των μεθόδων της ταξινομημένης λίστας, είναι ένα ενδεικτικό της ικανότητας του συστήματος να εντοπίζει τις «ελαττωματικές» μεθόδους.

Όσο αφορά στα ποσοστά του precision, παρατηρούμε ότι όσο αυξάνεται ο συνολικός αριθμός των μεθόδων, λόγου χάρη από 3% σε 5%, η τιμή του precision μειώνεται. Αυτό είναι απολύτως λογικό και αναμενόμενο, αφού αναζητούμε τον ίδιο αριθμό μεθόδων αλλά αυτή τη φορά σε μεγαλύτερο σύνολο. Κατά συνέπεια, το σύνολο αυτό θα περιέχει και «σκουπίδια», δηλαδή άλλες λέξεις κώδικα –μεθόδους συγκεκριμένα– που δεν χρειαζόμαστε.

Συγκρίνοντας όμως το precision των προηγούμενων πινάκων με τις τιμές του πίνακα 6.4 που αφορά στο 100% των ανακτημένων μεθόδων, παρατηρούμε βελτίωση, που μεταφράζεται σε αύξηση του ποσοστού, επειδή το σύνολο των μεθόδων της ταξινομημένης λίστας (number of methods in ranked list) έχει μειωθεί αισθητά σε σχέση με πριν.

Το πιο σημαντικό συμπέρασμα από τα δεδομένα που παρουσιάστηκαν είναι εκείνο που έχει αναφερθεί ήδη: το μέγεθος του συνόλου των μεθόδων που εντοπίζονται και ταξινομούνται από το σύστημα είναι τόσο μεγάλο που ακόμη και σε πολύ μικρά ποσοστά όπως το 3% έχει μικρό ποσοστό precision. Αυτό συνεπάγεται ότι απαιτείται μία βελτίωση του εργαλείου ώστε να κρατάει ακόμη λιγότερες μεθόδους στο σύνολο ή ακόμη και την επιλογή μίας διαφορετικής μετρικής για την απόδοση βάρους. Παρ’ όλα αυτά όμως λαμβάνοντας υπόψιν μας ότι το σύστημα που υλοποιήθηκε αποτελεί το θεμέλιο για τη δόμηση ενός άλλου, πιο προχωρημένου, θεωρούμε ικανοποιητική την αύξηση του precision για τα ποσοστά αυτά.

6.2.2 Εφαρμογή στο Dolphin

Το Dolphin είναι μία εφαρμογή διαχείρισης αρχείων (File Manager) ανοιχτού κώδικα (open source) και αποτελεί μέρος των εφαρμογών του KDE. Είναι σχεδιασμένο με έμφαση στην ευχρηστία και την απλότητα, ενώ επιτρέπει την ευελιξία και την προσαρμογή και λειτουργεί σε Linux συστήματα.

Τα δεδομένα που προέκυψαν από το στάδιο προ-επεξεργασίας των αναφορών αστοχιών για την έκδοση 2.1 του Dolphin παρουσιάζονται στον πίνακα που ακολουθεί. Από τα 7663 bug reports που ανακτήθηκαν από το Bugzilla, για περεταίρω ανάλυση χρειάστηκαν μόνο εκείνα που περιείχαν σχόλια (number of bug reports with comments) και από αυτά για την ανάλυσή μας ήταν απαραίτητα όσα περιελάμβαναν SCE στα σχόλιά τους (number of bug reports including SCE in comments).

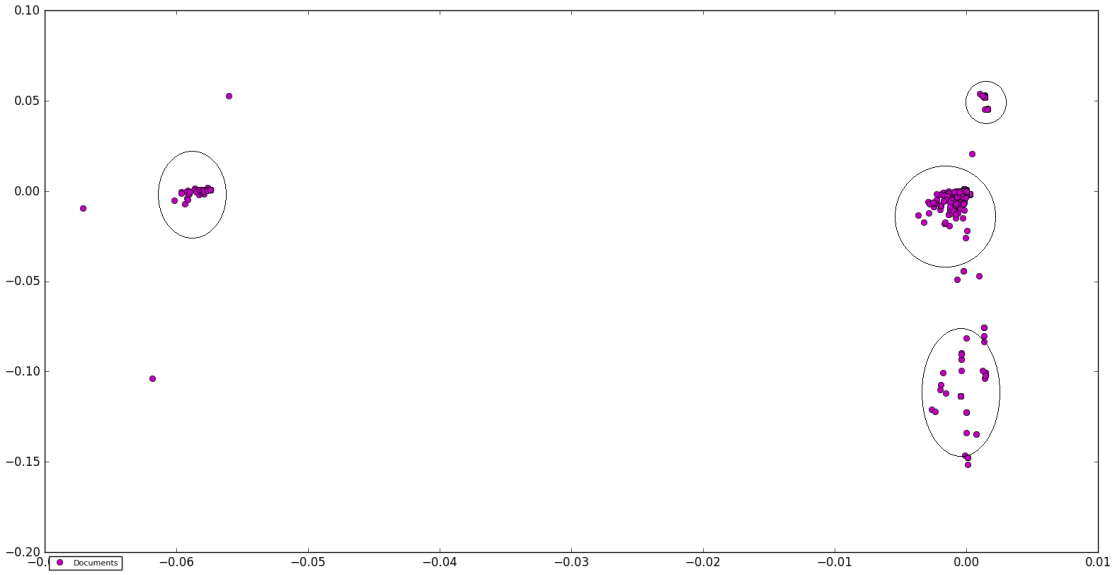
Number of bug reports retrieved from Bugzilla	7663
Number of bug reports with comments	7227
Number of bug reports including SCE in comments	1145
Number of bug reports that would be related with LSI input	1144

Πίνακας 6.8: Πλήθος των αναφορών αστοχιών που χρησιμοποιήθηκαν στα στάδια της προ-επεξεργασίας τους

Είναι σημαντικό να γίνει στο σημείο αυτό μία σύγκριση με το πλήθος των αναφορών σφαλμάτων που είχαμε στη διάθεσή μας από το Bugzilla για το Amarok. Ο συνολικός αριθμός των bug reports για το Amarok ήταν 17976 από τα οποία τελικά τα 7278 περιείχαν SCE και μπορούσαν να χρησιμοποιηθούν. Και στην περίπτωση του Amarok αλλά και εδώ παρατηρούμε ότι υπάρχει μεγάλη διαφορά μεταξύ των αναφορών που έχουν σχόλια και εκείνων που τελικά περιλαμβάνουν SCE. Όσο μικρότερος είναι ο αριθμός των αναφορών αστοχιών που περιέχουν SCE, τόσο περισσότερο μειώνεται το σύνολο των bug reports που θα χρησιμοποιηθούν για το clustering, γεγονός που είναι πιθανό να δημιουργήσει δυσκολίες στις συσχετίσεις των αναφορών κατά την εφαρμογή του LSI.

Ως είσοδοι στο LSI χρησιμοποιήθηκαν δώδεκα (12) αναφορές αστοχιών –μία κάθε φορά – για τις οποίες όπως και προηγουμένως γνωρίζουμε εκ των προτέρων τις μεθόδους που αποτελούν το resolution τους.

Το αποτέλεσμα του LSI είναι η δημιουργία clusters από συσχετισμένες αναφορές αστοχιών. Τα clusters φαίνονται στην εικόνα που ακολουθεί στην επόμενη σελίδα. Αντίστοιχα με τα clusters του Amarok, κάθε σημείο στο επίπεδο αντιπροσωπεύει μία αναφορά αστοχίας. Το μεγάλο πλήθος των αναφορών είναι συγκεντρωμένες σε ένα cluster, που σημαίνει ότι οι αναφορές αυτές σχετίζονται εννοιολογικά μεταξύ τους. Ταυτόχρονα όμως δημιουργούνται και άλλα δύο clusters που περιέχουν και εκείνα τα υπόλοιπα bug reports. Όπως είναι αναμενόμενο, δεν γίνεται όλες οι αναφορές να ομαδοποιούνται, γι’ αυτό φαίνεται να υπάρχουν κάποιες που ενώ πλησιάζουν στα clusters, δεν ανήκουν σε αυτά.



Εικόνα 6.2: Clustering για τις αναφορές αστοχιών του Dolphin

Για καθεμία αναφορά εισόδου έχουμε τα παρακάτω αποτελέσματα που αφορούν στο πλήθος των αναφορών σφαλμάτων που σχετίζονται με καθένα από τα bug reports (#related bug reports), στο πλήθος των SCE που σχετίζονται με το input (#related SCE) και τέλος στον αριθμό των SCE που προκύπτουν μετά την εφαρμογή των σχέσεων του .rsf αρχείου (#related SCE using code relations) :

Input Bug Reports (IDs)	**#related bug reports (LSI result)**	**#SCE in the input bug report**	**#related SCE (PS size)**	**#related SCE using code relations**
161385	399	3	227	1145
250787	391	2	206	1122
267171	461	1	379	1342
287829	464	3	388	1312
302264	134	15	49	177
303742	168	3	191	937
304524	372	6	329	1272
306147	484	24	375	1380
306167	492	5	435	1452
306459	444	11	376	1323
307254	135	2	65	244
308018	92	7	126	977

Πίνακας 6.9: Στοιχεία μετά την εφαρμογή του LSI

Κατόπιν δημιουργούνται τα σύνολα E_i για κάθε λέξη που περιλαμβάνεται στο Extended PS και E_k για τις λέξεις της αναφοράς εισόδου και χρησιμοποιείται η μετρική Jaccard για την απόδοση βάρους σε κάθε SCE. Τα αποτελέσματα είναι:

Bug Reports (IDs)	**Methods**	**Ranked Position in the Sorted List**	**Methods in the Sorted List**
161385	DolphinViewContainer.setUrl(KUrl)	2	1112
250787	InformationPanelContent.InformationPanelContent (QWidget*)	14	1090
	InformationPanelContent.showItem(KFileItem)	1	
	InformationPanelContent.showItems(KFileItemList)	2	
	InformationPanelContent.showIcon(KFileItem)	5	
	InformationPanelContent.showPreview (KFileItem,QPixmap)	3	
267171	UpdateItemStatesThread.run()	1	1297
	UpdateItemStatesThread.UpdateItemStatesThread()	2	
	VersionControlObserver.updateItemStates()	Not found	
287829	DolphinView.selectedItems()	2	1276
302264	VersionControlObserver.updateItemStates()	5	169
	UpdateItemStatesThread.setData(KVersionControlPlugin*, QList<VersionControlObserver::ItemState>)	8	
	UpdateItemStatesThread.run()	25	
303742	DolphinView.slotRoleEditingFinished (int,QByteArray,QVariant)	30	911
304524	KStandardItemListWidget.editedRoleChanged (QByteArray,QByteArray)	8	1231
	KStandardItemListWidget.closeRoleEditor()	2	
306147	DolphinView.slotRoleEditingFinished (int,QByteArray,QVariant)	64	1330
306167	KStandardItemListWidget.paint (QPainter*,QStyleOptionGraphicsItem*,QWidget*)	5	1400
306459	DolphinMainWindow.createSecondaryView(int)	786	1277
	KFileItemModel.KFileItemModel(QObject*)	7	
307254	DolphinViewActionHandler.slotTrashActivated (Qt::MouseButtons,Qt::KeyboardModifiers)	3	233
308018	KStandardItemListWidget.closeRoleEditor()	2	956
	KStandardItemListWidget.editedRoleChanged (QByteArray,QByteArray)	1	

Πίνακας 6.10: Αποτελέσματα bug localization

Μελετώντας λεπτομερώς τα δεδομένα, παρατηρούμε ότι στις αναφορές αστοχίας που χρησιμοποιήθηκαν ως είσοδοι για το LSI, το resolution ήταν άλλοτε μία μόνο συνάρτηση και άλλοτε περισσότερες. Το πλήθος των μεθόδων που ανακτώνται (Methods in the Sorted List) για κάθε bug report είναι πολύ μικρότερο σε σχέση με εκείνο για τις αναφορές εισόδου του Amarok. Αυτό συμβαίνει επειδή τα SCE του Dolphin είναι λιγότερα από εκείνα του Amarok, γεγονός που συνεπάγεται ότι οι σχέσεις που συνδέουν τις μεθόδους δε θα οδηγούν σε πολλές διαφορετικές. Στα αποτελέσματα που παρουσιάζονται στον πίνακα της προηγούμενης σελίδας μπορεί κανείς να δει τη θέση των μεθόδων στην ταξινομημένη λίστα (Ranked Position in the Sorted List), καθώς και το μέγεθος της εκάστοτε λίστας. Σε αντίθεση με δύο μεθόδους, οι οποίες δεν έχουν ικανοποιητική θέση στο ranking, οι υπόλοιπες έχουν αρκετά υψηλό σκορ με αποτέλεσμα να καταλαμβάνουν τις καλύτερες θέσεις της λίστας.

Όπως έχει ήδη αναφερθεί, στο σύστημα που δημιουργήθηκε, το βάθος προσπέλασης του γράφου είναι μεταβλητό, πράγμα που συνεπάγεται ότι ο χρήστης είναι εκείνος που καθορίζει την τιμή του. Έτσι στην αναφορά 267171 δεν υπήρχε κάποια σχέση του FETCH Tool που να φέρει στο σύνολό μας τη συγκεκριμένη μέθοδο. Μεταβάλλοντας την παράμετρο του βάθους, είναι πιθανό η μέθοδος αυτή να εντοπιστεί, αλλά αυτό μπορεί να έχει ως αποτέλεσμα να γεμίσει το σύνολό μας με περισσότερες μεθόδους που δεν σχετίζονται με το σφάλμα της αναφοράς αυτής.

Στην αναφορά αστοχίας με id 306459 η μέθοδος που χρειάστηκε τροποποίηση για να διορθωθεί το σφάλμα ήταν η DolphinMainWindow.createSecondaryView(int). Η συγκεκριμένη εντοπίστηκε από το σύστημα, όμως το βάρος που της αποδόθηκε την κατέταξε σε πολύ χαμηλή θέση, 786^{η} στις 1227 μεθόδους, δηλαδή στο top 64%. Αυτό προέκυψε εξαιτίας της κυρίως της διαφορετικής κλάσης της μεθόδου αυτής με την μέθοδο KFileItemModel.KFileItemModel(QObject*), η οποία εμφανίστηκε 7^{η} στην κατάταξη. Φυσικά πολύ σημαντικό ρόλο παίζει και το πλήθος των SCE της εφαρμογής Dolphin. Όσο λιγότερες λέξεις κώδικα υπάρχουν σε ένα πρόγραμμα λογισμικού, τόσο λιγότερες είναι και οι σχέσεις που θα αναπτύσσονται μεταξύ τους και εντοπίζονται από το FETCH Tool. Συνεπώς, η πιθανότητα να υπάρχει μονοπάτι στο γράφο των σχέσεων-μεθόδων που να οδηγεί από κάποια μέθοδο σε κάποια άλλη μικραίνει, με αποτέλεσμα η createSecondaryView(int) να μην εμφανίζει κοινά στοιχεία με τα SCE της αναφοράς σφάλματος και έτσι να κατατάσσεται στις τελευταίες θέσεις.

Παρατηρούμε ότι εκτός από τη μέθοδο, η οποία δεν εντοπίστηκε από το σύστημά bug localization, υπάρχουν και τρεις ακόμη μέθοδοι που κατατάχθηκαν στις χαμηλότερες θέσεις στην ταξινομημένη λίστα συγκριτικά με τις υπόλοιπες. Πιο συγκεκριμένα, αυτές είναι οι εξής: αρχικά η DolphinView.slotRoleEditingFinished (int,QByteArray,QVariant), η οποία έχει 30^{η} θέση στις 911 μεθόδους –bug report 303742, η μέθοδος που αποτελεί το resolution της αναφοράς αστοχίας με id 306147 DolphinView.slotRoleEditingFinished(int,QByteArray,QVariant) στη θέση 64 στις 1330 και τέλος η μέθοδος DolphinMainWindow.createSecondaryView(int) που κατατάχθηκε 786^{η} στις 1277 για την αναφορά 306459.

Ακολουθεί ο πίνακας με το precision και recall για καθεμία από τις προηγούμενες αναφορές σφαλμάτων. Είναι αναμενόμενο ότι αφού οι μέθοδοι εντοπίστηκαν όλες (εκτός από μία) το ποσοστό του recall θα είναι 100% (και 66.67% στην προκειμένη περίπτωση). Και πάλι όμως τα ποσοστά του precision είναι

ιδιαίτερα χαμηλά, γεγονός που οφείλεται στον μεγάλο αριθμό μεθόδων που ανακτώνται από το σύστημα.

