Ατομική Διπλωματική Εργασία

Δημιουργία συντακτικού αναλυτή κώδικα σε Java και συνένωση με εργαλείο τεμαχισμού προγράμματος για τη βελτιστοποίηση του πηγαίου κώδικα

Χριστόδουλος Μιχαήλ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιούλιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Δημιουργία συντακτικού αναλυτή κώδικα σε Java και συνένωση με εργαλείο τεμαχισμού προγράμματος για τη βελτιστοποίηση του πηγαίου κώδικα

Χριστόδουλος Μιχαήλ

Επιβλέπων Καθηγητής
Ανδρέας Ανδρέου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Ιούλιος 2010

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους γονείς μου για την στήριξη και συμπαράσταση τους όλα τα χρόνια φοίτησης μου στο Τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου. Ήταν πάντοτε δίπλα μου στην δύσκολη και επίπονη πορεία για την κατάκτηση αυτού του πτυχίου.

Θα ήθελα επίσης να ευχαριστήσω τον Δρ. Ανδρέα Ανδρέου και τον Δρ. Αναστάση Σοφοκλέους για την κατανόηση, καθοδήγηση και βοήθεια στις δυσκολίες που αντιμετώπισα για να φέρω εις πέρας αυτή την εργασία.

Περίληψη

Ο στόχος αυτής της διπλωματικής εργασίας είναι η δημιουργία ενός συντακτικού αναλυτή σε Java που να εντοπίζει κάποια χρονοβόρα κομμάτια (όπως είναι πχ. οι επαναληπτικοί βρόγχοι, οι αναδρομικές μεθόδοι και οι μεθόδοι γενικότερα) στον πηγαίο κώδικα ενός προγράμματος. Μέσα από την εμβέλεια αυτών των κομματιών μπορούν να παρθούν οι γραμμές που περιέχουν μεταβλητές. Στην συνέχεια, αυτές οι γραμμές μπορούν να χρησιμοποιηθούν (“συνενωθούν”) από ένα εργαλείο τεμαχισμού προγράμματος. Αυτό θα έχει σαν αποτέλεσμα τον τεμαχισμό του πηγαίου κώδικα σε πιο μικρά κομμάτια, που θα περικλείονται μέσα στην εμβέλεια κάποιου γενικότερου χρονοβόρου κομματιού. Έτσι, θα μπορούμε αργότερα να μελετήσουμε ένα μικρότερο κομμάτι κώδικα και να προτείνουμε κάποιες εισηγήσεις για βελτιστοποίηση του. Τέλος, αν μπορούμε να βελτιστοποιήσουμε τον συγκεκριμένο πηγαίο κώδικα, κάνουμε τις απαραίτητες αλλαγές και χρησιμοποιούμε ένα profiler για να δούμε αν έχουμε πετύχει τον στόχο μας, συγκρίνοντας τους χρόνους εκτέλεσης του προγράμματος (αυτόν που είχε αρχικά με αυτόν που είχε μετά την βελτιστοποίηση).

Σε γενικές γραμμές, θα αναφερθούμε στον τεμαχισμό προγράμματος και τι μπορεί να μας προσφέρει. Θα μιλήσουμε γενικά για βελτιστοποίηση προγράμματος και τα πλεονεκτήματα - μειονεκτήματα που τον συνοδεύουν. Θα κάνουμε εισηγήσεις για βελτιστοποίηση πηγαίου κώδικα. Θα γίνει παρουσίαση της όλης μεθοδολογίας, καθώς και πειραματικές μετρήσεις της επίδοσης διαφόρων μικρών προγραμμάτων. Τελειώνοντας, θα μιλήσουμε για τα συμπεράσματα μας, καθώς και για μελλοντική εργασία πάνω σε αυτού του είδους την μελέτη.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Στις μέρες μας η ραγδαία ανάπτυξη και τα επιτεύγματα της τεχνολογίας έχουν κάνει την ζωή όλων μας ευκολότερη και πιο άνετη. Αυτά όλα τα οφείλουμε κυρίως σε αρκετούς που πρότειναν και θεμελίωσαν αρχικά κάποιες ιδέες και πάνω σ' αυτές τις ιδέες ήρθαν αργότερα άλλοι να κτίσουν και να πραγματοποιήσουν επιτεύγματα μεγαλύτερης αξίας από αυτά που αρχικά προτάθηκαν. Έτσι, κατάφεραν να προσφέρουν αυτή την έξαρση στον τομέα της τεχνολογίας που υπάρχει σήμερα.

Η εξέλιξη των ηλεκτρονικών υπολογιστών (Η/Υ) άρχισε εδώ και αρκετές δεκαετίες. Αρχικά οι Η/Υ ήταν τεράστια μηχανήματα, δύσκολοι στην χρήση, ακριβοί και προοριζόμενοι μόνο για πολύ εξειδικευμένες εφαρμογές. Από τότε οι Η/Υ εξελίχθηκαν ραγδαία, παρουσιάσθηκαν νέα μοντέλα, με πολύ αυξημένες δυνατότητες και σε πιο προσιτό κόστος.

Αυτή η εξέλιξη οφείλεται στην ανάπτυξη τόσο λογισμικού (software) όσο και του κατάλληλου υλικού (hardware) που να υποστηρίζει το λογισμικό. Σήμερα η ανάπτυξη των επεξεργαστών και οι ταχύτητες επεξεργασίας που μπορούν να μας προσφέρουν,

βοηθούν εφαρμογές λογισμικού να επεξεργάζονται δεδομένα πολύ πιο γρήγορα και πιο αποδοτικά από ότι παλαιότερα.

Λόγω της ανάπτυξης στο τομέα των ταχυτήτων των επεξεργαστών και των προσιτών τιμών στις οποίες αυτοί πουλιούνται, οι οργανισμοί που αναπτύσσουν λογισμικά, συνήθως ξεχνούν την έννοια της βελτιστοποίησης προγραμμάτων. Όμως, παρόλα αυτά, ακόμη εξακολουθούν να υπάρχουν σήμερα εφαρμογές λογισμικού που χρειάζονται μέχρι και 4 - 5 ημέρες για να επεξεργαστούν δεδομένα που τους παρέχονται. Για το λόγο αυτό πρέπει να δοθεί έμφαση και στους διάφορους ελέγχους που μπορούν να γίνουν στην προσπάθεια βελτιστοποίησης αυτών των εφαρμογών. Δεν μπορούμε πάντοτε να κάνουμε υπομονή μέχρι να κυκλοφορήσουν νέα μοντέλα επεξεργαστών που να είναι ταχύτερα έτσι ώστε να έχουμε και το επιθυμητό αντίκτυπο στην εφαρμογή μας.