Bug Reports (IDs)	**Methods**	**Ranked position in list / number of methods in list**	**Precision (%)**	**Recall (%)**
161385	DolphinViewContainer.setUrl(KUrl)	2/1112	0.089	100
250787	InformationPanelContent.InformationPanelContent (QWidget*)	14/1090	0.460	100
	InformationPanelContent.showItem(KFileItem)	1/1090		
	InformationPanelContent.showItems(KFileItemList)	2/1090		
	InformationPanelContent.showIcon(KFileItem)	5/1090		
	InformationPanelContent.showPreview (KFileItem,QPixmap)	3/1090		
267171	UpdateItemStatesThread.run()	1/1297	0.154	66.67
	UpdateItemStatesThread.UpdateItemStatesThread()	2/1297		
	VersionControlObserver.updateItemStates()	Not found		
287829	DolphinView.selectedItems()	2/1276	0.078	100
302264	VersionControlObserver.updateItemStates()	5/169	1.775	100
	UpdateItemStatesThread.setData(KVersionControlPlugin*,QList<VersionControlObserver::ItemState>)	8/169		
	UpdateItemStatesThread.run()	25/169		
303742	DolphinView.slotRoleEditingFinished (int,QByteArray,QVariant)	30/911	0.120	100
304524	KStandardItemListWidget.editedRoleChanged (QByteArray,QByteArray)	8/1231	0.162	100
	KStandardItemListWidget.closeRoleEditor()	2/1231		
306147	DolphinView.slotRoleEditingFinished (int,QByteArray,QVariant)	64/1330	0.075	100
306167	KStandardItemListWidget.paint (QPainter*,QStyleOptionGraphicsItem*,QWidget*)	5/1400	0.071	100
306459	DolphinMainWindow.createSecondaryView (int)	786/1277	0.156	100
	KFileItemModel.KFileItemModel(QObject*)	7/1277		
307254	DolphinViewActionHandler.slotTrashActivated (Qt::MouseButtons,Qt::KeyboardModifiers)	3/233	0.429	100
308018	KStandardItemListWidget.closeRoleEditor()	2/956	0.209	100
	KStandardItemListWidget.editedRoleChanged (QByteArray,QByteArray)	1/956		

Πίνακας 6.11: Precision και Recall για τα αποτελέσματα του bug localization

Ο αριθμός των δεδομένων που ανακτήθηκαν από το bug localization σύστημα μας για κάθε μία αναφορά αστοχίας που χρησιμοποιήθηκε ως είσοδος είναι αρκετά μεγάλος και όπως αναφέρθηκε είναι η αιτία του πολύ μικρού ποσοστού του precision. Έτσι λοιπόν προστέθηκε στο σύστημα, που υλοποιήθηκε, ένα επιπλέον χαρακτηριστικό: η δυνατότητα να επιλέγει ο χρήστης ποιο ποσοστό της ταξινομημένης λίστας θέλει να δει.

Το χαρακτηριστικό αυτό είναι ιδιαίτερα χρήσιμο και προσθέτει ένα σημαντικό πλεονέκτημα στη διαχείριση των μεθόδων που ανακτώνται, αφού ο προγραμματιστής θα μπορεί να επιλέξει το μέγεθος της ταξινομημένης λίστας. Συνεπώς, ξεκινώντας από ένα μικρό ποσοστό της λίστας, αν το αποτέλεσμα δεν είναι ικανοποιητικό, συνεχίζουν αυξάνοντάς το έως ότου να καταλήξουν στον επιθυμητό αριθμό.

Στους πίνακες που ακολουθούν παρουσιάζονται αντίστοιχα τα αποτελέσματα που προκύπτουν για το 3%, το 5% και το 7% των μεθόδων που ανακτώνται και επιστρέφονται στο χρήστη ως αποτέλεσμα του συστήματος.

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
161385	3.03	100	33
250787	15.15	100	32
267171	5.26	66.67	38
287829	2.63	100	38
302264	20	33.33	5
303742	0	0	27
304524	5.55	100	36
306147	0	0	39
306167	2.38	100	42
306459	2.63	50	38
307254	16.66	100	6
308018	7.14	100	28

Πίνακας 6.12: Precision και Recall για το 3% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
161385	1.82	100	55
250787	9.25	100	54
267171	3.125	66.67	64
287829	1.58	100	63
302264	25	66.67	8
303742	2.22	100	45
304524	3.28	100	61
306147	1.51	100	66
306167	1.43	100	70
306459	1.59	50	63
307254	9.10	100	11
308018	4.26	100	47

Πίνακας 6.13: Precision και Recall για το 5% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
161385	1.30	100	77
250787	6.59	100	76
267171	2.22	66.67	90
287829	1.12	100	89
302264	18.18	66.67	11
303742	1.58	100	63
304524	2.34	100	86
306147	1.07	100	93
306167	1.02	100	98
306459	1.12	50	89
307254	6.25	100	16
308018	3.03	100	66

Πίνακας 6.14: Precision και Recall για το 7% των μεθόδων της ταξινομημένης λίστας

Συγκρίνοντας τους πίνακες αυτούς με τον προηγούμενο που αφορά στο 100% των μεθόδων, παρατηρούμε ότι το ποσοστό του precision αυξήθηκε σημαντικά. Αυτό ήταν αναμενόμενο αφού το πλήθος των μεθόδων της ταξινομημένης λίστας ελαττώθηκε, ενώ ο αριθμός των μεθόδων που αναζητούμε (bug resolution) παρέμεινε σταθερός. Επιπλέον είναι σημαντικό να προσέξουμε ότι καθώς αυξάνονται οι μέθοδοι της ταξινομημένης λίστας, τόσο το ποσοστό του precision μειώνεται.

Σχετικά με το ποσοστό του recall, από τα δεδομένα που παρουσιάστηκαν, διαπιστώνουμε ότι παραμένει σε πολύ υψηλά επίπεδα. Η αστοχία, που περιγράφεται στην αναφορά σφάλματος με id 267171, απαιτούσε την τροποποίηση τριών μεθόδων του πηγαίου κώδικα. Από αυτές το bug localization σύστημα εντόπισε τις δύο στο 100% των μεθόδων και συνεπώς το ποσοστό του recall (66.67%) για τη συγκεκριμένη αναφορά, θα παραμείνει το ίδιο και για το 3%, 5% και 7%, όπως φαίνεται στους πίνακες.

Διαφορές εντοπίζονται στις αναφορές αστοχίας με αναγνωριστικό αριθμό 302264, 303742 και 306147 και αφορούν κυρίως το 3% των μεθόδων. Η πρώτη αναφορά παρατηρούμε ότι έχει recall 33.33%, που οφείλεται στο γεγονός ότι για το 3% του πλήθους της ταξινομημένης λίστας –και συγκεκριμένα για 5 μεθόδους– ανακτήθηκε μόνο μία από τις τρεις μεθόδους που αποτελούσαν το resolution της. Παρ' όλα αυτά, όταν το πλήθος των μεθόδων στο σύνολο αυξάνεται στο 5% και 7% εντοπίζονται δύο μέθοδοι και αυτό μεταβάλλει το ποσοστό του precision στο 66.67%. Οι άλλες δύο αναφορές (303742 και 306147) ανήκουν στην ίδια περίπτωση: για το 3% των μεθόδων καμία από εκείνες που τροποποιήθηκαν δεν υπάρχει στην ταξινομημένη λίστα και για τον λόγο αυτό τόσο το recall, όσο και το precision είναι 0%. Αυτό όμως αλλάζει για το 5% και 7% καθώς σε αυτό το ποσοστό, το σύστημα εντοπίζει όλες τις μεθόδους και επομένως το recall γίνεται 100%.

Τέλος, η bug report 306459 έχει δύο μεθόδους για resolution, οι οποίες για το 100% των μεθόδων εντοπίζονται από το σύστημα. Όμως η μία εκ των δύο κατατάχθηκε σε αρκετά χαμηλή θέση (786^{η}) και κατά συνέπεια δεν εντοπίζεται για μικρότερα ποσοστά του συνόλου της ταξινομημένης λίστας.

6.2.3 Εφαρμογή στο Konqueror

Το Konqueror είναι ένας δωρεάν και ανοιχτού κώδικα περιηγητής διαδικτύου (web browser) καθώς και διαχειριστής αρχείων (file manager), ο οποίος παρέχει τη δυνατότητα οπτικοποίησης αρχείων του συστήματος όπως τοπικά αρχεία, αρχεία ενός απομακρυσμένου FTP/SFTP εξυπηρετητή και συμπιεσμένα αρχεία. Η εφαρμογή αυτή αποτελεί κύριο τμήμα του KDE project, έχει υλοποιηθεί από εθελοντές προγραμματιστές για λειτουργικά συστήματα τύπου Unix και Windows. Σαν web browser στηρίζεται στην KHTML μηχανή που σημαίνει ότι υποστηρίζει όλες τις τελευταίες τεχνολογίες όπως HTML5, Javascript, CSS3 και άλλα. Επιπλέον χρησιμοποιεί τις τελευταίες KDE τεχνολογίες και παρέχει ειδικό πρόγραμμα για άνοιγμα PDF αρχείων, επεξεργαστή όχι μόνο κειμένου αλλά και φύλλων υπολογισμού χωρίς κάποια άλλη ξεχωριστή εφαρμογή.

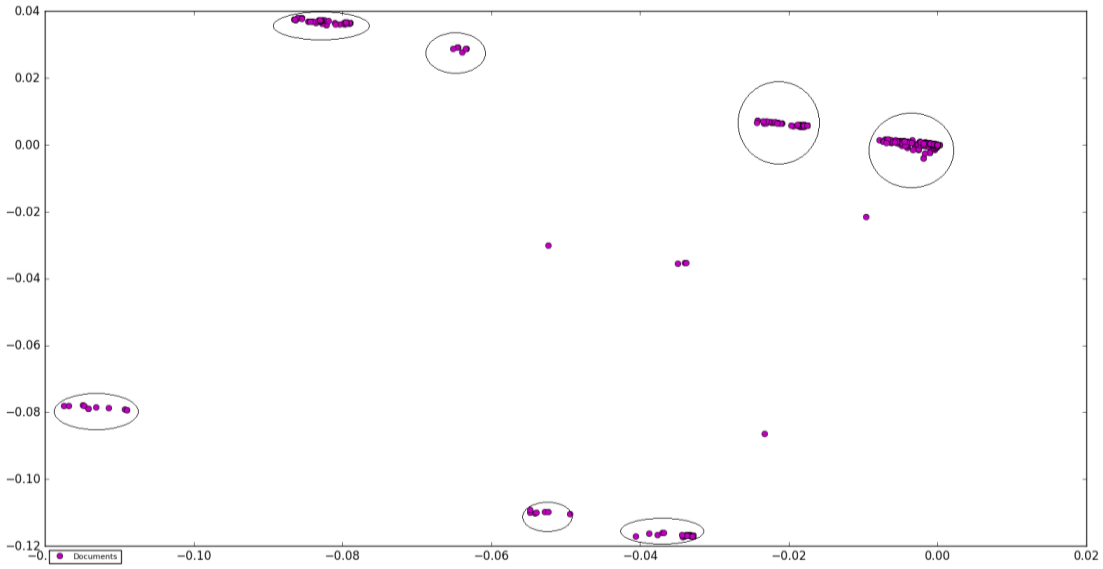
Έχει ήδη αναφερθεί σε προηγούμενα κεφάλαια και παραγράφους ότι το πρώτο στάδιο του συστήματος που δημιουργήθηκε είναι το στάδιο προ-επεξεργασίας των αναφορών αστοχιών. Σε αυτό το στάδιο για την εφαρμογή Konqueror και συγκεκριμένα για την έκδοση 4.0 ανακτήθηκαν συνολικά 38480 αναφορές από το Bugzilla. Στον πίνακα που ακολουθεί υπάρχουν δεδομένα για καθένα από τα βήματα της προ-επεξεργασίας όπως το πλήθος των αναφορών στις οποίες περιέχονται σχόλια (38228) και το πλήθος των αναφορών που περιέχουν SCE στα σχόλιά τους.

Number of bug reports retrieved from Bugzilla	38480
Number of bug reports with comments	38228
Number of bug reports including SCE in comments	14433
Number of bug reports that would be related with LSI input	14432

Πίνακας 6.15: Πλήθος των αναφορών αστοχιών που χρησιμοποιήθηκαν στα στάδια της προ-επεξεργασίας τους

Συγκριτικά με τις δύο προηγούμενες εφαρμογές (Amarok, Dolphin), το Konqueror έχει τις περισσότερες αναφορές σφαλμάτων που ταυτόχρονα περιέχουν σχόλια με SCE (14432). Ο μεγάλος αριθμός των αναφορών συντελεί στη δημιουργία καλύτερων και ακριβέστερων clusters, όταν σε αυτές εφαρμοστεί το LSI. Αυτό συνεπάγεται μεγαλύτερη ακρίβεια στον εννοιολογικό συσχετισμό των αναφορών αστοχιών και ως εκ τούτου τα αποτελέσματα που θα προκύψουν αναμένεται να είναι καλύτερα.

Για το LSI χρησιμοποιήθηκαν πέντε (5) bug reports για τις οποίες έχουμε βρει τις μεθόδους που χρειάστηκαν τροποποίηση προκειμένου να επιλυθεί το σφάλμα στο οποίο αναφερόταν κάθε μία. Με τη μέθοδο αυτή δημιουργήθηκαν clusters στα οποία ανήκουν αναφορές αστοχιών που σχετίζονται εννοιολογικά μεταξύ τους. Και πάλι κάθε αναφορά σφάλματος απεικονίζεται με ένα σημείο στο ακόλουθο διάγραμμα. Αυτή τη φορά δεν συγκεντρώνονται οι περισσότερες αναφορές σε ένα cluster, αλλά δημιουργούνται αρκετά μικρότερα και φυσικά υπάρχουν αρκετές αναφορές που δεν συσχετίζονται με κανένα από αυτά με αποτέλεσμα να μην ανήκουν σε κανένα cluster.



Εικόνα 6.3: Clustering για τις αναφορές αστοχιών του Konqueror

Για καθεμία αναφορά εισόδου παρουσιάζονται στον ακόλουθο πίνακα δεδομένα που αναφέρονται στο πλήθος των αναφορών σφαλμάτων που σχετίζονται με καθένα από τα bug reports (#related bug reports), στο πλήθος των SCE που σχετίζονται με το input (#related SCE) και τέλος στον αριθμό των SCE που προκύπτουν μετά την εφαρμογή των σχέσεων του rsf αρχείου (#related SCE using code relations:

Input Bug Reports (IDs)	**#related bug reports (LSI result)**	**#SCE in the input bug report**	**#related SCE (PS size)**	**#related SCE using code relations**
153117	3341	3	8288	11887
153533	7931	6	8921	12554
155225	6056	36	8559	12134
155434	7850	519	9573	13231
156658	895	3	5976	9066

Πίνακας 6.16: Στοιχεία μετά την εφαρμογή του LSI

Μετά την ολοκλήρωση της εφαρμογής του LSI, βρισκόμαστε πια στο στάδιο της κυρίως επεξεργασίας των αναφορών αστοχιών, για τις οποίες όπως έχει ήδη αναφερθεί έχουν συγκεντρωθεί οι λέξεις κώδικα που περιλαμβάνονται στα σχόλιά τους. Έτσι για καθεμία SCE του Extended PS δημιουργείται ένα σύνολο E_i το οποίο συγκρίνεται με το σύνολο E_k που περιλαμβάνει όλες τις λέξεις κώδικα της αναφοράς εισόδου καθώς και όσες SCE συνδέονται με αυτές μέσω των σχέσεων που προέκυψαν από την ανάλυση του Fetch Tool. Η σύγκριση των συνόλων με τη μετρική Jaccard προσδίδει σε κάθε SCE ένα βάρος με βάση το οποίο κατατάσσονται οι μέθοδοι και δημιουργείται η ταξινομημένη λίστα. Η θέση της κάθε μεθόδου στη λίστα αυτή φαίνεται στον πίνακα που ακολουθεί.

Bug Reports (IDs)	Methods	Ranked Position in the Sorted List	Methods in the Sorted List
153117	hp_removeDupe(KCompletionMatches,QString,KCompletionMatches::Iterator)	3	599
153533	KonqMainWindow.slotAddClosedUrl(KonqFrameBase*)	Not found	9819
	ViewMgrTest.testAddTab()	4	
	ViewMgrTest.testDuplicateSplittedTab()	374	
155225	KonqView.saveConfig(KConfigGroup,QString,KonqFrameBase::Options)	16	9419
155434	KHTMLView.focusNextPrevNode(bool)	8	10266
	DocumentImpl.setActiveNode(NodeImpl*)	1687	
	DocumentImpl.setFocusNode(NodeImpl*)	687	
156658	KonquerorAdaptor.openBrowserWindow(QString,QByteArray)	6	6889
	KonquerorAdaptor.createNewWindow(QString,QString,QByteArray,bool)	5	
	KonquerorAdaptor.createNewWindowWithSelection(QString,QStringList,QByteArray)	4	
	KonquerorAdaptor.createBrowserWindowFromProfile(QString,QString,QByteArray)	1	
	KonquerorAdaptor.createBrowserWindowFromProfileAndUrl(QString,QString,QString,QByteArray)	2	
	KonquerorAdaptor.createBrowserWindowFromProfileUrlAndMimeType(QString,QString,QString,QString,QByteArray)	3	

Πίνακας 6.16: Αποτελέσματα bug localization

Αρχική παρατήρηση που μπορούμε να κάνουμε σύμφωνα με τα παραπάνω δεδομένα είναι ότι το πλήθος των μεθόδων στην ταξινομημένη λίστα (Methods in the Sorted List) είναι πολύ μεγάλο σε σχέση με τις μεθόδους που αποτελούν το resolution κάθε αναφοράς σφάλματος εισόδου. Επιπροσθέτως είναι θετικό ότι όλες οι μέθοδοι εντοπίστηκαν από το bug localization σύστημα και οι περισσότερες έχουν καταταχθεί σε πολύ υψηλές θέσεις. Εξαίρεση αποτελούν:

α) η μέθοδος της αναφοράς αστοχίας που χρησιμοποιήθηκε ως είσοδος στο LSI με id 153533, KonqMainWindow.slotAddClosedUrl(KonqFrameBase*) η οποία δεν εντοπίστηκε αλλά και η ViewMgrTest.testDuplicateSplittedTab() που κατατάχθηκε στο top 3.81% των μεθόδων της λίστας.

β) Στην αναφορά 155434 οι μέθοδοι DocumentImpl.setActiveNode(NodeImpl*) και DocumentImpl.setFocusNode(NodeImpl*) οι οποίες παρόλο που εντοπίστηκαν κατατάχθηκαν σε χαμηλές θέσεις, 1687 και 687 αντίστοιχα. Βέβαια συγκριτικά με το πλήθος των μεθόδων που ανακτήθηκαν (10266) αυτές βρίσκονται στο 16,4% και 6,7% αντίστοιχα.

Και στις δύο παραπάνω περιπτώσεις αναφορών εισόδου οι μέθοδοι που είτε δεν εντοπίστηκαν, είτε δεν βρίσκονται σε υψηλές θέσεις ανήκουν σε διαφορετική κλάση από τις μεθόδους εκείνες που ήταν στις πρώτες θέσεις της ταξινομημένης λίστας.

Έτσι λοιπόν συγκεκριμένα για την αναφορά 153533 δεν υπήρχε σχέση στο βάθος που επιλέχθηκε ώστε να φέρει στη λίστα τη μέθοδο που δεν εντοπίστηκε και αντίστοιχα για την αναφορά 155434 η μέθοδος που δεν εντοπίστηκε ανήκε σε διαφορετική κλάση και δεν συνδεόταν για το συγκεκριμένο βάθος με κάποια από τα SCE που συγκεντρώθηκαν με την τεχνική LSI. Αξίζει να σημειωθεί όμως ότι και στο Konqueror οι μέθοδοι που κατατάχθηκαν σε σχετικά χαμηλές θέσεις, έχουν ονόματα τα οποία εννοιολογικά εμφανίζουν ομοιότητες με κάποιες από τις μεθόδους των υψηλότερων θέσεων.

Ακολουθεί ο πίνακας που περιλαμβάνει την προηγούμενη πληροφορία σε συνδυασμό με τα ποσοστά precision και recall. Το ποσοστό του recall διατηρείται σε πολύ υψηλά επίπεδα, γεγονός αναμενόμενο αφού όλες οι μέθοδοι εκτός από μία εντοπίστηκαν από το bug localization σύστημα. Παρ' όλα αυτά οι μέθοδοι που ανακτώνται είναι αρκετές και αυτό δυσχεραίνει την αναζήτηση εκείνης που πρέπει να τροποποιηθεί από έναν προγραμματιστή. Αυτός είναι και ο λόγος που τα ποσοστά του precision είναι αρκετά μικρά.

Bug Reports (IDs)	**Methods**	**Ranked position in list / number of methods in list**	**Precision (%)**	**Recall (%)**
153117	hp_removeDupe(KCompletionMatches,QString,KCompletionMatches::Iterator)	3/599	0.167	100
153533	KonqMainWindow.slotAddClosedUrl(KonqFrameBase*)	Not found	0.020	66.66
	ViewMgrTest.testAddTab()	4/9819		
	ViewMgrTest.testDuplicateSplittedTab()	374/9819		
155225	KonqView.saveConfig(KConfigGroup,QString,KonqFrameBase::Options)	16/9419	0.012	100
155434	KHTMLView.focusNextPrevNode(bool)	8/10266	0.029	100
	DocumentImpl.setActiveNode(NodeImpl*)	1687/10266		
	DocumentImpl.setFocusNode(NodeImpl*)	687/10266		
156658	KonquerorAdaptor.openBrowserWindow(QString,QByteArray)	6/6889	0.087	100
	KonquerorAdaptor.createNewWindow(QString,QString,QByteArray,bool)	5/6889		
	KonquerorAdaptor.createNewWindowWithSelection(QString,QStringList,QByteArray)	4/6889		
	KonquerorAdaptor.createBrowserWindowFromProfile(QString,QString,QByteArray)	1/6889		
	KonquerorAdaptor.createBrowserWindowFromProfileAndUrl(QString,QString,QString,QByteArray)	2/6889		
	KonquerorAdaptor.createBrowserWindowFromProfileUrlAndMimeType(QString,QString,QString,QString,QByteArray)	3/6889		

Πίνακας 6.17: Precision και Recall για τα αποτελέσματα του bug localization

Στοχεύοντας λοιπόν στη μείωση του αριθμού των μεθόδων, με σκοπό η ταξινομημένη λίστα να γίνει πιο εύκολα διαχειρίσιμη και προσπαθώντας να αυξήσουμε το ποσοστό του precision, χωρίς όμως να «χαλάσουμε» το αντίστοιχο ποσοστό του recall, επιλέγουμε ένα συγκεκριμένο ποσοστό, στην προκειμένη περίπτωση 3%, 5% και 7% και παρατηρούμε τα εξής:

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
153117	5.56	100	18
153533	0.68	66.66	294
155225	0.35	100	282
155434	0.33	33.33	307
156658	2.43	100	206

Πίνακας 6.18: Precision και Recall για το 3% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
153117	3.33	100	30
153533	0.41	66.66	490
155225	0.22	100	470
155434	0.19	33.33	513
156658	1.45	100	344

Πίνακας 6.19: Precision και Recall για το 5% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
153117	2.38	100	42
153533	0.41	66.66	490
155225	0.22	100	470
155434	0.19	33.33	513
156658	1.45	100	344

Πίνακας 6.20: Precision και Recall για το 7% των μεθόδων της ταξινομημένης λίστας

Συγκρίνοντας τα δεδομένα των προηγούμενων πινάκων με αυτά που παρουσιάζονται στον πίνακα 6.17, κάνουμε τις ακόλουθες παρατηρήσεις:

- Το ποσοστό του precision σημείωσε σημαντική αύξηση σε σχέση με εκείνο που αφορά στο 100% των μεθόδων που συγκεντρώνονται από το σύστημά μας (πίνακας 6.17). Είναι το ίδιο φαινόμενο που παρατηρήθηκε και στις προηγούμενες εφαρμογές λογισμικού που αναλύθηκαν. Αναζητούμε, δηλαδή, τον ίδιο αριθμό μεθόδων, σε μικρότερο σύνολο, με αποτέλεσμα το precision να αυξάνεται, αφού πιθανά «άχρηστες» μέθοδοι εξαλείφονται.

- Αν και το precision βελτιώθηκε για όλες τις αναφορές αστοχίας που χρησιμοποιήθηκαν ως είσοδοι στο bug localization σύστημα που υλοποιήθηκε, υπάρχουν κάποιες –συγκεκριμένα η πρώτη και η τελευταία– των οποίων το ποσοστό εμφανίζει τη μεγαλύτερη αύξηση. Για την αναφορά 153117, αυτό συνέβη επειδή οι μέθοδοι που συγκεντρώθηκαν μετά την προσπέλαση του γράφο των σχέσεων ήταν λιγότερες σε σχέση με εκείνες που προέκυψαν για τις υπόλοιπες αναφορές αστοχιών. Αντίθετα η αναφορά αστοχίας με id 156658 εμφάνισε μεγάλη διαφοροποίηση στο ποσοστό του precision, παρ' όλο που το σύνολο των μεθόδων της ταξινομημένης λίστας είναι αρκετά μεγάλο (200 μέθοδοι), εξαιτίας του αριθμού των μεθόδων που αναζητούμε σε αυτό (πέντε).

- Όσο αφορά στο recall, διατηρώντας το 100% του συνόλου των μεθόδων της ταξινομημένης λίστας, σύμφωνα με τον πίνακα 6.17, για όλες τις αναφορές αστοχίας, εντοπίστηκαν όλες οι μέθοδοι που αποτελούν το resolution τους. Εξαίρεση αποτελεί η αναφορά 153533, για την οποία εντοπίστηκαν μόνο δύο μέθοδοι, αντί για τρεις που τροποποιήθηκαν και επομένως το ποσοστό του recall για τη συγκεκριμένη είναι 66.66%. Είναι λοιπόν προφανές ότι και για μικρότερο αριθμό μεθόδων (number of methods in ranked list), η μέθοδος KonqMainWindow.slotAddClosedUrl(KonqFrameBase*) δε θα εντοπίστει. Για το λόγο αυτό το recall διατηρείται στο 66.66%.

- Στους πίνακες της προηγούμενης σελίδας βλέπουμε ότι η αναφορά 155434 έχει μικρότερό recall (33.33%) συγκριτικά με το ποσοστό που είχε στον πίνακα 6.17 (100%). Αυτό συμβαίνει διότι οι δύο από τις τρείς μεθόδους που τροποποιήθηκαν προκειμένου να επιλυθεί το bug της εφαρμογής, δεν περιλαμβάνονται στο 3%, 5% και 7% του συνολικού αριθμού των μεθόδων.

  Συνεπώς, καταλήγουμε στο συμπέρασμα ότι χρειαζόμαστε αφενός μεγαλύτερο ποσοστό από το 7% έτσι ώστε να συμπεριληφθούν και αυτές οι μέθοδοι και αφετέρου μια προσέγγιση στην οποία το μέγεθος της ταξινομημένης λίστας να είναι εύκολα διαχειρίσιμο από έναν άνθρωπο.

6.2.4 Εφαρμογή στο Kopete

Το Kopete είναι μία εφαρμογή αποστολής άμεσων μηνυμάτων (instant messenger) που υποστηρίζει πρωτόκολλα όπως το AIM, ICQ, Windows Live Messenger, Yahoo, Jabber, Gadu-Gadu και άλλα. Είναι λογισμικό ανοιχτού κώδικα και αποτελεί μέρος του KDE project. Επιπλέον είναι σχεδιασμένο ώστε να προσφέρει ευελιξία καθώς είναι επεκτάσιμο σε σύστημα πολλαπλών πρωτοκόλλων κατάλληλο όχι μόνο για προσωπική, αλλά και για επαγγελματική χρήση.

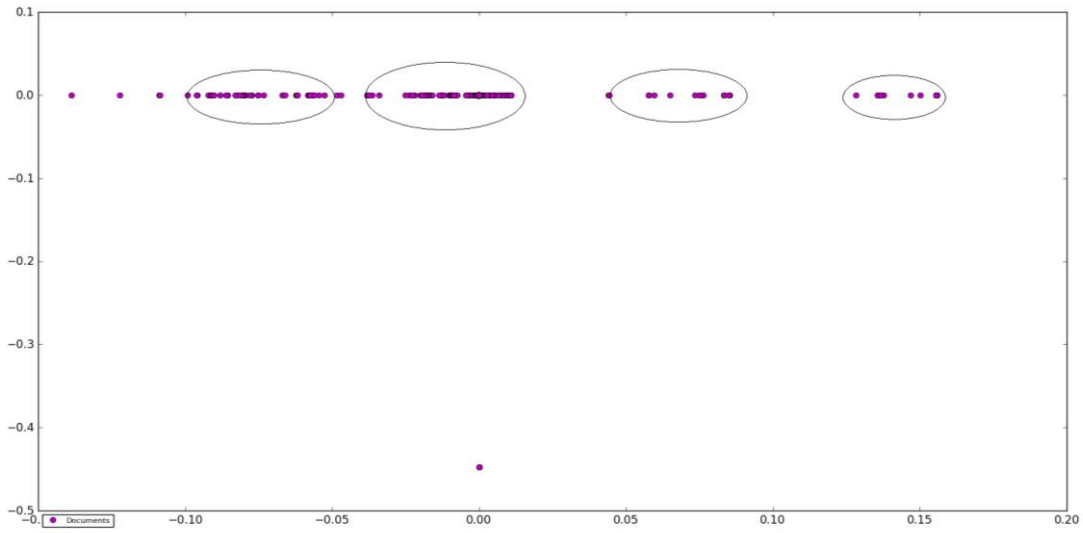
Στον πίνακα που ακολουθεί παρουσιάζονται ορισμένες γενικές πληροφορίες σχετικά με τον αριθμό των αναφορών αστοχιών που ανακτήθηκαν από το Bugzilla για την έκδοση 1.0.80 του Kopete (9872 αναφορές συνολικά), τον αριθμό των αστοχιών οι οποίες περιλαμβάνουν σχόλια και έπειτα από σχετική επεξεργασία αναγράφεται και ο αριθμός των bug reports που περιλαμβάνουν λέξεις κώδικα στα σχόλιά τους καθώς και εκείνες που θα συσχετιστούμε την αναφορά εισόδου κατά την εφαρμογή της τεχνικής LSI.

Number of bug reports retrieved from Bugzilla	9872
Number of bug reports with comments	9814
Number of bug reports including SCE in comments	3266
Number of bug reports that would be related with LSI input	3265

Πίνακας 6.21: Πλήθος των αναφορών αστοχιών που χρησιμοποιήθηκαν στα στάδια της προ-επεξεργασίας τους

Το επόμενο βήμα της επεξεργασίας των αναφορών σφαλμάτων, αφού έχουν ήδη διαχωριστεί οι λέξεις κώδικα (SCE), είναι η εφαρμογή της τεχνικής LSI με σκοπό τον εννοιολογικό συσχετισμό των αναφορών. Πιο συγκεκριμένα χρησιμοποιήθηκαν οκτώ (8) αναφορές για τις οποίες και πάλι γνωρίζουμε εκ των προτέρων τις μεθόδους που τροποποιήθηκαν προκειμένου να επιλυθεί η αστοχία που παρουσιάστηκε. Το αποτέλεσμα της τεχνικής αυτής είναι η δημιουργία ομάδων (clusters) αποτελούμενων από bug reports.

Τα clusters αυτά φαίνονται στην ακόλουθη εικόνα. Όπως και για τις προηγούμενες εφαρμογές, έτσι και για το kopete, κάθε σημείο αντιστοιχεί σε μία αναφορά και κάθε κύκλος περιλαμβάνει τις αναφορές εκείνες στις οποίες έχει εντοπιστεί εννοιολογική ομοιότητα. Παρατηρούμε ότι ειδικά για δύο clusters είναι εξαιρετικά δύσκολο να γίνει διαχωρισμός, καθώς υπάρχουν αναφορές που έχουν εντοπιστεί ανάμεσά τους. Φυσικά όπως είναι αναμενόμενο κάποια bug reports δεν εμφανίζουν πλήρη εννοιολογική συσχέτιση με αντίστοιχα κάποιου cluster, γεγονός το οποίο απεικονίζεται με τα σημεία εκείνα του διαγράμματος που βρίσκονται ενδιάμεσα ή ακόμη και πολύ πιο μακριά από τα clusters που έχουν δημιουργηθεί. Χαρακτηριστικό παράδειγμα είναι η αναφορά κάτω στο κέντρο καθώς και οι αναφορές αστοχιών αριστερά του πρώτου cluster.



Εικόνα 6.4.: Clustering για τις αναφορές αστοχιών του Kopete

Για τις αναφορές εισόδου που επιλέχτηκαν ακολουθεί παρακάτω πίνακας με πληροφορίες που σχετίζονται με τον αριθμό των SCE που εντοπίστηκαν σε κάθε μία αναφορά εισόδου (#SCE in the input bug report), το πλήθος των αναφορών που συσχετίστηκαν με την εκάστοτε αναφορά (#related bug reports), τον αριθμό των SCE που σχετίζονται με την αναφορά εισόδου και αποτελούν το Problem Set (#related SCE) και τέλος το σύνολο των SCE που σχετίζονται με την αναφορά χρησιμοποιώντας τις σχέσεις που προέκυψαν από το FETCH Tool (#related SCE using code relations).