Οπόταν αν μπορούμε με διάφορους ελέγχους να εντοπίσουμε κάποια χρονοβόρα κομμάτια στον κώδικα της εφαρμογής μας, μπορούμε να προσπαθήσουμε να τα βελτιώσουμε ώστε να γίνουν πιο γρήγορα. Έτσι δεν θα χρειάζονται πλέον 4 – 5 ημέρες ή και παραπάνω για την επεξεργασία των δεδομένων μας, αλλά σαφώς λιγότερες. Έτσι, μπορούμε να παρατηρήσουμε ότι η βελτιστοποίηση ενός προγράμματος λογισμικού είναι σε πολλές περιπτώσεις αρκετά σημαντική αφού μπορεί να μας γλιτώσει πολύτιμο χρόνο και χρήμα. Μπορεί να μας γλιτώσει σε χρόνο, αφού μπορεί να βοηθήσει την εφαρμογή μας να επεξεργάζεται δεδομένα πολύ πιο γρήγορα. Από την άλλη, μπορεί να μας γλιτώσει απο περιττά έξοδα, αφού δεν θα χρειάζεται πλέον η αγορά πιο ακριβού εξοπλισμού ή ακόμα μιας άλλης εφαρμογής δαπανηρότερης.

1.2 Σκοπός

Σκοπός αυτής της διπλωματικής εργασίας είναι να προτείνει μια μεθοδολογία με στόχο την απομόνωση συγκεκριμένων κομματιών κώδικα που μπορεί να είναι χρονοβόρα. Έπειτα ο/η προγραμματιστής/προγραμματίστρια του συγκεκριμένου προγράμματος ή κάποιος/κάποια τρίτος/τρίτη με την κατάλληλη εμπειρία, να μπορεί να μελετήσει αυτά τα απομονωμένα κομμάτια κώδικα και να προσπαθήσει να βρει τρόπους βελτιστοποίησης τους, κάνοντας χρήση της εμπειρίας και γνώσης που διαθέτει.

Πιο συγκεκριμένα, ο σκοπός αυτής της διπλωματικής εργασίας είναι η δημιουργία ενός συντακτικού αναλυτή σε γλώσσα προγραμματισμού Java [7] που με την βοήθεια ενός εργαλείου κατάτμησης κώδικα λογισμικού (slicing tool) να μπορεί να δώσει ένα πιο μικρό πρόγραμμα στον/στην χρήστη που να περικλείει τα κομμάτια που θέλει να ελέγξει για τυχόν βελτιστοποιήσεις. Ο συντακτικός αναλυτής θα έχει σαν στόχο να εντοπίσει πρώτα κάποια σημεία στον κώδικα που ως επί το πλείστο μπορεί να είναι χρονοβόρα, με την βοήθεια ενός profiler [16] ή και χωρίς. Τέλος, αν ο/η χρήστης μπορεί να κάνει κάποιες βελτιστοποιήσεις, τότε ο profiler μπορεί να ξαναχρησιμοποιηθεί για να εξακριβωθεί αν ο χρόνος του προγράμματος έχει πράγματι βελτιωθεί.

1.3 Ανασκόπηση Κεφαλαίων

Στο κεφάλαιο 2 γίνεται μια αναφορά σε μεθόδους - εργαλεία για βελτιστοποίηση προγραμμάτων. Δίδεται ουσιαστικά ένα τεχνικό υπόβαθρο.

Στο κεφάλαιο 3 δίδεται το θεωρητικό υπόβαθρο γύρω από τεμαχισμό προγράμματος λογισμικού (program slicing) καθώς και βελτιστοποίησης προγραμμάτων. Δίδονται επίσης εισηγήσεις για βελτιστοποίηση κώδικα, κυρίως σε γλώσσα προγραμματισμού Java, αλλά κάποιες από τις εισηγήσεις ισχύουν και γενικότερα

Στο κεφάλαιο 4 δίδεται το σχήμα της μεθοδολογίας που ακολουθήθηκε. Γίνεται η παρουσίαση του profiler, του προγράμματος συντακτικής ανάλυσης και ενός εργαλείου για τεμαχισμό (κατάτμηση) κώδικα. Επίσης, δίδονται τα αποτελέσματα πειραματικών μετρήσεων της επίδοσης διαφόρων προγραμμάτων, που διεξάχθηκαν με τον profiler με σκοπό την σύγκριση της αντίστοιχης επίδοσης τους.

Στο κεφάλαιο 5 δίδεται μια ανακεφαλαίωση και συμπεράσματα από την όλη εργασία. Δίδονται, επίσης προτάσεις για μελλοντική εργασία.

Κεφάλαιο 2

Τεχνικό Υπόβαθρο

2.1 Γενικά

Ο καλύτερος τρόπος με τον οποίο μπορεί να γίνει βελτιστοποίηση ενός προγράμματος [9,17,18,19] είναι συνήθως με την αυτοματοποίηση που μπορεί να προσφέρει ο compiler (μεταγλωττιστής) [1] ενός προγράμματος ή με την επέμβαση ενός/μιας προγραμματιστή/προγραμματίστριας, κάνοντας κάποιες αλλαγές στον κώδικα που μπορούν να επιφέρουν κάποια βελτιστοποίηση. Όμως, τις περισσότερες φορές, η καλύτερη μέθοδος είναι με την χρήση κάποιου πιο γρήγορου αλγόριθμου [12] από αυτόν που έχει χρησιμοποιηθεί μέχρι στιγμής.

Η επιλογή ενός καλού αλγόριθμου είναι ένα δύσκολο έργο και ως επί το πλείστο δεν υπάρχει κάποια καθορισμένη διαδικασία για την επιλογή του καταλληλότερου αλγόριθμου. Τα πλεονεκτήματα ενός καλού αλγόριθμου είναι η σημαντική αλλαγή στην ταχύτητα επεξεργασίας των δεδομένων, καθώς και η σταθερότητα που προσφέρει στο πρόγραμμα. Μπορούμε να πούμε ότι είναι ο πιο σημαντικός παράγοντας στην βελτιστοποίηση της ταχύτητας ενός προγράμματος και μπορεί να πετύχει εκεί που οι εναλλακτικές προσεγγίσεις αποτυγχάνουν.

Ο compiler μιας γλώσσας προγραμματισμού μπορεί επίσης να κάνει αρκετές βελτιστοποιήσεις [1] σ' ένα πρόγραμμα και ίσως και καλύτερα από ότι μπορεί να τις κάνει ένας/μια προγραμματιστής/προγραμματίστρια. Κάποιες από τις πιο σημαντικές

βελτιστοποιήσεις είναι η μετακίνηση σταθερών εκφράσεων έξω από τα loops (βρόγχους επαναλήψεων), η αποθήκευση μεταβλητών σε registers (καταχωρητές), η δυνατότητα να κάνουν inline μεθόδους/συναρτήσεις (δηλαδή να γίνει ενσωμάτωση του κώδικα της μεθόδου/συνάρτησης εκεί όπου γίνεται η κλήση της μεθόδου/συνάρτησης), η απαλοιφή κώδικα που δεν χρησιμοποιείται κάπου στο πρόγραμμα και είναι αχρείαστος (dead code elimination) και επίσης η δυνατότητα loop unrolling [1,9,17,18,19,21] (ξετύλιγμα επαναληπτικών βρόγχων) ή πιο γενικά loop transformations (μετασχηματισμούς επαναληπτικών βρόγχων) [21].

Παρόλα αυτά ο/η προγραμματιστής/προγραμματίστρια μπορούν να βελτιστοποιήσουν ένα πρόγραμμα καλύτερα από ότι ένας compiler, διότι κατέχουν επιπρόσθετες γνώσεις για τον κώδικα του συγκεκριμένου προγράμματος που ο compiler δεν τις κατέχει. Όμως, ως συνήθως για τα περισσότερα προγράμματα (κώδικες) ένας καλός compiler γνωρίζει και έχει αρκετές πληροφορίες για να πάρει σωστές αποφάσεις όσο αφορά τις βελτιστοποιήσεις που μπορούν να πραγματοποιηθούν. Δεν τίθεται θέμα να προσπαθεί κάποιος να κάνει βελτιστοποίηση στο "χέρι" όταν μπορεί ο ίδιος ο compiler να πραγματοποιήσει την ίδια εργασία πιο γρήγορα και πιο αποτελεσματικά.

Όπως και να έχει όμως, η βελτιστοποίηση ενός ολόκληρου συστήματος αναλαμβάνεται από προγραμματιστές/προγραμματίστριες διότι είναι πολύ περίπλοκο να γίνει με την αυτοματοποίηση που μπορούν να προσφέρουν οι compilers. Έτσι, οι προγραμματιστές/προγραμματίστριες αλλάζουν κομμάτια κώδικα με σκοπό να βελτιώσουν την επίδοση του συστήματος. Επίσης κάποια κομμάτια κώδικα μπορούν να γραφτούν σε μια γλώσσα που είναι πιο κοντά στην γλώσσα μηχανής (πχ. assembly[1]), κάνοντας την εκτέλεση του συγκεκριμένου κομματιού κώδικα ταχύτερη. Όμως, αυτό είναι συνήθως κάτι πολύ δύσκολο και χρονοβόρο και τα άτομα που γνωρίζουν τέτοιου είδους γλώσσες χαμηλού επιπέδου, αποτελούν μειονότητα.

Αν όλες οι προσπάθειες για βελτιστοποίηση ενός προγράμματος αποτύχουν, τότε ο εναλλακτικός τρόπος για να γίνει κάτι πιο γρήγορο είναι να δούμε τα πράγματα απ' τη σκοπιά του υλικού (hardware) και της αρχιτεκτονικής του ηλεκτρονικού υπολογιστή πάνω στον οποίο βρίσκεται το πρόγραμμα το οποίο θέλουμε να βελτιώσουμε.

Δυο από τα πιο σημαντικά εργαλεία που προσφέρουν τεράστια βοήθεια σ' αυτούς που επιθυμούν να ασχοληθούν με την βελτιστοποίηση ενός προγράμματος, είναι τα εργαλεία για debugging (αποσφαλμάτωση κώδικα) και profiling.

2.2 Debugger (Αποσφαλματωτής)

Ένας αποσφαλματωτής είναι ένα εργαλείο/πρόγραμμα Η/Υ που χρησιμοποιείται για να ελέγξει και να πραγματοποιήσει αποσφαλμάτωση προγραμμάτων.

Τυπικά, ένας debugger προσφέρει την δυνατότητα εκτέλεσης ενός προγράμματος βήμα προς βήμα και της παρατήρησης των τιμών που παίρνουν οι μεταβλητές του προγράμματος κατά την βήμα προς βήμα εκτέλεσης τους. Μερικοί debuggers έχουν τη δυνατότητα να τροποποιήσουν την κατάσταση ενός προγράμματος ενώ εκτελείται, παρά απλά να την παρατηρήσουν. Μπορούν επίσης, να συνεχίσουν την εκτέλεση σε μια διαφορετική θέση στο πρόγραμμα για να παρακαμφθούν κάποιες εντολές που γνωρίζουμε εκ των προτέρων το αποτέλεσμα της εκτέλεσης τους και δεν μας ενδιαφέρει να τις παρακολουθήσουμε.

Βασικό εργαλείο σε ένα debugger είναι το breakpoint, που είναι ένα σημείο στον κώδικα στο οποίο επιθυμούμε να σταματήσει η εκτέλεση του προγράμματος. Επίσης, κάποιοι debuggers έχουν και τα λεγόμενα *watches* (παρακολουθητές), τα οποία μπορούν ανά πάσα στιγμή να παρακολουθούν τις τιμές κάποιας μεταβλητής, η οποία μεταβάλλεται κατά την διάρκεια εκτέλεσης του προγράμματος.