Input Bug Reports (IDs)	**#related bug reports (LSI result)**	**#SCE in the input bug report**	**#related SCE (PS size)**	**#related SCE using code relations**
243653	2613	1	7417	12018
251226	2612	5	7414	12021
254494	592	25	4134	7739
265295	2612	9	7412	12017
268056	2610	12	7412	12020
270797	591	5	4129	7731
273070	590	5	4128	7730
277606	590	14	4128	7725

Πίνακας 6.22: Στοιχεία μετά την εφαρμογή του LSI

Από τον πίνακα αυτό μπορούμε να παρατηρήσουμε ότι παρ' όλο που οι αναφορές αστοχιών, που χρησιμοποιήθηκαν ως είσοδοι για την εφαρμογή της τεχνικής LSI, δεν περιλάμβαναν πολλά SCE, χρησιμοποιώντας τις αναφορές σφαλμάτων που συσχετίστηκαν εννοιολογικά με αυτές, αλλά και τις σχέσεις των SCE

του κώδικα κατορθώνουμε να δημιουργήσουμε ένα ικανοποιητικά μεγάλο σύνολο SCE σχετικών με κάθε μία αναφορά. Το μέγεθος του Extended PS είναι σημαντικός παράγοντας για την επιτυχία του bug localization καθώς τα σύνολα E_i που θα δημιουργηθούν θα περιλαμβάνουν περισσότερες λέξεις και επιπλέον θα είναι περισσότερα σε πλήθος. Όπως έχει ήδη αναφερθεί, κάθε τέτοιο σύνολο συγκρίνεται με το σύνολο E_k της αναφοράς εισόδου που περιλαμβάνει όλα τα SCE και εκείνα που προκύπτουν από την ανάλυση του FETCH Tool. Έτσι λοιπόν προσδίδουμε σε κάθε SCE ένα βάρος και από τη διαδικασία αυτή παίρνουμε τα ακόλουθα αποτελέσματα:

Input Bug Reports (IDs)	**Methods**	**Ranked position in the Sorted List**	**Methods in the Sorted List**
243653	KActionMenu::KConfigGroup::Kopete::Account.edit Account(QWidget*)	Not found	
251226	GoogleTalk.logout(QString)	1	10020
254494	Kopete::MetaContact.setDisplayNameSourceContact (Contact*)	6	6682
265295	GoogleTalk.logout(QString)	1	10017
268056	Client.smt_messageSent()	124	10021
	Client.gotFileMessage(int,QString,QByteArray, Buffer)	287	
	Client.deleteTasks()	74	
270797	TranslatorPlugin.TranslatorPlugin(QObject*, QStringList)	12	6676
	QString::KSelectAction::Kopete::Message::Kopete:: MetaContact::Kopete::ChatSession::TranslatorGUICl ient::TranslatorLanguages::TranslatorPlugin::Translat orPlugin.TranslatorPlugin(QObject*,QStringList)	37	
273070	TranslatorPlugin.TranslatorPlugin(QObject*, QStringList)	12	6674
	QString::KSelectAction::Kopete::Message::Kopete:: MetaContact::Kopete::ChatSession::TranslatorGUICl ient::TranslatorLanguages::TranslatorPlugin::Translat orPlugin.TranslatorPlugin(QObject*,QStringList)	37	
277606	Kopete::MetaContact.setDisplayNameSourceContact (Contact*)	10	6672

Πίνακας 6.23: Αποτελέσματα bug localization

Από τον παραπάνω πίνακα αξίζει να παρατηρήσουμε ότι τα αποτελέσματα του bug localization συστήματος δεν είναι απογοητευτικά για τα bug reports που επιλέχθηκαν. Συγκεκριμένα μόνο μία μέθοδος, που αποτελεί το resolution της αναφοράς 243653, δεν εντοπίστηκε από το σύστημα. Αυτό συνέβη διότι η αναφορά αστοχίας δεν περιείχε πολλές λέξεις κώδικα, παρά μόνο μία, γεγονός που δυσκόλεψε την ανάλυση χρησιμοποιώντας τις σχέσεις μεταξύ των SCE. Αυτό σημαίνει ότι στο επιλεγμένο βάθος, οι λέξεις κώδικα που εντοπίστηκαν δεν περιλαμβάνουν τη συγκεκριμένη μέθοδο.

Στις τρεις επόμενες αναφορές σφαλμάτων –σύμφωνα με τα δεδομένα του πίνακα– οι μέθοδοι που χρειάστηκαν τροποποίηση εντοπίστηκαν με επιτυχία από το σύστημα και κατατάχθηκαν στις πρώτες θέσεις της ταξινομημένης λίστας. Συγκεκριμένα τα ποσοστά επιτυχίας είναι 0.009%, 0.089% και 0.009% αντίστοιχα.

Ακολουθεί η αναφορά με id 268056 για την οποία η ταξινομημένη λίστα έχει αποτελείται από 10021 μεθόδους. Παρ' όλο που με την πρώτη ματιά φαίνεται να μην είναι ιδιαίτερα ικανοποιητικά τα αποτελέσματα, αν τα συγκρίνουμε με το μέγεθος της λίστας θα παρατηρήσουμε οι μέθοδοι που εντοπίστηκαν ότι ανήκουν στο 1.23%, στο 2.86% και στο 0.73% αντίστοιχα. Επομένως η μέθοδος με τη χειρότερη κατάταξη δηλαδή η Client.gotFileMessage(int,QString,QByteArray, Buffer) που εμφανίζεται 287^{η} στις 10021, ανήκει σχεδόν στο 3% των αποτελεσμάτων.

Στη συνέχεια έχουμε τις επόμενες δύο αναφορές αστοχιών, 270797 και 273070, οι οποίες έχουν τις ίδιες μεθόδους για resolution. Στις μεθόδους αυτές αποδόθηκε περίπου το ίδιο βάρος και στις δύο αναλύσεις. Από τον προηγούμενο πίνακα με τα στοιχεία έπειτα από την εφαρμογή της τεχνικής LSI βλέπουμε ότι οι δύο αναφορές έχουν ίδιο αριθμό αναφορών που έχουν συσχετιστεί με αυτές –για την ακρίβεια διαφέρουν κατά ένα related bug report. Επιπροσθέτως οι δύο αναφορές σφαλμάτων έχουν το ίδιο πλήθος SCE και κατ' επέκταση οι λέξεις κώδικα που προέκυψαν μετά την ανάλυση και χρησιμοποιώντας τις σχέσεις του FETCH Tool θα είναι ίδιες και για τις δύο αναφορές σφαλμάτων. Όλα τα παραπάνω συντελούν στο γεγονός ότι οι μέθοδοι αυτές κατατάχθηκαν στις ίδιες θέσεις στις αντίστοιχες ταξινομημένες λίστες τους με τη χειρότερη να ανήκει στο 0.55%.

Η τελευταία αναφορά που χρησιμοποιήθηκε στην ανάλυση του bug localization συστήματος έχει id 277606 και η μέθοδος που τροποποιήθηκε κατατάχθηκε στην 10^{η} θέση στις 6672, δηλαδή στο 0.15%.

Συμπεραίνουμε λοιπόν ότι τα αποτελέσματα του συστήματος για το Kopete ανήκουν όλα σε υψηλές θέσεις και πιο συγκεκριμένα όλα είναι σε ποσοστό μικρότερο του 2.86%. Με βάση αυτό τα ποσοστά του recall περιμένουμε να είναι αρκετά υψηλά κοντά στο 100%, ενώ εκείνα του precision θα είναι αρκετά μικρά επειδή οι μέθοδοι που ανακτώνται και «συναγωνίζονται» στην ταξινομημένη λίστα είναι πολλές (από 6676 έως 10021).

Στην επόμενη σελίδα ακολουθεί ένας πίνακας, παρόμοιος με τον προηγούμενο, που περιέχει τις αναφορές αστοχιών, τις μεθόδους που έπρεπε να εντοπιστούν από το σύστημα και τη θέση τους στην ταξινομημένη λίστα των μεθόδων. Επιπλέον θα περιλαμβάνει και το ποσοστό του precision και recall για καθένα bug report που αναλύθηκε.

Bug Reports (IDs)	Methods	Ranked position in list / number of methods in list	Precision (%)	Recall (%)
243653	KActionMenu::KConfigGroup::Kopete::Account.editAccount(QWidget*)	Not found	0	0
251226	GoogleTalk.logout(QString)	1/10020	0.0099	100
254494	Kopete::MetaContact.setDisplayNameSourceContact(Contact*)	6/6682	0.0897	100
265295	GoogleTalk.logout(QString)	1/10017	0.0099	100
268056	Client.smt_messageSent()	124/10021	0.0299	100
	Client.gotFileMessage(int,QString,QByteArray, Buffer)	287/10021		
	Client.deleteTasks()	74/10021		
270797	TranslatorPlugin.TranslatorPlugin(QObject*, QStringList)	12/6676	0.0299	100
	QString::KSelectAction::Kopete::Message::Kopete::MetaContact::Kopete::ChatSession::TranslatorGUIClient::TranslatorLanguages::TranslatorPlugin::TranslatorPlugin.TranslatorPlugin(QObject*,QStringList)	37/6676		
273070	TranslatorPlugin.TranslatorPlugin(QObject*, QStringList)	12/6674	0.0299	100
	QString::KSelectAction::Kopete::Message::Kopete::MetaContact::Kopete::ChatSession::TranslatorGUIClient::TranslatorLanguages::TranslatorPlugin::TranslatorPlugin.TranslatorPlugin(QObject*,QStringList)	37/6674		
277606	Kopete::MetaContact.setDisplayNameSourceContact(Contact*)	10/6672	0.0149	100

Πίνακας 6.24: Precision και Recall για τα αποτελέσματα bug localization

Από τα δεδομένα του πίνακα γίνεται αντιληπτό ότι το ποσοστό του precision δεν είναι ικανοποιητικό, αφού είναι αρκετά μικρό. Το γεγονός αυτό δικαιολογείται όμως από το μεγάλό αριθμό μεθόδων που ανακτώνται από το σύστημα. Η διαδικασία ανάκτησης λαμβάνει υπόψιν της τις σχέσεις που έχουν προκύψει από την ανάλυση του κώδικα με το FETCH Tool καθώς και το βάθος το οποίο έχει οριστεί. Επιπλέον παίζει σημαντικό ρόλο και το πλήθος των αναφορών αστοχιών που εμφανίζουν εννοιολογική ομοιότητα με την αναφορά εισόδου. Τα στοιχεία του πίνακα 6.22 επιβεβαιώνουν ότι πολλές αναφορές συσχετίζονται (για τις περισσότερες αναφορές εισόδου είναι πάνω από 2000) με καθεμία αναφορά εισόδου. Απόρροια αυτού είναι η δημιουργία μεγάλων ταξινομημένων λιστών που περιέχουν πολλές περισσότερες μεθόδους από εκείνες που χρειαζόμαστε. Σε αντίθεση με το ποσοστό του precision,

έρχεται το recall, το οποίο είναι 100% για όλες τις αναφορές –εκτός από την πρώτη– που σημαίνει ότι όλες οι μέθοδοι που αποτελούν το resolution τους εντοπίζονται.

Εξαιτίας του υπέρογκου συνόλου μεθόδων που δημιουργείται για κάθε αναφορά εισόδου, είναι επιτακτική η ανάγκη να διατηρηθεί ένα ποσοστό αυτό ως έξοδος στο χρήστη. Παρακάτω λοιπόν παρουσιάζονται τα αποτελέσματα (precision, recall) για κάθε μία από τις προηγούμενες αναφορές εισόδου, όταν επικεντρωθούμε στο 3%, 5% και 7% της τελικής ταξινομημένης λίστας.

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
243653	0	0	300
251226	0.33	100	300
254494	0.5	100	200
265295	0.33	100	300
268056	0.667	66.67	300
270797	1	100	200
273070	1	100	200
277606	0.5	100	200

Πίνακας 6.25: Precision και Recall για το 3% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
243653	0	0	500
251226	0.199	100	501
254494	0.299	100	334
265295	0.2	100	500
268056	0.399	66.67	501
270797	0.601	100	333
273070	0.601	100	333
277606	0.3	100	333

Πίνακας 6.26: Precision και Recall για το 5% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
243653	0	0	701
251226	0.143	100	701
254494	0.214	100	467
265295	0.143	100	701
268056	0.285	66.67	701
270797	0.428	100	467
273070	0.428	100	467
277606	0.214	100	467

Πίνακας 6.27: Precision και Recall για το 7% των μεθόδων της ταξινομημένης λίστας

Μία αρχική, αλλά σημαντική, παρατήρηση είναι ότι ο αριθμός των μεθόδων (number of methods in ranked list) που δίνονται ως έξοδοι από το σύστημα που υλοποιήθηκε για την αναγνώριση των προβληματικών σημείων του κώδικα, είναι αρκετά μεγάλος ακόμη και για περιορισμένο αριθμό του τελικού συνόλου των μεθόδων. Κάτι τέτοιο όμως είναι λογικό, εάν ληφθεί υπόψιν το γεγονός ότι το μικρότερο σύνολο που δημιουργήθηκε είναι 6674 και το μεγαλύτερο είναι 10021. Άμεση συνέπεια αυτού είναι και τα ποσοστά του precision που παραμένουν χαμηλά.

Παρ' όλο που το precision είναι χαμηλό, παρατηρούμε σημαντική άνοδο σε σχέση με τα δεδομένα του πίνακα 6.24, γεγονός που επιβεβαιώνει ότι μικρότερο –και συνεπώς ευκολότερα διαχειρίσιμο σύνολο– έχει καλύτερα αποτελέσματα ακρίβειας συγκριτικά με το 100% των μεθόδων που ανακτώνται. Επιπλέον, όσο αυξάνεται το πλήθος των μεθόδων που συγκεντρώνονται, τόσο περισσότερο ελαττώνεται η τιμή του precision. Όπως έχει αναφερθεί και σε προηγούμενες αναλύσεις αυτό συμβαίνει επειδή αυξάνοντας το σύνολο των μεθόδων υπάρχει μεγαλύτερη πιθανότητα να ανακτηθούν μέθοδοι που δεν σχετίζονται με το resolution που αναζητούμε.

Στο σημείο αυτό θα επικεντρωθούμε στις αναφορές αστοχίας που παρουσιάζουν χαμηλό ποσοστό recall. Η μέθοδος που αποτελεί το resolution της αναφοράς αστοχίας με id 243653 δεν εντοπίστηκε από το bug localization σύστημα για το 100% των μεθόδων της ταξινομημένης λίστας. Επομένως είναι βέβαιο ότι και για μικρότερα ποσοστά αυτής, δε θα έχουμε επιτυχία και το ποσοστό τόσο του precision όσο και του recall για την συγκεκριμένη αναφορά θα είναι 0%.

Η αναφορά αστοχίας 268056 για το 100% του συνόλου έχει 100% recall, που σημαίνει ότι και οι τρεις μέθοδοι που χρειάστηκαν τροποποίηση για την επίλυση του σφάλματος, εντοπίστηκαν από το σύστημα που υλοποιήθηκε. Όμως όπως φαίνεται στον πίνακα 6.24 κατατάχθηκαν στις θέσεις 124 και 287 μεταξύ 10021 άλλων αναφορών. Αποτέλεσμα αυτού είναι η μία από αυτές να μην συμπεριλαμβάνεται στο 3%, 5% και 7% και έτσι το ποσοστό του recall να είναι 66.67%.

6.2.5 Εφαρμογή στο GTK+

Το GTK+ είναι μία εργαλειοθήκη, που αποτελεί μέρος του GNU Project, σχεδιασμένη για πολλές πλατφόρμες (multi-platform toolkit) για τη δημιουργία γραφικών διεπαφών χρήστη (graphical user interface) και είναι κατάλληλο τόσο για μικρά projects όσο και για ολοκληρωμένες εφαρμογές. Το gtk+ είναι γραμμένο στην γλώσσα προγραμματισμού C, παρ' όλα αυτά όμως έχει υλοποιηθεί έτσι ώστε να υποστηρίζει και άλλες γλώσσες, όχι μόνο C/C++, αλλά ακόμη και Perl και Python.

Το πρώτο στάδιο στο επίπεδο προ-επεξεργασίας των αναφορών αστοχιών είναι η συλλογή των αναφορών από το Bugzilla repository. Συγκεκριμένα για το gtk+ 3.19, όπως φαίνεται στα δεδομένα που παρουσιάζονται στον ακόλουθο πίνακα, ανακτήθηκαν χρησιμοποιώντας το Bugzilla API συνολικά 21803 αναφορές σφαλμάτων από τις οποίες οι 21803 περιείχαν σχόλια. Επόμενο στάδιο είναι ένας επιπλέον διαχωρισμός των bug reports και η διαλογή εκείνων που στα σχόλιά τους περιλαμβάνονται λέξεις κώδικα (14008 σε πλήθος). Αυτές οι αναφορές είναι εκείνες που θα συσχετιστούν μεταξύ τους προκειμένου να δημιουργηθούν τα clusters.

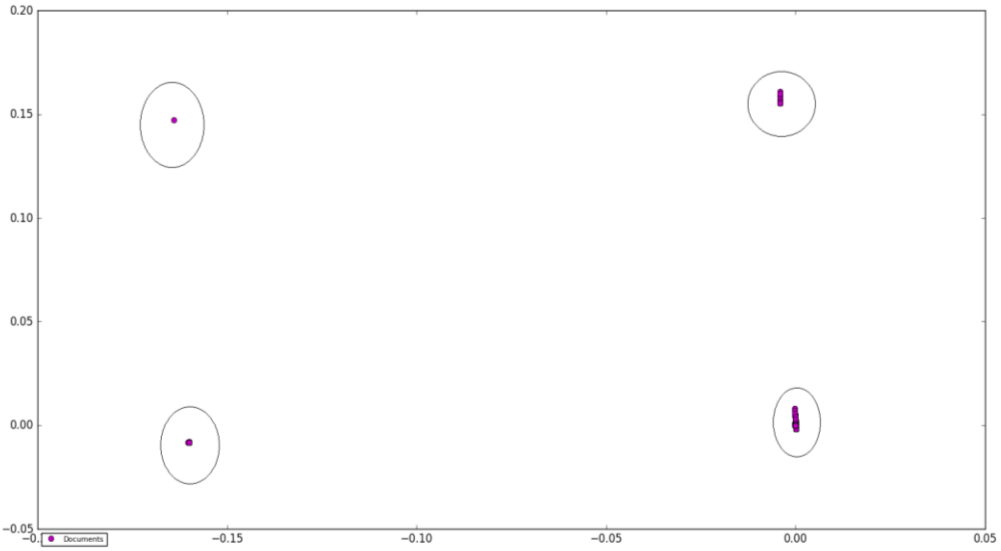
Number of bug reports retrieved from Bugzilla	21803
Number of bug reports with comments	21803
Number of bug reports including SCE in comments	14008
Number of bug reports that would be related with LSI input	14007

Πίνακας 6.28: Πλήθος των αναφορών αστοχιών που χρησιμοποιήθηκαν στα στάδια της προ-επεξεργασίας τους

Στη συνέχεια, επιλέχθηκαν τυχαία δεκατρείς (13) αναφορές, κάθε μία εκ των οποίων θα χρησιμοποιηθεί ως είσοδος για την τεχνική LSI και επομένως θα συσχετιστεί με τις υπόλοιπες 14007 αναφορές. Για τις αναφορές αστοχιών εισόδου, γνωρίζουμε τις μεθόδους που έχουν τροποποιηθεί προκειμένου να επιλυθεί το πρόβλημα που περιγράφεται. Αυτό μας εξυπηρετεί στο στάδιο της αξιολόγησης του bug localization συστήματος, καθώς θα μπορέσουμε να συγκρίνουμε τις μεθόδους αυτές με τα αποτελέσματα της ταξινομημένης λίστας που δίνει το σύστημά μας.