Έτσι, το γεγονός ότι προσφέρεται η δυνατότητα σε κάποιο/κάποια που θέλει να βελτιστοποιήσει ένα πρόγραμμα, να το εκτελέσει βήμα προς βήμα και να παρακολουθήσει την πορεία εκτέλεσής του, μπορεί να του προσφέρει αρκετή βοήθεια. Με αυτό τον τρόπο μπορεί να παρατηρήσει καλύτερα κάτι στον κώδικα του προγράμματος που να μπορεί να γραφτεί με πιο έξυπνο και αποδοτικό τρόπο και να πετύχει στη βελτίωση της ταχύτητάς του.

2.3 Profiler (Αναλυτής Επίδοσης)

Το profiling [9,15,17,18,19] ενός προγράμματος (μια μορφή δυναμικής ανάλυσης ενός προγράμματος) είναι η εξέταση της συμπεριφοράς ενός προγράμματος, που χρησιμοποιεί πληροφορίες από την εκτέλεση του προγράμματος. Ο σκοπός αυτής της ανάλυσης είναι να καθοριστούν ποια τμήματα ενός προγράμματος χρειάζονται να βελτιστοποιηθούν και σαν αποτέλεσμα να αυξηθεί η γενική ταχύτητά του, ή να μειωθεί η απαίτηση σε μνήμη που χρειάζεται, ή μερικές φορές και τα δύο.

Με την χρήση ενός profiler (αναλυτή επίδοσης) μπορούν να εντοπισθούν τα τμήματα σ' ένα πρόγραμμα τα οποία είναι χρονοβόρα. Οι προγραμματιστές/προγραμματίστριες ενός προγράμματος συνήθως έχουν την διαίσθηση ότι γνωρίζουν ποια είναι αυτά τα χρονοβόρα τμήματα, όμως πολλές φορές η διαίσθηση αυτή, τους εξαπατά.

Όταν εντοπισθεί κάποιο τμήμα το οποίο δείχνει να είναι χρονοβόρο, τότε το πρώτο πράγμα που μπορούμε να σκεφτούμε είναι αν το συγκεκριμένο τμήμα κώδικα μπορεί να γραφτεί με ένα καλύτερο αλγόριθμο. Στις περισσότερες περιπτώσεις, για ένα είδος προβλήματος, μπορεί να υπάρχει καλύτερος αλγόριθμος από αυτό που έχει ήδη χρησιμοποιηθεί. Για παράδειγμα, αν θέλουμε να ταξινομήσουμε μια τεράστια λίστα με αριθμούς, θα χρησιμοποιούσαμε ένα αλγόριθμο όπως το *mergesort* [12] ή το *quicksort* [12] αντί του *bubblesort* [12], διότι είναι σαφώς πιο γρήγοροι.

Παρόλα αυτά όμως, στις πλείστες των περιπτώσεων, η λύση δεν δίδεται με την απλή αντικατάσταση ενός αλγόριθμου με κάποιο άλλο. Αυτό λόγω του ότι δεν υπάρχουν αλγόριθμοι που να αποτελούν πανάκεια για όλες τις περιπτώσεις προβλημάτων. Γι' αυτό όμως υπάρχουν διάφορες εισηγήσεις για το πως ένα πρόγραμμα μπορεί να βελτιστοποιηθεί. Αυτές θα εξεταστούν στο επόμενο κεφάλαιο (αναλόγως βέβαια και της γλώσσας προγραμματισμού που μας ενδιαφέρει). Επομένως, μόλις βρεθεί κάποιος τρόπος με τον οποίο θεωρητικά ένα πρόγραμμα μπορεί να γίνει γρηγορότερο, τότε μπορούμε να εκτελέσουμε τον profiler στο διαφοροποιημένο πρόγραμμα (αφού γίνουν οι βελτιστοποιήσεις) και να δούμε αν πετύχαμε καλύτερη επίδοση.

Ένα παράδειγμα profiler είναι αυτό του Netbeans IDE 6.5.1 [15] που έχει χρησιμοποιηθεί και για τις ανάγκες αυτής της διπλωματικής εργασίας. Παράδειγμα της λειτουργίας του και πειραματικές διαδικασίες/μετρήσεις που έχουν γίνει με την χρήση του θα δοθούν στο κεφάλαιο 4.

Κεφάλαιο 3

Θεωρητικό Υπόβαθρο

3.1 Εισαγωγή

Σ' αυτό το κεφάλαιο θα μιλήσουμε για το θεωρητικό υπόβαθρο που χρειάζεται κανείς για να κατανοήσει περαιτέρω κάποια θέματα που χρησιμοποιεί η μεθοδολογία που ακολουθήθηκε για την εκπόνηση αυτής της διπλωματικής εργασίας. Πιο συγκεκριμένα, θα γίνει αναφορά στο τι είναι τεμαχισμός προγράμματος και ποια η έννοια της βελτιστοποίησης προγράμματος. Επιπλέον, θα γίνουν εισηγήσεις για βελτιστοποίηση κώδικα που ισχύουν γενικότερα για γλώσσες προγραμματισμού με παρόμοια σύνταξη όπως αυτή της Java, όμως θα συγκεντρωθούμε και σε αποκλειστικές εισηγήσεις που αφορούν την τελευταία.

3.2 Τεμαχισμός Προγράμματος (Program Slicing)

Τεμαχισμός προγράμματος/κώδικα (Program Slicing) [2,5,8] είναι μια τεχνική διάσπασης που πραγματοποιείται εξάγοντας δηλώσεις-εντολές ενός προγράμματος (program statements) που σχετίζονται με κάποιο υπολογισμό. Άτυπα, ένα τεμάχιο κώδικα (slice) κάνει την εξής ερώτηση : "*Τι program statements πιθανό να επηρεάσουν τον υπολογισμό της μεταβλητής V στο statement s;*". Αυτή η ερώτηση ουσιαστικά αποτελεί αυτό που ονομάζουμε κριτήριο τεμαχισμού κώδικα (slicing criterion). Τυπικά, ένα slicing criterion αποτελείται από ένα ζεύγος (αριθμός γραμμής, μεταβλητή). Έτσι, το program slicing είναι μια τεχνική εξαγωγής κομματιών ενός προγράμματος που επηρεάζουν ένα σύνολο από μεταβλητές που ενδιαφέρουν τον χρήστη. Με την συγκέντρωση του ενδιαφέροντος στον υπολογισμό μόνο συγκεκριμένων μεταβλητών, η διαδικασία του slicing μπορεί να χρησιμοποιηθεί στην απαλοιφή κομματιών κώδικα που δεν επηρεάζουν αυτές τις μεταβλητές και σαν αποτέλεσμα να μειωθεί η έκταση του συγκεκριμένου προγράμματος. Το μειωμένο κομμάτι του προγράμματος είναι αυτό που ονομάζεται slice.