Όπως είναι γνωστό από προηγούμενες αναφορές στην τεχνική LSI, ο σκοπός της είναι η σημασιολογική συσχέτιση των κειμένων που υπάρχουν σε ένα repository –στην προκειμένη περίπτωση οι 14007 αναφορές σφαλμάτων– με το κείμενο εισόδου, δηλαδή την καθεμία από τις δεκατρείς αναφορές. Είναι προφανές ότι δεν γίνεται η αναφορά εισόδου να ταιριάζει εννοιολογικά και με τις 14007 αναφορές του gtk+. Έτσι λοιπόν το LSI δημιουργεί ομάδες (clusters) με τις αναφορές που σχετίζονται μεταξύ τους και τοποθετεί την αναφορά εισόδου στο cluster με τα bug reports που εμφανίζουν τη μεγαλύτερη ομοιότητα.

Στην εικόνα που ακολουθεί φαίνονται τα clusters που δημιουργήθηκαν για το gtk+ μετά την εφαρμογή του LSI.



Εικόνα 6.5.: Clustering για τις αναφορές αστοχιών του GTK+

Από το προηγούμενο διάγραμμα μπορούμε να παρατηρήσουμε ότι στην περίπτωση του gtk+ οι αναφορές αστοχιών ομαδοποιούνται και διαχωρίζονται ξεκάθαρα μεταξύ τους. Πιο συγκεκριμένα δημιουργούνται clusters τα οποία έχουν διακριτά όρια το ένα με το άλλο, καθώς δεν υπάρχει αναφορά σφάλματος που είναι πιθανό να ανήκει σε περισσότερα από ένα clusters ή ακόμη και να βρίσκεται στα όρια δύο ομάδων. Αιτία αυτού του γεγονότος είναι ότι τα bugs που έχουν καταγραφεί για το gtk+ οφείλονται σε παρόμοια σφάλματα τα οποία αναγνωρίζονται από τον αλγόριθμο LSI που δημιουργεί τις 4 κατηγορίες που φαίνονται στην εικόνα.

Input Bug Reports (IDs)	**#related bug reports (LSI result)**	**#SCE in the input bug report**	**#related SCE (PS size)**	**#related SCE using code relations**
757282	10992	1	14261	22853
757805	11241	4	14314	22883
758442	11624	12	14446	22990
758609	11006	2	14273	22863
758901	1720	3	6495	19014
759091	10936	2	14244	22850
759299	11439	7	14361	22918
759705	11082	132	14277	22865
759764	10984	5	14249	22852
760213	10975	15	14236	22843
760640	11695	2	14469	23009
760942	245	1	1851	9167
761128	10816	2	14226	22822

Πίνακας 6.29: Στοιχεία μετά την εφαρμογή του LSI

Επόμενο βήμα είναι η καταγραφή των δεδομένων μετά το πέρας της εφαρμογής του LSI. Στον προηγούμενο πίνακα λοιπόν, που είναι ο αντίστοιχος με τις προηγούμενες ενότητες, βρίσκονται πληροφορίες σχετικές με τον αριθμό των SCE που εντοπίστηκαν σε κάθε μία από τις 13 αναφορές, το πλήθος των αναφορών που συσχετίζονται εννοιολογικά με καθένα bug report εισόδου καθώς και το πλήθος των SCE που αποτελούν το Problem Set και το Extended Problem Set, χρησιμοποιώντας τις σχέσεις του FETCH Tool.

Αυτό που παρουσιάζει το διάγραμμα του LSI επιβεβαιώνεται και από τα στοιχεία του πίνακα. Συγκεκριμένα παρατηρούμε ότι για κάθε αναφορά σφάλματος εισόδου, υπάρχουν πολλές άλλες αναφορές (περίπου 14.000), οι οποίες σχετίζονται εννοιολογικά με την πρώτη, με εξαίρεση βέβαια δύο αναφορές που συσχετίζονται με λιγότερες. Επιπλέον είναι σημαντικό το γεγονός ότι παρά τις ελάχιστες λέξεις κώδικα που περιλαμβάνει κάθε αναφορά αστοχίας εισόδου, το σύνολο Extended PS είναι αρκετά μεγάλο σε όλες τις περιπτώσεις, πράγμα που σημαίνει ότι υπάρχουν περισσότερες λέξεις και συνεπώς περισσότερα σύνολα E_i τα οποία συγκρίνονται με το σύνολο E_k. Τα αποτελέσματα της σύγκρισης αυτής μας δίνουν τη λίστα με τις μεθόδους του πηγαίου κώδικα που είναι πιθανό να χρειάζονται τροποποίηση για να επιλυθεί το σφάλμα.

Στον πίνακα της επόμενης σελίδας παρουσιάζονται αναλυτικά τα αποτελέσματα του bug localization συστήματός μας. Σε γενικές γραμμές παρατηρούμε ότι τα αποτελέσματα είναι αρκετά ικανοποιητικά γεγονός που οφείλεται στην ύπαρξη των τεσσάρων διακριτών ομάδων bug reports (για την ακρίβεια τριών ομάδων και μία αναφορά μόνη της). Έτσι κάθε bug report ομαδοποιείται αμέσως σε μία «οικογένεια» αναφορών που χρησιμοποιούνται για την ανάκτηση και ταξινόμηση των μεθόδων. Παρ' όλο που υπάρχουν μέθοδοι οι οποίες φαίνεται να είναι πολύ χαμηλά στην ταξινομημένη λίστα, στην πραγματικότητα βρίσκονται σε καλό ποσοστό σε σχέση με τον αριθμό εκείνων που έχουν ανακτηθεί και συγκεκριμένα όλες βρίσκονται κάτω από το 1.085% του συνολικού αριθμού μεθόδων (περίπου 21.000 μέθοδοι σε όλες τις αναφορές σφαλμάτων).

Πιο συγκεκριμένα παρατηρούμε ότι για τις αναφορές αστοχιών με αναγνωριστικό αριθμό: 757805, 758901, 759091, 759705 και 760942 τα αποτελέσματα του bug localization περιλαμβάνουν τις μεθόδους που χρειάστηκαν τροποποίηση στις θέσεις της πρώτης δεκάδας της λίστας των μεθόδων, με εξαίρεση τη μέθοδο get_tab_at_pos(GtkNotebook*,gint,gint) της αναφοράς 759091 που κατατάχθηκε στη θέση 11 καθώς επίσης και τη μέθοδο gtk_window_show(GtkWidget*) της αναφοράς 759705 που βρίσκεται στη θέση 22.

Αρχίζοντας από την πρώτη αναφορά αστοχίας (id 757282), το resolution της οποίας περιλαμβάνει τρεις μεθόδους, σύμφωνα με τα στοιχεία του ακόλουθου πίνακα δύο από τις μεθόδους αυτές κατατάσσονται στην πρώτη δεκάδα του συνολικού αριθμού των μεθόδων (21555), ενώ η άλλη στη θέση 94. Η θέση αυτή αντιστοιχεί στο 0.43% των μεθόδων που ανακτήθηκαν γεγονός που υποδεικνύει ότι δεν είναι εξαιρετικά άσχημη η κατάταξή της. Παρατηρούμε όμως ότι λεκτικά παρουσιάζει ομοιότητες σε σχέση με τις άλλες μεθόδους, καθώς ξεκινούν όλες από "gtk_window" και αυτό ενισχύει την πιθανότητα να βρεθεί σε κάποια καλύτερη θέση αν εφαρμοστεί μία τεχνική που να λαμβάνει υπόψη της τέτοιου είδους λεκτικές συσχετίσεις.

Input Bug Reports (IDs)	Methods	Ranked position in the Sorted List	Methods in the Sorted List
757282	gtk_window_resize(GtkWindow*,gint,gint)	9	21555
	gtk_window_compute_configure_request_size(GtkWindow*,GdkGeometry*,guint,gint*,gint*)	94	
	gtk_window_resize_to_geometry(GtkWindow*, gint,gint)	3	
757805	get_shadow_width(GtkWindow*,GtkBorder*)	2	21572
	_gtk_window_get_shadow_width(GtkWindow*,G tkBorder*)	3	
758442	gtk_style_context_set_path(GtkStyleContext*, GtkWidgetPath*)	13	21651
	gtk_css_node_declaration_add_to_widget_path(Gt kCssNodeDeclaration*,GtkWidgetPath*,guint)	235	
758609	gtk_window_show(GtkWidget*)	171	21563
	gtk_window_move(GtkWindow*,gint,gint)	2	
758901	gdk_wayland_window_configure(GdkWindow*,in t,int,int)	5	18407
759091	get_widget_coordinates(GtkWidget*,GdkEvent*,g int*,gint*)	5	21555
	tab_prelight(GtkNotebook*,GdkEvent*)	1	
	get_tab_at_pos(GtkNotebook*,gint,gint)	11	
	tk_notebook_leave_notify(GtkWidget*,GdkEvent Crossing*)	2	
759299	gdk_wayland_window_set_transient_for (GdkWindow*,GdkWindow*)	12	21602
759705	gtk_window_show(GtkWidget*)	22	21564
	gtk_window_realize(GtkWidget*)	5	
759764	warn_response(GtkDialog*,gint)	6	21555
760213	gdk_wayland_device_set_window_cursor (GdkDevice*,GdkWindow*,GdkCursor*)	20	21548
760640	gtk_notebook_destroy(GtkWidget*)	83	21659
760942	paint_border_window(GtkTextView*,cairo_t*, G_BEGIN_DECLSGtkTextWindowType,GtkStyl eContext*,char)	10	9053
761128	gtk_entry_draw_text(GtkEntry*,cairo_t*)	132	21535

Πίνακας 6.30: Αποτελέσματα bug localization

Όσο αφορά στα bug reports 758442, 760640 και 761128, συγκρίνοντας τα αποτελέσματα του πίνακα με εκείνα των υπόλοιπων αναφορών, συμπεραίνουμε ότι η θέση των μεθόδων στην ταξινομημένη λίστα είναι χαμηλή, παρ' όλο που η μέθοδος με τη χαμηλότερη θέση κάθε μίας βρίσκεται στο 1.085%, 0.383% και 0.6129% αντίστοιχα. Το γεγονός αυτό μπορεί να οφείλεται κατά κύριο λόγο στον τεράστιο όγκο δεδομένων (μεθόδων) που ανακτήθηκαν μέσω των relations του FETCH Tool, όπως επίσης και στο βάθος (depth=1) που επιλέχθηκε. Επιπροσθέτως αξίζει να τονίσουμε ότι το πλήθος των SCE και συγκεκριμένα των μεθόδων που ανακτήθηκαν είναι αρκετά μεγάλο διότι έχουν δημιουργηθεί μόνο τέσσερεις ομάδες και όπως είναι προφανές η αντιστοιχία bug report με ένα cluster συνεπάγεται αυτόματα και τον συσχετισμό της με τις αναφορές που ανήκουν στο cluster αυτό και που εξαιτίας του μικρού αριθμού ομάδων, είναι πάρα πολλές. Αποτέλεσμα αυτού είναι να αυξάνεται και το πλήθος των SCE και κατά συνέπεια και των μεθόδων, γεγονός που δυσχεραίνει το διαχωρισμό τους σε σημαντικότερες και μη.

Για την επίλυση του σφάλματος της αναφορά αστοχίας 758609, σύμφωνα με το Bugzilla, χρειάστηκαν τροποποίηση στον πηγαίο κώδικα οι δύο μέθοδοι που αναφέρονται στον προηγούμενο πίνακα. Η μία από αυτές κατατάχθηκε από το bug localization σύστημά μας στη δεύτερη θέση της ταξινομημένης λίστας, που αντιστοιχεί στο 0.009% των μεθόδων που ανακτήθηκαν, ενώ αντίθετα η δεύτερη κατατάχθηκε στη θέση 171 στις 21563 μεθόδους, δηλαδή στο 0.79%. Εξαιτίας του μεγάλου αριθμού SCE που προκύπτουν απ' τις τόσες αναφορές αστοχιών, οι οποίες συσχετίζονται με την αναφορά εισόδου, είναι λογικό κάποιες να θεωρήθηκαν περισσότερο σημαντικές από τη συγκεκριμένη μέθοδο λόγω της μεγαλύτερης ομοιότητας που εμφάνισαν με τις λέξεις της αρχικής αναφοράς. Όμως μπορούμε να παρατηρήσουμε ότι οι δύο αυτές μέθοδοι που αποτελούν το resolution της αναφοράς 758609 παρουσιάζουν λεκτική ομοιότητα μεταξύ τους με εξαίρεση τις λέξεις move και show, γεγονός που ενισχύει την θεωρία ότι μία προσέγγιση με βάση την εννοιολογική σημασία των μεθόδων θα μπορούσε να βελτιώσει το εν λόγω αποτέλεσμα.

Οι επόμενες δύο αναφορές που υπάρχουν στον πίνακα και δεν σχολιάστηκαν ακόμη είναι οι 759299 και 759764 έχουν η καθεμία από μία μέθοδο που είναι βασική για την επίλυση του εκάστοτε σφάλματος. Και οι δύο μέθοδοι βρίσκονται σε πολύ καλές θέσεις στην ταξινομημένη λίστα του bug localization συστήματος. Για την ακρίβεια η μέθοδος που αντιστοιχεί στην αναφορά αστοχίας 759299 είναι η gdk_wayland_window_set_transient_for (GdkWindow*,GdkWindow*) και κατέχει τη θέση 12 στις 21602 μεθόδους που ανακτήθηκαν. Ακόμη καλύτερη θέση έχει η μέθοδος warn_response(GtkDialog*,gint) του bug report 759764, που κατατάσσεται έκτη στις 21555 μεθόδους.

Παρακάτω ακολουθεί ένας ακόμη πίνακας που περιλαμβάνει την ίδια πληροφορία με τον προηγούμενο και επιπλέον το ποσοστό precision και recall για κάθε μία αναφορά που χρησιμοποιήθηκε ως είσοδος στο bug localization σύστημα. Όπως είναι αναμενόμενο το ποσοστό του precision θα είναι ιδιαίτερα μικρό, επειδή το bug localization σύστημα προσθέτει στη λίστα πολλές μεθόδους, ενώ εκείνες που αναζητούμε είναι λίγες. Για να μειωθεί ο αριθμός των SCE και κατά συνέπεια να αυξηθεί το ποσοστό του precision, θα μπορούσαμε να επιλέξουμε ένα ποσοστό των συνολικών μεθόδων που ανακτώνται. Παρ' όλα αυτά επειδή όλες οι μέθοδοι εντοπίζονται με επιτυχία, γι' αυτό το ποσοστό του recall είναι σταθερό στο 100%.