Το slicing έχει πολλές εφαρμογές διότι επιτρέπει την απλοποίηση ενός προγράμματος, δίνοντας έμφαση σ' ένα υποτμήμα του αρχικού προγράμματος, αφού στον κώδικα μιας εφαρμογής απομένουν τα κομμάτια εκείνα που θέλει να απομονώσει ο χρήστης. Μεταξύ άλλων, το program slicing μπορεί να βοηθήσει στο βελτιστοποίηση προγράμματος (program optimization), αποσφαλμάτωση προγράμματος (program debugging), reverse engineering, συντήρηση λογισμικού (software maintenance), στον έλεγχο λογισμικού (testing), βοήθεια στη δημιουργία παράλληλων προγραμμάτων (parallelization), ενοποίηση λογισμικών προγραμμάτων (software integration), μετρικά λογισμικού (software metrics) και σε άλλους τομείς.

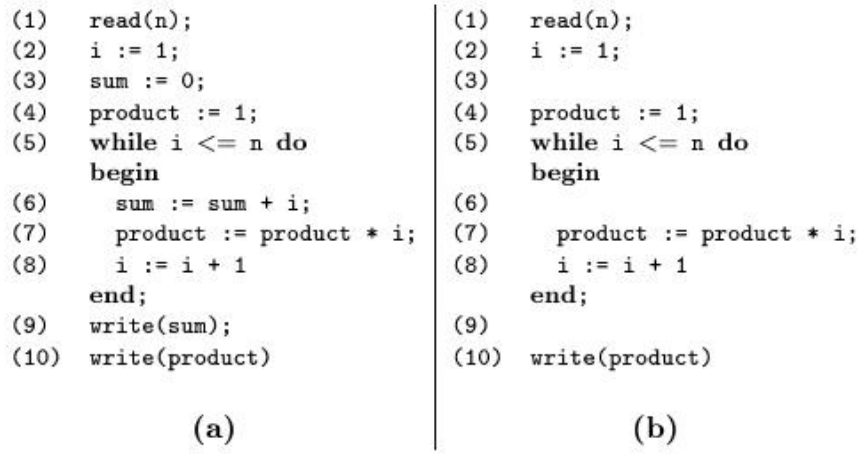
Τα program slices όπως αρχικά παρουσιάστηκαν (από τον Mark Weiser [5]), ονομάστηκαν executable backward static slices (εκτελέσιμα στατικά οπισθοδρομικά slice). Executable διότι το slice πρέπει να είναι ένα executable πρόγραμμα. Backward εξαιτίας της κατεύθυνσης των ακμών (direction edges) που διασχίζονται όταν το slice υπολογίζεται χρησιμοποιώντας ένα dependence graph (γράφο εξαρτήσεων). Static διότι υπολογίζονται χωρίς να λαμβάνουν υπόψη τους την είσοδο (input) του προγράμματος.

Γενικά υπάρχουν δύο είδη program slices, το backward slice και το forward (προς τα εμπρός) slice. Το backward slice αποτελείται από τις γραμμές εκείνες του κώδικα οι οποίες επηρεάζουν το κριτήριο που έχει επιλεχθεί, ενώ το forward slice αποτελείται από τις γραμμές του κώδικα που επηρεάζονται από το κριτήριο που έχει επιλεχθεί.

Τα slices μπορούν να δημιουργηθούν είτε static (στατικά), είτε dynamic (δυναμικά). Στο static slicing [2,4,5,6,8,11], η είσοδος (input) ενός προγράμματος είναι άγνωστη εκ το προτέρων και γι' αυτό το slice πρέπει να διατηρήσει πληροφορίες για όλες τις δυνατές εισόδους. Από την άλλη, στο dynamic slicing [2,3,4,5,6,8,11], η είσοδος ενός προγράμματος είναι γνωστή εκ το προτέρων και έτσι το slice χρειάζεται να διατηρήσει πληροφορίες μόνο για την συγκεκριμένη είσοδο. Το dynamic slice είναι μικρότερο από το static slice, όμως περιορίζει την εφαρμογή του σε μια συγκεκριμένη είσοδο για ένα πρόγραμμα. Είναι όμως ιδιαίτερα χρήσιμο, ειδικά σε debugging κώδικα όπου μπορεί να απομονώσει τον κώδικα για μια συγκεκριμένη είσοδο.

3.2.1 Στατική Κατάτμηση Κώδικα (Static Slicing)

Το static slice ενός προγράμματος σε σχέση με μια μεταβλητή *var*, σ' ένα κόμβο *n*, αποτελείται από όλους τους κόμβους/εντολές που η εκτέλεση τους μπορεί να επηρεάσει την τιμή της *var* στο *n*. Διαφορετικά μπορεί να ειπωθεί πως το static slice ενός προγράμματος αποτελείται από όλες τις εντολές/δηλώσεις (statements) ενός προγράμματος *P* που πιθανών να επηρεάσουν την τιμή της μεταβλητής *var* σε κάποιο σημείο *p* του προγράμματος.



Εικόνα 3.1: (a) Ένα παράδειγμα προγράμματος. (b) Ένα static slice του προγράμματος με κριτήριο (10,product) [8]

Η εικόνα 3.1 (a) [8] παρουσιάζει ένα παράδειγμα προγράμματος, όπου ζητείται από τον χρήστη ένας αριθμός n και υπολογίζεται το sum (άθροισμα) και γινόμενο (product) των πρώτων n θετικών αριθμών. Το slice ενός προγράμματος *P* δημιουργείται με βάση ένα κριτήριο, criterion = (line,var), όπου line είναι ο αριθμός μιας συγκεκριμένης γραμμής κώδικα (statement) και var ένα σύνολο μεταβλητών που περιλαμβάνονται μέσα στο πρόγραμμα *P*. Η εικόνα 3.1 (b) [8] παρουσιάζει το slice του συγκεκριμένου προγράμματος με κριτήριο (10, product). Στην εικόνα 3.1 (b) [8], παρατηρούμε πως έχουν απαλειφθεί όλες οι γραμμές οι οποίες δεν επηρεάζουν το τελικό αποτέλεσμα της μεταβλητής product. Έτσι, είναι εύκολο να παρατηρήσουμε πως το slice πρόγραμμα που παράχθηκε είναι μικρότερο σε έκταση από το αρχικό πρόγραμμα και αυτό μπορεί να διευκολύνει την περαιτέρω επεξεργασία του συγκεκριμένου κομματιού κώδικα.

3.2.2 Δυναμική Κατάτμηση Κώδικα (Dynamic Slicing)

Το static slice περιλαμβάνει όλες τις πιθανές εκτελέσεις ενός προγράμματος. Δηλαδή, τα static slices διατηρούν τον υπολογισμό της μεταβλητής *var* στο σημείο *p* για όλες τις εισόδους (inputs) ενός προγράμματος. Όμως, σε πολλές περιπτώσεις είναι πιο χρήσιμο ένα slice που διατηρεί πληροφορίες για την συμπεριφορά ενός προγράμματος για μια συγκεκριμένη είσοδο, παρά για όλες τις εισόδους. Αυτού του είδους το slice είναι το dynamic (δυναμικό).

Αν μας δοθεί ένα ιστορικό εκτέλεσης (execution history) *hist* ενός προγράμματος *P* για ένα test-case (περίπτωση-έλεγχου) *test* και για μια μεταβλητή *var*, τότε το dynamic slice του *P* σε σχέση με το *hist* και την *var,* είναι το σύνολο όλων των δηλώσεων/εντολών στο *hist* που η εκτέλεση τους είχε κάποια επιρροή στην τιμή της *var* στο τέλος του προγράμματος.

Ένα dynamic slice για ένα πρόγραμμα *P* αποτελείται από την πλειάδα (*P, I, line, Var*) όπου *P*, *I*, *line* και *Var*, συμβολίζουν το πρόγραμμα (program), την είσοδο (input) του προγράμματος για την οποία θα προσδιοριστεί το dynamic slice, την γραμμή και την μεταβλητή αντιστοίχως. Όλες οι δηλώσεις του προγράμματος που δεν τυγχάνουν προσπέλασης κατά τη διάρκεια της εκτέλεσης της συγκεκριμένης εισόδου *I* και εκείνες που δεν επηρεάζουν την τιμή των μεταβλητών στο *Var*, αποκλείονται από το slice του *P*.

```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     i := i + 1
      end;
(8)   write(x)
```

(a)

```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)             ;
(7)     i := i + 1
      end;
(8)   write(x)
```

(b)

Εικόνα 3.2: (a) Ένα άλλο παράδειγμα προγράμματος. (b) Dynamic slice του προγράμματος με κριτήριο (n = 2, 8^1, x) [8]

Η εικόνα 3.2 (b) [8] δείχνει ένα παράδειγμα dynamic slice ενός προγράμματος με κριτήριο (n = 2, 8^1, x), όπου 8^1 δηλώνει την πρώτη εμφάνιση της εντολής (statement) 8 στο ιστορικό εκτέλεσης *hist* του προγράμματος. Για είσοδο n = 2, το loop θα εκτελεστεί δύο φορές και οι αναθέσεις x := 17, x := 18, θα εκτελεστούν από μια φορά. Στο παράδειγμα, την πρώτη φορά που θα εκτελεστεί ο βρόγχος (loop), θα γίνει η ανάθεση x := 18. Την δεύτερη όμως φορά, θα γίνει η ανάθεση x := 17. Οπόταν το x := 18 δεν

επηρεάζει την τελική τιμή του x. Γι΄ αυτό στο dynamic slice αυτού του προγράμματος δεν συμπεριλαμβάνεται η δήλωση x := 18 (εικόνα 3.2 (b) [8]). Στο ίδιο παράδειγμα, το static slice με κριτήριο (8, x) θα περιλάμβανε όλο το αρχικό πρόγραμμα.

3.3 Βελτιστοποίηση Προγράμματος

Στην πληροφορική η βελτιστοποίηση προγράμματος [9,17,18,19] είναι η διαδικασία τροποποίησης ενός συστήματος λογισμικού έτσι ώστε να επεξεργάζεται κάποιες από τις συγκεκριμένες εργασίες πιο αποτελεσματικά ή ακόμη και να χρησιμοποιεί λιγότερους πόρους. Γενικότερα, ένα πρόγραμμα μπορεί να βελτιστοποιηθεί έτσι ώστε να εκτελεί κάποιες εργασίες γρηγορότερα και σαν αποτέλεσμα πολλές φορές να γίνεται λιγότερη αποθήκευση μνήμης από ότι προηγουμένως.

Ένας άλλος όρος είναι αυτός της βελτιστοποίησης κώδικα. Βελτιστοποίηση κώδικα, είναι η απαλοιφή αχρείαστων εντολών ή η αντικατάσταση εντολών από συνδυασμό άλλων εντολών που μπορούν να εκτελέσουν μια εργασία πολύ πιο γρήγορα.

Η βελτιστοποίηση ενός προγράμματος τις πλείστες των περιπτώσεων έχει σκοπό να βρει κάποια χρονοβόρα σημεία σ' ένα κώδικα. Στην τεχνολογία λογισμικού υπάρχει ο όρος του 90/10 law (νόμος 90/10). Αυτός ο όρος αναφέρεται στο γεγονός ότι το 90% της εκτέλεσης ενός προγράμματος συνήθως ξοδεύεται στην εκτέλεση μόνο 10% του κώδικα.

Είναι γνωστό από την διεθνή βιβλιογραφία και από μαθήματα δομών δεδομένων και αλγορίθμων [12], πως τα χρονοβόρα κομμάτια σ' ένα κώδικα είναι αυτά που περιέχουν loops (επαναληπτικούς βρόγχους) και αναδρομικές μεθόδους/συναρτήσεις. Επίσης, πολυσύνθετες αριθμητικές πράξεις και ιδιαίτερα ο τελεστής modulo (υπόλοιπο διαίρεσης) κρύβουν και αυτά κάποιο κόστος στην επίδοση. Αυτά είναι από τα πιο φανερά σημεία σε ένα κώδικα που θα μπορούσαν να "κατηγορηθούν", όχι πάντοτε βέβαια, ως χρονοβόρα. Σίγουρα, υπάρχουν και αρκετά άλλα σημεία που μετατρέπουν ένα κομμάτι κώδικα σε χρονοβόρο. Αυτά συνήθως έχουν να κάνουν με το τρόπο που είναι γραμμένος ένας κώδικας και κατά πόσο ο/η προγραμματιστής/προγραμματίστρια

εκμεταλλεύεται σωστά αυτά που του προσφέρει η γλώσσα προγραμματισμού που χρησιμοποιεί.