Input Bug Reports (IDs)	Methods	Ranked position in list / number of methods in list	Precision (%)	Recall (%)
757282	gtk_window_resize(GtkWindow*,gint,gint)	9/21555	0.0139	100
	gtk_window_compute_configure_request_size(GtkWindow*,GdkGeometry*,guint,gint*,gint*)	94/21555		
	gtk_window_resize_to_geometry(GtkWindow*, gint,gint)	3/21555		
757805	get_shadow_width(GtkWindow*,GtkBorder*)	2/21572	0.0092	100
	_gtk_window_get_shadow_width(GtkWindow*,G tkBorder*)	3/21572		
758442	gtk_style_context_set_path(GtkStyleContext*, GtkWidgetPath*)	13/21651	0.0092	100
	gtk_css_node_declaration_add_to_widget_path(Gt kCssNodeDeclaration*,GtkWidgetPath*,guint)	235/21651		
758609	gtk_window_show(GtkWidget*)	171/21563	0.0092	100
	gtk_window_move(GtkWindow*,gint,gint)	2/21563		
758901	gdk_wayland_window_configure(GdkWindow*, int,int,int)	5/18407	0.0054	100
759091	get_widget_coordinates(GtkWidget*,GdkEvent*,g int*,gint*)	5/21555	0.018	100
	tab_prelight(GtkNotebook*,GdkEvent*)	1/21555		
	get_tab_at_pos(GtkNotebook*,gint,gint)	11/21555		
	tk_notebook_leave_notify(GtkWidget*,GdkEvent Crossing*)	2/21555		
759299	gdk_wayland_window_set_transient_for (GdkWindow*,GdkWindow*)	12/21602	0.0046	100
759705	gtk_window_show(GtkWidget*)	22/21564	0.0092	100
	gtk_window_realize(GtkWidget*)	5/21564		
759764	warn_response(GtkDialog*,gint)	6/21555	0.0046	100
760213	gdk_wayland_device_set_window_cursor (GdkDevice*,GdkWindow*,GdkCursor*)	20/21548	0.0046	100
760640	gtk_notebook_destroy(GtkWidget*)	83/21659	0.0046	100
760942	paint_border_window(GtkTextView*,cairo_t*, G_BEGIN_DECLSGtkTextWindowType,GtkStyl eContext*,char)	10/9053	0.0111	100
761128	gtk_entry_draw_text(GtkEntry*,cairo_t*)	132/21535	0.0046	100

Πίνακας 6.31: Precision και Recall για τα αποτελέσματα bug localization

Έτσι λοιπόν για τις δεκατρείς μεθόδους που χρησιμοποιήθηκαν σαν είσοδοι στο bug localization σύστημα, κρατάμε το 3%, 5% και 7% του συνόλου της ταξινομημένης λίστας που δημιουργείται. Παρακάτω ακολουθούν οι πίνακες με το precision και recall για κάθε μία από αυτές:

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
757282	0.464	100	647
757805	0.309	100	648
758442	0.308	100	650
758609	0.309	100	647
758901	0.181	100	552
759091	0.618	100	647
759299	0.154	100	648
759705	0.309	100	647
759764	0.155	100	647
760213	0.155	100	646
760640	0.154	100	648
760942	0.369	100	271
761128	0.154	100	646

Πίνακας 6.32: Precision και Recall για το 3% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
757282	0.278	100	1078
757805	0.185	100	1079
758442	0.185	100	1083
758609	0.186	100	1078
758901	0.109	100	920
759091	0.371	100	1078
759299	0.093	100	1080
759705	0.186	100	1078
759764	0.092	100	1078
760213	0.093	100	1077
760640	0.092	100	1083
760942	0.221	100	453
761128	0.093	100	1077

Πίνακας 6.33: Precision και Recall για το 5% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
757282	0.199	100	1509
757805	0.133	100	1510
758442	0.132	100	1516
758609	0.133	100	1509
758901	0.077	100	1288
759091	0.265	100	1509
759299	0.066	100	1512
759705	0.133	100	1509
759764	0.066	100	1509
760213	0.066	100	1508
760640	0.066	100	1516
760942	0.158	100	634
761128	0.066	100	1507

Πίνακας 6.34: Precision και Recall για το 7% των μεθόδων της ταξινομημένης λίστας

Τα συμπεράσματα στα οποία καταλήγουμε είναι παρόμοια με εκείνα των άλλων εφαρμογών. Τα ποσοστά του precision είναι αρκετά μικρά και αυτό οφείλεται στον μεγάλο αριθμό μεθόδων που συγκεντρώνονται στην εκάστοτε ταξινομημένη λίστα. Όπως παρατηρούμε στον πίνακα 6.31, ήδη για το αρχικό σύνολο, οι μέθοδοι που ανακτώνται κυμαίνονται από 18407 έως και 21659 με μόνη εξαίρεση την αναφορά αστοχίας 760942 για την οποία δημιουργήθηκε σύνολο της τάξης των 9053 μεθόδων. Συνεπώς ποσοστά τέτοιας τάξης για το precision είναι δικαιολογημένα.

Παρά το μικρό του ποσοστό όμως βλέπουμε ότι όσο περιορίζουμε το μέγεθος της ταξινομημένης λίστας, τόσο βελτιώνονται και οι τιμές precision. Όπως τις προηγούμενες φορές, έτσι και τώρα, αυτό συμβαίνει επειδή αναζητούμε τον ίδιο αριθμό μεθόδων σε μικρότερο σύνολο από το αρχικό. Για το 3% των μεθόδων παρατηρούμε ότι το precision για κάθε αναφορά σφάλματος άλλες φορές δεκαπλάσια και άλλες ακόμη και εκατόνταπλασιάζεται. Αυτό εξαρτάται και από το πλήθος των μεθόδων που αποτελούν το resolution κάθε αναφοράς εισόδου και είναι εκείνες που αναζητούμε στην ταξινομημένη λίστα. Προφανώς, όσο περισσότερες είναι οι μέθοδοι αυτές, τόσο μεγαλύτερο είναι και το precision. Αντίστοιχα για το 5% και 7% τα ποσοστά είναι σχεδόν δεκαπλάσια από τα αρχικά.

Σχετικά με το recall αξίζει να παρατηρήσουμε ότι για το 100% του συνόλου των μεθόδων, η θέση εκείνων που αναζητούμε στη λίστα είναι αρκετά υψηλή σε σχέση με το πλήθος αυτών που ανακτήθηκαν. Το ίδιο, όπως ήταν αναμενόμενο, συμβαίνει και στο 3%, 5% και 7% και επομένως το ποσοστό του recall παραμένει 100% για όλες τις αναφορές αστοχιών εισόδου.

6.2.6 Εφαρμογή στο Nautilus

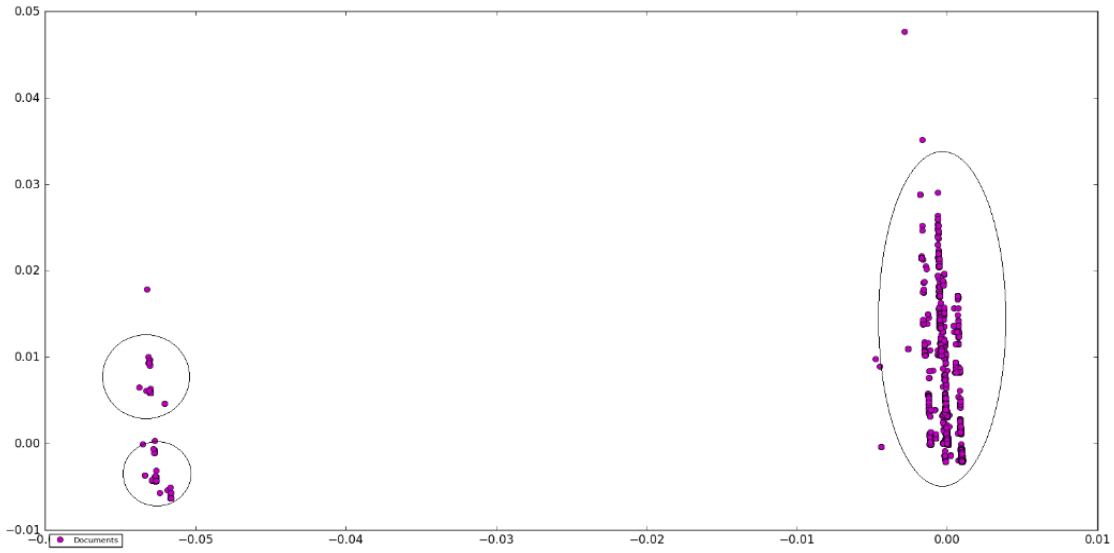
Το Nautilus είναι η επίσημη εφαρμογή διαχείρισης αρχείων (File Manager) ανοιχτού κώδικα (open source) σχεδιασμένη για την επιφάνεια εργασίας του GNOME. Αντικατέστησε το Midnight Commander στο GNOME 1.4 το 2001 και έκτοτε είναι ο προεπιλεγμένος διαχειριστής αρχείων του GNOME. Βασικό χαρακτηριστικό του Nautilus είναι ότι υποστηρίζει την περιήγηση τοπικών αρχείων, καθώς και συστήματα αρχείων που διατίθενται μέσω FTP, HTTP, WebDAV και SFTP διακομιστές. Επιπλέον, ο χρήστης έχει τη δυνατότητα να δημιουργεί σελιδοδείκτες και να παρακολουθεί το ιστορικό των φακέλων που επισκέφθηκε, όπως συμβαίνει και με τους Web Browsers, επιτρέποντας εύκολη πρόσβαση σε φακέλους που είχε ήδη περιηγηθεί παλαιότερα.

Η έκδοση 3.8.0 του Nautilus που χρησιμοποιήθηκε είναι εκείνη για την οποία μπορούσαν να ανακτηθούν οι περισσότερες αναφορές σφαλμάτων και ταυτόχρονα να βρεθούν αναφορές για τις οποίες υπάρχει καταγεγραμμένο το resolution τους. Όπως φαίνεται λοιπόν στον ακόλουθο πίνακα, αρχικά συγκεντρώθηκαν από το Bugzilla repository 52134 bug reports. Από αυτά επιλέχθηκαν εκείνα τα οποία περιείχαν σχόλια (number of bug reports with comments) και από αυτά έγινε η διαλογή μεταξύ εκείνων που περιέχουν SCE στα σχόλια των χρηστών (number of bug reports including SCE in comments).

Number of bug reports retrieved from Bugzilla	52134
Number of bug reports with comments	52130
Number of bug reports including SCE in comments	21014
Number of bug reports that would be related with LSI input	21013

Πίνακας 6.35: Πλήθος των αναφορών αστοχιών που χρησιμοποιήθηκαν στα στάδια της προ-επεξεργασίας τους

Παρατηρούμε από τον πίνακα ότι παρ' όλο που το πλήθος των αναφορών αστοχιών είναι 52134 και από αυτές μόνο τέσσερις δεν περιέχουν σχόλια, SCE απαντώνται μόνο σε 21014 bug reports. Αυτό σημαίνει ότι περίπου 30000 αναφορές «απορρίπτονται» γιατί δεν μπορούν να χρησιμοποιηθούν αφού δεν περιέχουν SCE στα σχόλιά τους και έτσι η τεχνική LSI δεν μπορεί να συσχετίσει εννοιολογικά την αναφορά εισόδου με κάποια από αυτές. Εννοιολογικός και σημασιολογικός συσχετισμός σημαίνει ότι οι αναφορές, που ο αλγόριθμος του LSI θα αποφασίσει ότι ανήκουν στην ίδια ομάδα, περιγράφουν το ίδιο ή κάποιο παρόμοιο σφάλμα. Έχοντας λοιπόν αφαιρέσει από την αναφορά εισόδου –συγκεκριμένα για το nautilus και από τις επτά (7) αναφορές που χρησιμοποιήθηκαν– το σχόλιο που αναφέρει τις μεθόδους που αποτελούν το resolution της, η τεχνική LSI δημιουργεί τα clusters στα οποία τοποθετούνται αναφορές με τη μεγαλύτερη ομοιότητα και φαίνονται στην εικόνα που ακολουθεί στην επόμενη σελίδα.



Εικόνα 6.6.: Clustering για τις αναφορές αστοχιών του Nautilus

Κάθε σημείο στο διάγραμμα αντιπροσωπεύει μία αναφορά σφάλματος και όσα από αυτά περικλείονται από την ίδια γραμμή ανήκουν στο ίδιο cluster. Φαίνεται λοιπόν πως μετά την εφαρμογή του αλγόριθμου LSI προκύπτουν συνολικά τρεις ομάδες (clusters), οι δύο εκ των οποίων όμως δεν είναι διακριτές μεταξύ τους, καθώς υπάρχουν αναφορές αστοχιών που βρίσκονται στα όρια των δύο clusters και αυτό δυσχεραίνει την διαδικασία της ομαδοποίησής τους. Το cluster στα αριστερά συγκεντρώνει το μεγαλύτερο πλήθος αναφορών, ενώ στα δύο άλλα clusters κατατάχθηκαν λιγότερες αναφορές αστοχιών. Τέλος, υπάρχουν bug reports που ο αλγόριθμος δεν κατάφερε να τα συσχετίσει εννοιολογικά με καμία ομάδα και βρίσκονται μόνα τους, έξω από τα clusters.

Input Bug Reports (IDs)	**#related bug reports (LSI result)**	**#SCE in the input bug report**	**#related SCE (PS size)**	**#related SCE using code relations**
333265	1164	4	926	3382
697183	871	12	677	2751
697890	16710	3	2496	4481
698190	1935	8	1364	3833
702546	16842	30	2504	4498
703349	1290	14	799	3012
711480	16710	11	2479	4473

Πίνακας 6.36: Στοιχεία μετά την εφαρμογή του LSI

Για τις αναφορές εισόδου που επιλέχτηκαν ο προηγούμενος πίνακας περιέχει πληροφορίες που σχετίζονται με τον αριθμό των SCE που εντοπίστηκαν σε κάθε μία αναφορά εισόδου (#SCE in the input bug report), το πλήθος των αναφορών που συσχετίστηκαν με την εκάστοτε αναφορά (#related bug reports), τον αριθμό των SCE που σχετίζονται με την αναφορά εισόδου και αποτελούν το Problem Set (#related SCE)

και τέλος το σύνολο των SCE που σχετίζονται με την αναφορά χρησιμοποιώντας τις σχέσεις που προέκυψαν από το FETCH Tool (#related SCE using code relations).

Μία πρώτη παρατήρηση είναι ότι τα bug reports που θέσαμε ως είσοδο στο bug localization σύστημα δεν περιλαμβάνουν αρκετές λέξεις κώδικα (κατά μέσο όρο 8 SCE σε κάθε bug report). Μόνη εξαίρεση αποτελεί το 702546 στου οποίου τα σχόλια εμφανίζονται 30 λέξεις κώδικα. Μετά όμως από την εφαρμογή του LSI μερικές αναφορές εισόδου συσχετίζονται τελικά με περίπου 1000 αναφορές του repository, ενώ κάποιες άλλες που φαίνεται να ανήκουν στο μεγάλο cluster παρουσιάζουν ομοιότητα με σχεδόν 16700, γεγονός που αυξάνει τις σχετικές SCE για κάθε μία αναφορά εισόδου. Αυτό είναι ιδιαίτερα χρήσιμο, καθώς έτσι επεκτείνουμε το σύνολο των SCE που θα χρησιμοποιηθεί αργότερα για τη σύγκριση και τον προσδιορισμό βάρους σε κάθε μία από αυτές. Δημιουργείται λοιπόν ένα σύνολο E_k, το οποίο περιλαμβάνει τις λέξεις κώδικα που βρίσκονται στην αναφορά εισόδου, καθώς και εκείνες που προέρχονται από τις σχέσεις του FETCH Tool και συγκρίνεται με τη μετρική Jaccard με κάθε σύνολο E_i που αποτελείται από καθεμία SCE ενός related bug report σε συνδυασμό με άλλες που προέρχονται από τις σχέσεις του FETCH.

Αποτέλεσμα κάθε παραπάνω σύγκρισης είναι ο υπολογισμός ενός βάρους που προσδιορίζει κάθε μέθοδο και υποδεικνύει το πόσο πιθανό είναι να χρειάζεται τροποποίηση για την επίλυση του σφάλματος. Ακολουθεί ο πίνακας με τις μεθόδους αυτές και τη θέση που έχουν στην ταξινομημένη λίστα μεθόδων:

Input Bug Reports (IDs)	**Methods**	**Ranked position in the Sorted List**	**Methods in the Sorted List**
333265	real_update_menus(NautilusView*)	10	3247
697183	apply_columns_settings(NautilusListView*, char**,char**)	6	2695
	create_and_set_up_tree_view(NautilusListView*)	3	
697890	filtering_changed_callback(gpointer)	23	3673
	invalidate_one_count(gpointer,gpointer, gpointer)	88	
	collect_all_directories(gpointer,gpointer, gpointer)	25	
698190	column_chooser_use_default_callback (NautilusColumnChooser*,NautilusListView*)	1	3673
	get_column_order(NautilusListView*)	7	
	get_visible_columns(NautilusListView*)	8	
	nautilus_list_view_reset_to_defaults (NautilusView*)	3	
702546	nautilus_window_slot_dispose(GObject*)	5	4284
71480	custom_icon_file_chooser_response_cb (GtkDialog*,gint,NautilusPropertiesWindow*)	73	4267
703349	append_directory_contents_fields (NautilusPropertiesWindow*,GtkGrid*)	143	2914

Πίνακας 6.37: Αποτελέσματα bug localization

Όπως φαίνεται από τα δεδομένα του πίνακα τα αποτελέσματα είναι ιδιαίτερα ικανοποιητικά για την εφαρμογή Nautilus. Από τις επτά αναφορές αστοχίας που εξετάστηκαν, οι μέθοδοι των τεσσάρων κατατάχθηκαν στις υψηλότερες θέσεις των αντίστοιχων ταξινομημένων λιστών και συγκεκριμένα κάτω από το 1% των μεθόδων που ανακτήθηκαν σε κάθε λίστα. Έτσι για την αναφορά αστοχίας με id 333265, η μέθοδος, που αποτελεί το resolution της, κατατάχθηκε στην $10^{η}$ θέση μεταξύ 3247 άλλων μεθόδων που ανακτήθηκαν και αντιστοιχεί στο 0.308% των συνολικών μεθόδων της λίστας.

Ίδια περίπτωση είναι και η bug report 697183, για την επίλυση της οποίας χρειάστηκε να τροποποιηθούν δύο μέθοδοι. Το bug localization σύστημα που υλοποιήθηκε κατατάσσει τις συγκεκριμένες μεθόδους στις θέσεις 6 και 3 ανάμεσα στις 2695 άλλες, η μεγαλύτερη των οποίων αντιστοιχεί στο 0.223% των μεθόδων της ταξινομημένης λίστας.