3.3.1 Επίπεδα Βελτιστοποίησης

Η βελτιστοποίηση ενός προγράμματος μπορεί να γίνει σε διάφορα επίπεδα.

- Το πιο ψηλό επίπεδο είναι αυτό όπου μπορούμε να εκμεταλλευτούμε και να βελτιστοποιήσουμε τον σχεδιασμό της εφαρμογής (προγράμματος), κάνοντας χρήση των διαθέσιμων πόρων. Οι διαθέσιμοι αυτοί πόροι, συνήθως είναι η καλή επιλογή αποδοτικών αλγορίθμων που σαν αποτέλεσμα θα μας δώσουν καλύτερη ποιότητα κώδικα. Η επιλογή του κατάλληλου αλγορίθμου έχει επιπτώσεις στην απόδοση περισσότερο από οποιοδήποτε άλλο στοιχείο στον σχεδιασμό της εφαρμογής. Επομένως, είναι το πρώτο πράγμα που πρέπει να αποφασιστεί κατά το σχεδιασμό.

- Γράφοντας καλή ποιότητα κώδικα και αποφεύγοντας όσο γίνεται διαδικασίες και εντολές που ξέρουμε το κόστος τους εκ των προτέρων, χρησιμοποιώντας κάποιες άλλες πιο γρήγορες και αποδοτικές, είναι και αυτό ένα επίπεδο βελτιστοποίησης.

- Όπως αναφέραμε και στο προηγούμενο κεφάλαιο η χρήση ενός compiler μπορεί να προσφέρει κάποιες βελτιστοποιήσεις αυτόματα για μας. Επιπρόσθετα, υπάρχουν compilers που μπορούν να δεχθούν παραμέτρους έτσι ώστε να γίνεται σε διαφορετικά επίπεδα λεπτομέρειας η χρήση τους [1,9]. Επομένως, υπάρχει και αυτό το επίπεδο βελτιστοποίησης.

- Το πιο χαμηλό επίπεδο βελτιστοποίησης, είναι ο κώδικας που χρησιμοποιεί μια συμβολική γλώσσα (assembly[1]). Η assembly μπορεί να μας δώσει πιο αποδοτικό κώδικα αφού ο/η προγραμματιστής/προγραμματίστρια μπορεί να εκμεταλλευτεί το πλήρες ρεπερτόριο των εντολών της μηχανής. Γι' αυτό άλλωστε τα περισσότερα παραδοσιακά λειτουργικά συστήματα έχουν γραφτεί χρησιμοποιώντας κώδικα assembly.

3.3.2 Βελτιστοποίηση με την χρήση αλγορίθμων

Όπως έχει γίνει αναφορά πιο πάνω και στο προηγούμενο κεφάλαιο, ο καλύτερος τρόπος βελτιστοποίησης είναι τις περισσότερες φορές να χρησιμοποιηθεί ένας αλγόριθμος που είναι πιο γρήγορος σε χρόνο εκτέλεσης έναντι κάποιου άλλου. Για παράδειγμα, εάν χρειάζεται ένας αλγόριθμος για ταξινόμηση μιας λίστας στοιχείων, ο χειρότερος αλγόριθμος που μπορεί να χρησιμοποιηθεί έχει χρόνο πολυπλοκότητας $O(n^2)$ [12] ενώ ο καλύτερος έχει χρόνο πολυπλοκότητας *O(nlogn)*. Επομένως, ο αλγόριθμος που έχει τον χειρότερο χρόνο εκτέλεσης μπορεί να αντικατασταθεί από ένα άλλο που είναι πολύ καλύτερος και σαν αποτέλεσμα να κάνουμε πιο γρήγορο το πρόγραμμα ή την εφαρμογή μας.

Αξίζει να σημειωθεί ότι εκτός από τους *σειριακούς* αλγόριθμους, υπάρχουν και οι *παράλληλοι* αλγόριθμοι [22]. Με ένα παράλληλο αλγόριθμο μπορούμε μερικές φορές να κάνουμε την ίδια εργασία που εκτελεί και ένας σειριακός αλγόριθμος, όμως πετυχαίνοντας καλύτερο χρόνο εκτέλεσης (πολυπλοκότητας). Για παράδειγμα, αν θέλουμε να βρούμε το άθροισμα Ν αριθμών, αυτό μπορεί να γίνει σε χρόνο *O(n)* με τον πιο γρήγορο σειριακό αλγόριθμο, ενώ με ένα παράλληλο αλγόριθμο μπορεί να γίνει σε χρόνο *O(logn)*.

3.3.3 Πότε απαιτείται η βελτιστοποίηση

Ο Donald Knuth είχε κάποτε κάνει την ακόλουθη δήλωση :
"Premature optimization is the root of all evil" [17, 19]
Με λίγα λόγια, η πρόωρη βελτιστοποίηση μπορεί να επιφέρει κακό παρά το επιθυμητό αποτέλεσμα. Αυτή η φράση χρησιμοποιείται για να περιγράψει μια κατάσταση όπου ένας/μια προγραμματιστής/προγραμματίστρια αφήνει τις έννοιες του από πλευράς απόδοσης του προγράμματος να επηρεάσουν το γράψιμο του κώδικά του. Δηλαδή, αντί να σκέφτεται το πως να φέρει εις πέρας το έργο που έχει να επιτελέσει, σκέφτεται πως μπορεί να το κάνει βέλτιστο από την αρχή, προτού να το γράψει. Αυτό σαν αποτέλεσμα μπορεί να οδηγήσει σε ένα κώδικα που δεν είναι τόσο καθαρός και ευανάγνωστος. Επιπλέον, μπορεί να συντάξει ένα κώδικα που είναι ανακριβής λόγω του ότι αποσπάται

η προσοχή του/της από το πως να κάνει βέλτιστο τον κώδικα και ξεχνώντας το πρόβλημα που πρέπει να επιλύσει ο συγκεκριμένος κώδικας. Έτσι, ίσως είναι καλύτερα αρχικά να επιδιώκουμε το επιθυμητό αποτέλεσμα και έπειτα να μας απασχολεί η βελτιστοποίηση του ("first make it work, then make it fast").