Παρομοίως, για την αναφορά 698190, οι τέσσερις μέθοδοι που επηρέαζαν τη λειτουργία της εφαρμογής και δημιουργούσαν το σφάλμα που περιγράφεται, ανακτήθηκαν από το σύστημά μας και μάλιστα στις πρώτες θέσεις. Για την ακρίβεια η μέθοδος get_visible_columns(NautilusListView*) κατατάχθηκε στη χαμηλότερη θέση από τις υπόλοιπες τρεις μεθόδους. Συγκεκριμένα το βάρος που της αποδόθηκε την κατέταξε $8^{η}$ στην ταξινομημένη λίστα μεταξύ 3673 μεθόδων συνολικά, δηλαδή στο 0.218% των μεθόδων.

Τέλος, η μέθοδος της αναφοράς 702546 ανήκει στο 0.117% των 4284 μεθόδων που ανακτήθηκαν, αφού ήρθε $5^{η}$ στην ταξινόμηση της λίστας. Η επιτυχία των παραπάνω μεθόδων οφείλεται στην επιλογή του βάθους και των σχέσεων που προέκυψαν από την ανάλυση του κώδικα από το FETCH Tool.

Παρ' όλα αυτά όμως υπάρχουν και οι υπόλοιπες αναφορές αστοχιών, οι οποίες παρά το γεγονός ότι δεν έχουν το ίδιο καλά αποτελέσματα με τις προηγούμενες, η κατάταξη των μεθόδων στην εκάστοτε ταξινομημένη λίστα αντιστοιχεί σε ποσοστά κάτω του 5% των συνολικών αναφορών. Συνεπώς η μέθοδος που τροποποιήθηκε για την επίλυση του bug της αναφοράς με αναγνωριστικό 71480, βρίσκεται στην $73^{η}$ θέση μεταξύ 4267 μεθόδων που ανακτήθηκαν. Η θέση αυτή αντιστοιχεί στο 1.711%. Κοντά στο ποσοστό αυτό βρίσκονται και τα αποτελέσματα της 697890 αναφοράς αστοχίας, της οποίας η μέθοδος με την χαμηλότερη θέση στην ταξινομημένη λίστα αντιστοιχεί στο 2.06%. Αυτό σημαίνει ότι από τις 3673 μεθόδους που ανακτήθηκαν από το bug localization σύστημα, το resolution της αναφοράς βρίσκεται στο 2% αυτών. Η τελευταία αναφορά αστοχίας που παρουσιάζεται στον πίνακα της προηγούμενης σελίδας και συγκεκριμένα η μέθοδος που αποτελεί το resolution της αστοχίας, ανακτήθηκε και τοποθετήθηκε στην ταξινομημένη λίστα του bug localization συστήματος και μάλιστα στο 4.907% του συνολικού αριθμού των μεθόδων.

Συμπεραίνουμε λοιπόν ότι δεν υπάρχει στη συγκεκριμένη εφαρμογή λογισμικού κάποια αναφορά αστοχίας που να έχει αποτελέσματα, τα οποία ξεπερνάνε το 5% των συνολικών μεθόδων που ανακτώνται. Αυτό σημαίνει ότι η επιλογή του βάθους (depth = 1) για το nautilus φέρνει, τόσες μεθόδους όσες είναι απαραίτητες για τον εντοπισμό εκείνων που τροποποιήθηκαν. Πιθανή αλλαγή στην τιμή του βάθους μπορεί να αυξήσει τον αριθμό των ανακτηθέντων μεθόδων το οποίο με τη σειρά του είναι δυνατό να επιφέρει διαφοροποιήσεις στην κατάταξη των μεθόδων στις αντίστοιχες ταξινομημένες λίστες.

Παρακάτω ακολουθεί ο πίνακας που περιλαμβάνει τις ίδιες αναφορές με του προηγούμενο, αφού αποτελούν άλλωστε και τις αναφορές εισόδου, με τη μόνη διαφορά ότι αυτή τη φορά περιέχει το ποσοστό των precision και recall. Στην περίπτωση του nautilus, όπως και στις προηγούμενες εφαρμογές που αναλύθηκαν και εξετάστηκαν από το bug localization σύστημά μας, το ποσοστό του precision είναι αρκετά μικρό. Αυτό συμβαίνει επειδή οι μέθοδοι που ανακτώνται συνδυάζοντας το βάθος αναζήτησης στον γράφο των σχέσεων με μερικές από τις σχέσεις αυτές, έφεραν τις κατάλληλες μεθόδους στην ταξινομημένη λίστα και ταυτόχρονα ανακτήθηκαν και αρκετές άλλες που σύμφωνα με την απόδοση βάρους δεν θεωρήθηκαν απαραίτητες για την επίλυση του εκάστοτε bug. Το ποσοστό του precision είναι δυνατό να μειωθεί αν αντί για όλες τις μεθόδους που ανακτώνται, επιλεχθεί μόνο ένα ποσοστό αυτών. Παρ' όλα αυτά επειδή όλες οι μέθοδοι εντοπίζονται με επιτυχία και γι' αυτό το ποσοστό του recall είναι σταθερό στο 100%.

Input Bug Reports (IDs)	**Methods**	**Ranked position in list / number of methods in list**	**Precision (%)**	**Recall (%)**
333265	real_update_menus(NautilusView*)	10/3247	0.031	100
697183	apply_columns_settings(NautilusListView*, char**,char**)	6/2695	0.074	100
	create_and_set_up_tree_view(NautilusListView*)	3/2695		
697890	filtering_changed_callback(gpointer)	23/3673	0.081	100
	invalidate_one_count(gpointer,gpointer, gpointer)	88/3673		
	collect_all_directories(gpointer,gpointer, gpointer)	25/3673		
698190	column_chooser_use_default_callback (NautilusColumnChooser*,NautilusListView*)	1/3673	0.109	100
	get_column_order(NautilusListView*)	7/3673		
	get_visible_columns(NautilusListView*)	8/3673		
	nautilus_list_view_reset_to_defaults (NautilusView*)	3/3673		
702546	nautilus_window_slot_dispose(GObject*)	5/4284	0.023	100
71480	custom_icon_file_chooser_response_cb (GtkDialog*,gint,NautilusPropertiesWindow*)	73/4267	0.023	100
703349	append_directory_contents_fields (NautilusPropertiesWindow*,GtkGrid*)	143/2914	0.034	100

Πίνακας 6.38: Precision και Recall για τα αποτελέσματα bug

Περιορίζοντας λοιπόν το πλήθος των μεθόδων που ανακτώνται και ταξινομούνται στη λίστα εξόδου του bug localization συστήματος στο 3%, 5% και 7%, παίρνουμε τα παρακάτω αποτελέσματα, τα οποία θα σχολιαστούν στη συνέχεια:

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
333265	1.02	100	98
697183	2.469	100	81
697890	2.727	100	110
698190	3.63	100	110
702546	0.775	100	129
71480	0.781	100	128
703349	1.149	100	87

Πίνακας 6.39: Precision και Recall για το 3% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
333265	0.617	100	162
697183	1.481	100	135
697890	1.63	100	184
698190	2.173	100	184
702546	0.467	100	214
71480	0.469	100	213
703349	0.685	100	146

Πίνακας 6.40: Precision και Recall για το 5% των μεθόδων της ταξινομημένης λίστας

Bug Reports (IDs)	Precision (%)	Recall (%)	Number of Methods in ranked list
333265	0.44	100	227
697183	1.058	100	189
697890	1.167	100	257
698190	1.556	100	257
702546	0.333	100	300
71480	0.334	100	299
703349	0.49	100	204

Πίνακας 6.41: Precision και Recall για το 7% των μεθόδων της ταξινομημένης λίστας

Σε σχέση με τις άλλες εφαρμογές, στο nautilus, τα ποσοστά του precision είναι μεγαλύτερα, τόσο για το 100% των μεθόδων όσο και για τα ποσοστά που παρουσιάζονται στους πίνακες 6.39, 6.40 και 6.41. Αυτό συμβαίνει επειδή οι μέθοδοι που ανακτώνται από το σύστημά μας δεν είναι τόσες πολλές όσες στις προηγούμενες εφαρμογές λογισμικού που αναλύθηκαν.

Μειώνοντας τον πλήθος των μεθόδων της ταξινομημένης λίστας, επιτυγχάνουμε την αύξηση του ποσοστού του precision. Ειδικά για 3% οι τιμές είναι πολύ υψηλές σε σχέση με τις αρχικές. Αυτό συμβαίνει επειδή το σύνολο των μεθόδων έχει μικρύνει, αλλά οι μέθοδοι που αναζητούμε μέσα σε αυτό έχουν παραμείνει σταθερές. Επιπροσθέτως, οι περισσότερες από τις αναφορές αστοχιών που χρησιμοποιήθηκαν για την ανάλυση του nautilus, απαιτούσαν την τροποποίηση περισσότερων από μίας μεθόδων και αυτό παίζει σημαντικό ρόλο στον υπολογισμό του precision, αυξάνοντας την τιμή του.

Για τα ποσοστά 5% και 7% παρατηρούμε σε σχέση με το αρχικό (πίνακας 6.38) είναι και αυτά βελτιωμένα. Συγκριτικά με το 3% όμως έχουν υποστεί μείωση, που είναι δικαιολογημένη αφού ο συνολικός αριθμός των μεθόδων της εκάστοτε ταξινομημένης λίστας είναι μεγαλύτερος από ότι πριν.

Σχετικά με το recall αξίζει να παρατηρήσουμε ότι για το 100% του συνόλου των μεθόδων, η θέση εκείνων που αναζητούμε στη λίστα είναι αρκετά υψηλή σε σχέση με το πλήθος αυτών που ανακτήθηκαν. Το ίδιο, όπως ήταν αναμενόμενο, συμβαίνει και στο 3%, 5% και 7% και επομένως το ποσοστό του recall παραμένει 100% για όλες τις αναφορές αστοχιών εισόδου.

7

Επίλογος

Στο κεφάλαιο αυτό, που αποτελεί και το κλείσιμο της διπλωματικής εργασίας, ανακεφαλαιώνονται όσα έχουν παρουσιαστεί στις προηγούμενες ενότητες. Αρχικά θα γίνει μία περιληπτική αναφορά στο σύστημα που υλοποιήθηκε και έπειτα θα αναλυθούν τα συμπεράσματα στα οποία καταλήγουμε μετά το στάδιο της αξιολόγησης του συστήματος συνολικά για όλες τις εφαρμογές λογισμικού που χρησιμοποιήθηκαν. Η διπλωματική εργασία θα καταλήξει με προτάσεις για μελλοντική έρευνα που θα αναφέρονται σε ζητήματα και τομείς που θα μπορούσαν να διερευνηθούν περαιτέρω ώστε να εξελιχθεί το ήδη υπάρχον σύστημα σε μία περισσότερο βελτιωμένη και αποτελεσματική έκδοση.

7.1 Συμπεράσματα

Ανακεφαλαιώνοντας λοιπόν τα όσα αναφέρθηκαν στις προηγούμενες ενότητες θα καταλήξουμε στα συμπεράσματα που προκύπτουν από την εκπόνηση αυτής της εργασίας.

Αρχικά, πραγματοποιήθηκε μία εκτενής περιγραφή και ανάλυση της έννοιας *αναφορά αστοχίας*, καθώς επίσης και του Bugzilla που αποτελεί το repository από το οποίο ανακτήθηκαν όλα τα bug reports. Εν συνεχεία παρουσιάστηκε ο κύκλος ζωής (lifecycle) μίας αναφοράς σφάλματος στο Bugzilla όσο και το μεταμοντέλο της για το συγκεκριμένο repository συνοδευόμενο από μία λεπτομερή επεξήγηση όλων των σχέσεων και των οντοτήτων που εμφανίζονται.

Μετά την κατανόηση των αναφορών αστοχιών που είναι και το βασικότερο στοιχείο για τη λειτουργία του συστήματος που υλοποιήθηκε, ασχοληθήκαμε με το σχεδιασμό του αλγορίθμου του bug localization μηχανισμού. Συνοπτικά, το περιβάλλον που δημιουργήθηκε δέχεται ως είσοδο μία αναφορά σφάλματος και στη συνέχεια δίνει ως έξοδο μία ταξινομημένη λίστα με τις μεθόδους ή συναρτήσεις της υπό μελέτης εφαρμογής λογισμικού, ώστε ο προγραμματιστής να έχει μία εικόνα σχετικά με τις εκείνες που είναι πιθανότερο να χρειάζονται τροποποίηση για να επιλυθεί το bug.

Για την αξιολόγηση του συστήματος που υλοποιήθηκε χρησιμοποιήθηκαν συγκεκριμένες εκδόσεις έξι εφαρμογών λογισμικού. Καθεμία επιλέχθηκε με άση τον αριθμό των αναφορών αστοχιών με status “Resolved” και του resolution που είχαμε διαθέσιμο, καθώς οι συγκεκριμένες αναφορές αποτέλεσαν την είσοδο του συστήματος. Τα παραπάνω bug reports επιλέχθηκαν τυχαία με μόνη προϋπόθεση να είναι γνωστές οι «ελλαττωματικές» μέθοδοι, δηλαδή εκείνες που τροποποιήθηκαν προκειμένου να επιλυθεί το εκάστοτε bug και να ανήκουν στην ίδια έκδοση (version) του προϊόντος.

Στοχεύοντας στη μείωση του αριθμού των μεθόδων, με σκοπό να γίνει περισσότερο διαχειρίσιμη από κάποιον προγραμματιστή η ταξινομημένη λίστα, και προσπαθώντας να μειώσουμε το ποσοστό του precision χωρίς όμως να επηρεαστεί το αντίστοιχο του recall, επιλέχθηκε ένα συγκεκριμένο ποσοστό μεθόδων της τελικής λίστας. Συγκεκριμένα παρουσιάστηκαν τα αποτελέσματα για 3%, 5% και 7%.

Σε γενικές γραμμές παρατηρούμε ότι το σύστημα που υλοποιήθηκε καταφέρνει με επιτυχία να εντοπίσει μεγάλο ποσοστό των μεθόδων που χρειάστηκαν τροποποίηση προκειμένου να επιλυθεί κάποιο bug στον πηγαίο κώδικα. Παρ’ όλα αυτά, σημαντικό μειονέκτημά του είναι το πλήθος των μεθόδων που ανακτώνται και τις οποίες θα πρέπει να λάβει υπόψιν του ο προγραμματιστής για να επιλύσει το σφάλμα.

Πιο αναλυτικά, τα συμπεράσματα στα οποία καταλήξαμε για την τεχνική του bug localization που υλοποιήθηκε είναι τα εξής:

- Στο στάδιο προ-επεξεργασίας των αναφορών αστοχιών, όπως έχει ήδη αναφερθεί, επιλέγονται οι αναφορές αστοχιών εκείνες που περιέχουν σχόλια. Επομένως, είναι προφανές ότι αποκλείονται όλες οι υπόλοιπες. Αυτό σημαίνει ότι τόσο το *πλήθος των bug reports του repository*, όσο και το *πλήθος εκείνων στις οποίες περιλαμβάνονται* σχόλια επηρεάζουν σημαντικά το τελικό αποτέλεσμα. Όσο περισσότερες είναι οι αναφορές αστοχιών, τόσο αυξάνεται η πιθανότητα ύπαρξης αναφορών με σχόλια

γεγονός που αυξάνει με τη σειρά του την πιθανότητα να βρεθούν εννοιολογικές ομοιότητες σε περισσότερες αναφορές αστοχίας.

- Ένας ακόμη σημαντικός παράγοντας, από τον οποίο κρίνεται το αποτέλεσμα του συστήματος που υλοποιήθηκε είναι το *βάθος* που επιλέχθηκε για την προσπέλαση του γράφου των σχέσεων. Όπως έχει ήδη αναφερθεί σε προηγούμενες ενότητες, το FETCH Tool εντοπίζει και δημιουργεί σχέσεις που συνδέουν τις οντότητες του πηγαίου κώδικα μεταξύ τους. Με βάση αυτές λοιπόν, δημιουργείται ένας γράφος ο οποίος περιλαμβάνει όλες τις οντότητες του και τις σχέσεις που εντοπίστηκαν. Για τη δημιουργία του Εκτεταμένου Συνόλου Μεθόδων (Extended Problem Set) προσπελάσαμε το γράφο σε βάθος που καθορίστηκε από το σύστημα μας. Παρ' όλα αυτά το βάθος προσπέλασης είναι δυνατό να τροποποιηθεί από τον χρήστη. Είναι σημαντικό όμως να τονισθεί ότι όσο αυξάνουμε το βάθος (depth), τόσο περισσότερες μέθοδοι συγκεντρώνονται στο Extended PS. Όμως δεν είναι απαραίτητα χρήσιμο αυτό. Ορισμένες αναφορές αστοχιών περιέχουν αρκετές λέξεις κώδικα, γεγονός που δημιουργεί ένα αρκετά μεγάλο Problem Set. Στις περιπτώσεις αυτές το μεγάλο βάθος προσπέλασης αυξάνει πάρα πολύ το σύνολο των μεθόδων στην ταξινομημένη λίστα, με αποτέλεσμα σε ορισμένες απ' τις μεθόδους να αποδίδεται υψηλό βάρος εξαιτίας των κοινών σημείων που εμφανίζουν με τις λέξεις της αναφοράς εισόδου, καθώς και να μειώνεται το ποσοστό του precision.

- Ακόμα μία μεταβλητή του συστήματος που υλοποιήθηκε είναι ο *βαθμός ομοιότητας* της αναφοράς εισόδου με τις υπόλοιπες αναφορές που υπάρχουν στο Bugzilla repository. Ο βαθμός ομοιότητας είναι σημαντικός παράγοντας για τη δημιουργία των clusters στη μέθοδο LSI. Αυτό σημαίνει ότι διαφοροποίηση στην τιμή του, επηρεάζει το πλήθος των αναφορών σφαλμάτων που θα τοποθετηθούν στο ίδιο cluster. Συνέπεια του προηγούμενου είναι η αύξηση ή μείωση του πλήθους των SCE (Source Code Entities) στο αρχικό σύνολο που δημιουργείται αφού έχουν συγκεντρωθεί οι λέξεις κώδικα όλων των «συγγενικών» αναφορών αστοχιών. Επομένως, αν ο χρήστης θεωρεί απαραίτητο να αυξήσει τον αριθμό των λέξεων κώδικα που προκύπτουν, ένας τρόπος είναι να μειώσει το κατώφλι του βαθμού ομοιότητας.

- Η μεθοδολογία που ακολουθήθηκε για τον εντοπισμό των μεθόδων που χρειάζονται τροποποίηση προκειμένου να επιλυθεί η αστοχία, έχει ως στόχο αρχικά τη δημιουργία ενός συνόλου (Problem Set) με τις λέξεις του πηγαίου κώδικα και εν συνεχεία την επέκταση αυτού στο Extended Problem Set, το οποίο περιέχει περισσότερες SCE από το αρχικό. Όμως χρειάζεται ιδιαίτερη προσοχή ώστε το σύνολο να μην έχει τεράστια έκταση και για το λόγο αυτό καθορίζεται το βάθος προσπέλασης του γράφου αλλά και οι *σχέσεις που επιλέχθηκαν* να χρησιμοποιηθούν. Το FETCH Tool δημιούργησε όλες εκείνες τις σχέσεις μεταξύ των οντοτήτων που περιγράφηκαν στο τρίτο κεφάλαιο. Από εκείνες όμως επιλέχθηκαν συγκεκριμένες σχέσεις, οι οποίες έχουν αναφερθεί προηγουμένως, δίνοντας ιδιαίτερη έμφαση σε εκείνες που φέρνουν μεθόδους στο Extended Set. Η επιλογή αυτή έχει σημαντική επίδραση στο αποτέλεσμα, αφού οι

σχέσεις που χρησιμοποιούμε είναι και εκείνες που καθορίζουν τις SCE που εμφανίζονται στο τελικό σύνολο.

- Ένας ακόμη παράγοντας που επηρεάζει το αποτέλεσμα του συστήματος που δημιουργήθηκε είναι ο τρόπος με τον οποίο είναι γραμμένες οι αναφορές αστοχιών. Είναι προφανές ότι η προσέγγιση του εν λόγω συστήματος στηρίζεται στο γεγονός ότι στις αναφορές αστοχιών υπάρχουν λέξεις κώδικα. Όσο περισσότερες SCE υπάρχουν στην περιγραφή και στα σχόλια μίας αναφοράς αστοχίας, τόσο περισσότερες πιθανότητες έχει το σύστημα να συσχετίσει το bug της αναφοράς με τις μεθόδους που χρειάζονται τροποποίηση και να τις ταξινομήσει στις πρώτες θέσεις της τελικής λίστας.

- Τέλος, μειώνοντας το πλήθος των αναφορών που ταξινομούνται υπάρχει κίνδυνος να «χάσουμε» κάποιες μεθόδους που εντοπίστηκαν στο 100% του συνόλου, αποκλείοντάς τες από το αυτό στην προσπάθειά μας να ελαττώσουμε το μέγεθος της λίστας. Παρ' όλα αυτά το συγκεκριμένο σύστημα, κρίνοντας από τα αποτελέσματά του, φάνηκε να μην εμφανίζει πρόβλημα, αφού οι περισσότερες μέθοδοι κατατάσσονται σε υψηλές θέσεις της λίστας.

7.2 Προτάσεις για μελλοντική έρευνα

Κατά τη διάρκεια μελέτης, υλοποίησης και συγγραφής του συγκεκριμένου πονήματος, προέκυψαν κάποια σημεία τα οποία θα μπορούσαν να διερευνηθούν περεταίρω με σκοπό τη βελτίωση του bug localization συστήματος που δημιουργήθηκε.

Αρχικό, αλλά και βασικότερο σημείο το οποίο δυσχεραίνει την επεξεργασία των αναφορών αστοχιών είναι το γεγονός ότι τα σχόλιά τους είναι γραμμένα σε φυσική γλώσσα, πιο συγκεκριμένα σε Αγγλικά. Όπως έχει ήδη αναφερθεί σε προηγούμενα κεφάλαια, από τις αναφορές αστοχιών που συγκεντρώθηκαν, επιλέχθηκαν εκείνες που περιείχαν λέξεις κώδικα στα σχόλια τους. Έτσι, αφαιρέθηκαν οι λέξεις της φυσικής γλώσσας και όπως είναι προφανές το νόημα του κειμένου δεν διατηρήθηκε, αφού επικεντρωθήκαμε μόνο στις **S**ource **C**ode **E**ntities (SCE). Χρησιμοποιώντας όμως κάποιο από τα εργαλεία που έχουν ως στόχο την επεξεργασία της φυσικής γλώσσας (Natural Language Processing Tools) και παρέχουν μεθόδους όπως ο χωρισμός κειμένου σε λέξεις (tokenization), σε προτάσεις (sentence segmentation), η αναγνώριση του μέρους του λόγου που ανήκει κάθε όρος του κειμένου (part of speech tagging) και η δημιουργία του συντακτικού δέντρου της κάθε πρότασης, θα μπορούσαν να διατηρηθούν και άλλες λέξεις ή φράσεις που θα προσδίδουν ένα διαφορετικό νόημα και ταυτόχρονα θα συνεισφέρουν στον συσχετισμό των αναφορών αστοχιών μεταξύ τους. Έτσι, θα παρουσίαζε ιδιαίτερο ενδιαφέρον να προστεθεί και αυτή η λογική στο σχεδιασμό του bug localization συστήματος προτού εφαρμοστεί η τεχνική LSI στις αναφορές σφαλμάτων.

Ένα άλλο σημείο το οποίο είναι άξιο μελέτης και πειραματισμού είναι η τεχνική ομαδοποίησης που χρησιμοποιείται για τη δημιουργία των clusters από αναφορές αστοχιών που εμφανίζουν εννοιολογική ομοιότητα μεταξύ τους. Υπάρχουν πολλές τεχνικές και αλγόριθμοι, μερικές από τις οποίες αναλύθηκαν στο κεφάλαιο 3. Ανάλογα με τα δεδομένα που έχουμε στη διάθεσή μας, κάποιες εμφανίζουν πλεονέκτημα στα αποτελέσματα σε σχέση με άλλες. Συνεπώς, θα μπορούσε να πραγματοποιηθεί μία πειραματική διαδικασία, με την οποία θα επιλεγόταν η πιο κατάλληλη μέθοδος σημασιολογικής ομαδοποίησης εγγράφων.

Σχετικά με το παραπάνω, θα ήταν χρήσιμο να μελετηθεί επιπλέον και η χρήση Machine Learning τεχνικών για το συσχετισμό και την ομαδοποίηση των αναφορών αστοχιών με βάση το περιεχόμενό τους. Σύμφωνα με δημοσιεύσεις και πειράματα άλλων ερευνητών, υπάρχουν και αναπτύσσονται διαφορετικές τεχνικές που βασίζονται στο Machine Learning και επιτυγχάνουν με αρκετή ακρίβεια τη δημιουργία clusters ομαδοποιημένων bug reports. Τέτοιες τεχνικές σε συνδυασμό με τη χρήση NLP εργαλείων, θα ήταν δυνατό να βελτιώσουν σημαντικά τα αποτελέσματα που παρουσιάστηκαν στο προηγούμενο κεφάλαιο.

Μία ακόμη πρόταση για μελλοντική έρευνα, είναι να εφαρμοστεί κάποια από τις τεχνικές ομαδοποίησης σε αρχεία του πηγαίου κώδικα. Αυτό απαιτεί την αφαίρεση των keywords της εκάστοτε γλώσσας προγραμματισμού από τον κώδικα, με χρήση κάποιων stemming εργαλείων και έπειτα την ομαδοποίηση των αρχείων προκειμένου αυτή τη φορά η αναφορά αστοχίας να συσχετιστεί με κάποιο cluster που θα αποτελείται από μεθόδους πηγαίου κώδικα. Παρ' όλο που βασίζεται σε διαφορετικό κορμό εργασίας από την παρούσα, θα μπορούσε να χρησιμοποιηθεί συνδυαστικά με τις προηγούμενες μεθόδους προκειμένου να δημιουργηθεί ένα ακόμη αποδοτικότερο σύστημα.

Επιπλέον παρατηρήθηκε και σχολιάστηκε στα αποτελέσματα το φαινόμενο πολλές από τις μεθόδους που χρειάστηκαν τροποποίηση να ανήκουν στην ίδια κλάση ή ακόμη και τα ονόματά τους να παρουσιάσουν λεκτική ομοιότητα. Αυτό είναι ένα χαρακτηριστικό, το οποίο θα βοηθούσε πιθανά στη βελτίωση του εργαλείου, καθώς φαίνεται και λογικό. Συχνά τυχαίνει κατά την εφαρμογή μίας αλλαγής σε μία μέθοδο, να χρειάζονται αλλαγή και άλλες μέθοδοι της ίδιας κλάσης. Ένα bug localization σύστημα θα μπορούσε, για παράδειγμα, να λαμβάνει υπόψιν του τις μεθόδους που κατέχουν υψηλές θέσεις στην ταξινομημένη λίστα και με βάση αυτές να προσδίνει νέο βάρος σε όσες από τις υπόλοιπες παρουσιάζουν εννοιολογική ή ακόμη και λεκτική μόνο ομοιότητα με αυτές. Το αποτέλεσμα μίας τέτοιας προσέγγισης θα ήταν να «ανεβάσει» στην κατάταξη μεθόδους που πιθανά να βαθμολογήθηκαν λανθασμένα, επειδή δεν είχαν πολλά κοινά μονοπάτια στο γράφο των σχέσεων με τις SCE της αναφοράς εισόδου.

Βιβλιογραφία

[1] B. Clearly, C. Exton, J. Buckley and M. English. *An Empirical Analysis of Information Retrieval based Concept Location Techniques in Software Comprehension*. Empirical Softw. Engg., 14(1):93-130, 2009.

[2] V.Dallmeier and T. Zimmermann. *Extraction of Bug Localization Benchmarks from History*. In ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering, pages 433-436, New York, NY, USA, 2007.

[3] E. Enslen, E. Hill, L.Pollock and K. Vijay-Shanker. *Mining Source Code to Automatically Split Identifiers for Software Analysis*. In Proceedings of the 2006 6th IEEE International Working Conference on Mining Software Repositories, MSR '09, pages 71-80, Washington, DC, USA, 2009, IEEE Computer Society.

[4] D. B. H. Field and D. Lawrie. *An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers*. In Proceedings of IASTED International Conference on Software Engineering and Applications, 2006.

[5] S. K. Lukins, N. A. Kraft and Letha. H. Etzkorn. *Source code Retrieval for Bug Localization using Latent Dirichlet Allocation*. In 15th Working Conference on Reverse Engineering, 2008.

[6] A. Marcus, A. Sergeyev, V. Rajlich and J. I. Maletic. *An Information Retrieval Approach to Concept Location in Source Code*. In ICSE '03: Proceedings of the 25th International Conference on Reverse Engineering, WCRE 2004, pages 214-223. IEEE Computer Society, 2004.

[7] I. Ruthven and M. Lalmas. *A Survey on the Use of Relevance Feedback for Information Access Systems.* The Knowledge Engineering Review, 2003.

[8] R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval,* volume 463. ACM press New York, 1999.

[9] D. M. Blei and J. D. Lafferty. Topic models. *In Text Mining: Classification, Clustering and Applications, pages 71-94.* Chapman & Hall, London, UK, 2009.

[10] D. M. Blei, A. Y. Ng and J. Jordan. *Latent Dirichlet Allocation*. Journal of Machine Learning Research, 3:993-1022, 2003.

[11] E. Keogh, S. Lonardi and C. Ratanamahatana. *Towards parameter- free data mining*. In Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining, pages 206-215, 2003.

[12] S. Rao and A. Kak. *Retrieval from software libraries for bug localization: a comparative study of generic and composite text models*. In Proceedings of the 8th Working Conference on Mining Software Repositories, pages 43-52, 2011.

[13] C. D. Manning, P. Raghavan and H. Schutze. *Introduction to Information Retrieval*, volume 1. Cambridge University Press Cambridge, UK, 2008.

[14] G. Salton, A. Wong and C. S. Yang. *A vector space model for automatic indexing.* Communications of the ACM, 18(11):620, 1975.

[15] M. Girolami and A. Kaban. *On an equivalence between PLSI and LDA,* in: Proc. 22nd Annu. ACM SIGIR Int. Conf. on Research and Development in Information Retrieval, Toronto, Ontario, Canada, July 2003, pages 433-434.

[16] *Understand Source Code Analysis and Metrics*, http://www.scitools.com/products/understand/.

[17] M. Steyves and T. Griffiths. *Probabilistic topic models,* in T. Landauer, D. McNamara, S. Dennis, W. Kintsch (Eds.), Handbook of Latent Semantic Analysis, Lawrence Erlbaum Associates, 2007.

[18] S. W. Thomas, M. Nagappan, D. Blostein and A. E. Hassan. *The Impact of Classifier Configuration and Classifier Combination on Bug Localization*, in IEEE Transactions on Software Engineering, volume 39, issue 10, pages 1427-1443, 2013.

[19] D. Poshyvanyk and A. Marcus. *Combingin. Formal Concept Analysis with Information Retrieval for Concept Location in Source Code*, in IEEE 15th International Conference on Program Comprehension (ICPC '07), pages 37-48, 2007.

[20] Object Management Group, Inc. *Meta Object Facility (MOF) Core Specification Version 2.0*, January 2006, http://www.omg.org/spec/XMI/2.1.1/.

[21] Object Management Group, Inc. *MOF 2.0/XMI Mapping Version 2.1.1*, December 2007.

[22] Object Management Group, Inc. *OMG-Meta Object Facility Version 1.4*, April 2002.

[23] L. Hiew. *Assisted Detection of Duplicate Bug Reports*, University of British Columbia, 2003.

[25] D. Marjanovic. *Developing a Meta Model for Release History Systems*, University of Zurich, Department of Informatics, 2006.

[26] S. Deerwester, S. Dumais, G. Furnas, T. K. Landauer and R. Harshman. *Indexing by Latent Semantic Analysis*, Journal of the American Society for Information Science 41.4 (Sep 1, 1990): 391.

[27] M. W. Berry, S. Dumais and G. W. O'Brien. *Using Linear Algebra for Intelligent Information Retrieval*, Society for Industrial and Applied Mathematics, 1995.

[28] J. Geiß. *Latent Semantic Indexing and Information Retrieval - A quest with BosSE*, January 2016.

[29] *An Overview of the Architecture of EIA's CASE Data Interchange Format (CDIF),* http://wi.wu-wien.ac.at/rgf/9606mobi.html

[30] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol and V. Rajlich. *Feature Location Using Probabilistic Ranking of Methods Based on Excecution Scenarios and Information Retrieval*, in IEEE Transactions on Software Engineering, Volume 33, Issue 6, pages 420-432, 2007.

[31] *Cosine Similarity,* https://en.wikipedia.org/wiki/Cosine_similarity.

[32] K. Sruthi and B. Venkateshwar Reddy. *Document Clustering on Various Similarity Measures*, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 3, Issue 8, 2013.

[33] S. Chauhan, P. Arora, P. Bhadana. *Algorithm for Semantic Based Similarity Measure*, International Journal of Engineering Science Invention Volume 2 Issue 6, pp.75-78, 2013.

[34] S. Yeasmin, C. K. Roy and K. A. Schneider. *How should we read and analyze bug reports: an interactive visualization using extractive summaries and topic evolution*, in CASCON '15 Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering, pages 171-180, 2015.

[35] B. Du Bois, B. Van Rompaey, K. Meijfroidt, E. Suijs. *Supporting Reengineering Scenarios with FETCH: an Experience Report*, ECEASST vol. 8, 2007.

[36] Object Management Group, Inc. *UML Infrastructure Specification*, Version 2.4.1, http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/.

[37] *Singular Value Decomposition*, https://en.wikipedia.org/wiki/Singular_value_decomposition.

[38] C. B. Lang, N. Pucker. *Mathematische Methoden in der Physik*, Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[39] G. Strang, *Lineare Algebra*, Springer, Berlin, Heidelberg, 2003.

[40] *Jaccard Index*, https://en.wikipedia.org/wiki/Jaccard_index.

[41] S. Sachdeva and B. Kastore. *Document Clustering: Similarity Measures*, Indian Institute of Technology, Kanpur, April 30, 2014.

[42] *Bugzilla*, https://www.bugzilla.org/docs/2.16/html/whatis.html.