Μια εναλλακτική προσέγγιση είναι πρώτα να σχεδιάζουμε την λύση ενός προβλήματος. Αν η σχεδίαση του προβλήματος μας είναι καθαρή και ευανάγνωστη, τότε όταν γράψουμε τον κώδικα μπορούμε να προσπαθήσουμε να βρούμε κάποια χρονοβόρα σημεία του. Αυτό μπορεί να γίνει με την βοήθεια ενός εργαλείου profiler. Αν βρούμε ποια είναι τα χρονοβόρα αυτά σημεία, τότε μπορούμε να ξεκινήσουμε να σκεφτόμαστε πως να τα βελτιώσουμε, είτε αλλάζοντας κάπως την σχεδίαση μας είτε γράφοντας κάπως πιο έξυπνα τον κώδικα μας.

3.3.4 Μειονεκτήματα βελτιστοποίησης προγράμματος

Κάποια βασικά μειονεκτήματα της βελτιστοποίησης προγράμματος που πρέπει να έχουμε υπόψη μας είναι τα εξής :

- Η βελτιστοποίηση μπορεί τις περισσότερες φορές να επιφέρει καινούργια προβλήματα και πιθανά σφάλματα (bugs) που να μην είναι τόσο εύκολη η ανίχνευση τους. Με άλλα λόγια, στην προσπάθεια μας να κάνουμε κάτι βέλτιστο μπορεί να καταστρέψουμε την δομή του προγράμματος μας επιφέροντας άλλα προβλήματα. Αυτό συμβαίνει κυρίως διότι για να γίνει μια βελτιστοποίηση χρειάζονται αλλαγές στον κώδικα, που δεν ελέγχονται αρκετά για την ορθότητα τους. Περισσότερο ελέγχονται αν πραγματικά βοηθούν στην βελτιστοποίηση. Επομένως, λογικά σφάλματα μπορούν εύκολα να προκύψουν.

- Η βελτιστοποίηση μπορεί σαν αποτέλεσμα να μας δώσει ένα τελικό δυσανάγνωστο και λιγότερο συντηρήσιμο κώδικα από τον αρχικό. Για παράδειγμα, η εύρεση του n-οστού αριθμού fibonacci είναι ένα γνωστό πρόβλημα που μπορεί να γραφτεί χρησιμοποιώντας αναδρομικό αλλά και μη-αναδρομικό τρόπο. Στο συγκεκριμένο πρόβλημα, η αναδρομή δίνει ένα πιο ευανάγνωστο κώδικα που είναι πιο

κοντά στον μαθηματικό τρόπο σκέψης επίλυσης του συγκεκριμένου προβλήματος. Παρόλα αυτά όμως, ο μη-αναδρομικός τρόπος που δεν είναι τόσο ευανάγνωστος, είναι πιο βέλτιστος.

- Πολλές φορές μπορεί να πετύχουμε στην αύξηση της ταχύτητας του κώδικα με αντίκτυπο όμως στην επεκτασιμότητα και επαναχρησιμοποίηση του. Αυτό συμβαίνει διότι ο κώδικας ενός βελτιστοποιημένου προγράμματος συνήθως είναι αρκετά πιο περίπλοκος και ακαταλαβίστικος από ότι είναι χωρίς βελτιστοποίηση. Αυτό μπορούμε να το παρατηρήσουμε και στο κεφάλαιο 4 όπου θα δούμε διαφορές στην επίδοση διαφόρων προγραμμάτων. Αν παρατηρήσουμε τους συγκεκριμένους κώδικες, εύκολα μπορούμε να προσέξουμε την διαφορά.

- Η βελτιστοποίηση του κώδικα σε ένα λειτουργικό σύστημα μπορεί να χειροτερέψει την απόδοση του σε ένα άλλο λειτουργικό σύστημα. Ένα πρόγραμμα σε Windows, μπορεί να χειρίζεται διαφορετικά κάποιες λειτουργίες από πλευράς υλικού (hardware) από ότι ένα πρόγραμμα σε Linux.

- Η βελτιστοποίηση προγράμματος είναι μια χρονοβόρα διαδικασία που πολλές φορές έχει σαν αποτέλεσμα την δημιουργία κώδικα μη κατανοητού. Για παράδειγμα, αν χρησιμοποιούμε ένα αλγόριθμο ταξινόμησης όπως αυτόν του bubblesort [12], σίγουρα ο κώδικας του συγκεκριμένου αλγόριθμου είναι αρκετά πιο απλός και κατανοητός σε σύγκριση με αυτούς του quicksort [12] και του mergesort [12] που είναι όμως πιο βέλτιστοι.

Έτσι προτού προχωρήσουμε σε τέτοιους υψηλούς στόχους, όπως είναι η βελτιστοποίηση προγράμματος, πρέπει πρώτα να σκεφτούμε τους λόγους για τους οποίους επιθυμούμε να βελτιστοποιήσουμε ένα πρόγραμμα και αν πραγματικά μπορεί να μας προσφέρει κάποιο όφελος.

3.4 Εισηγήσεις βελτιστοποίησης κώδικα

Πριν ξεκινήσουμε την αναφορά σε διάφορες εισηγήσεις για βελτιστοποίηση κώδικα, οφείλουμε να τονίσουμε ότι δεν υπάρχει κάποιος απόλυτος οδηγός, με τρόπους για βελτίωση της επίδοσης ενός προγράμματος. Αυτό συμβαίνει λόγω των διάφορων εφαρμογών που υπάρχουν και των διάφορων χρονοβόρων σημείων που μπορεί να περιλαμβάνει η καθεμιά, καθώς και διαφορετικών χαρακτηριστικών στην επίδοση. Επίσης, η επίδοση διαφέρει από hardware σε hardware (υλικό Η/Υ) και από λειτουργικό σύστημα σε λειτουργικό σύστημα.

Επίσης πρέπει να αναφέρουμε πως το καλύτερο όπλο που έχει κάποιος/κάποια απέναντι στην βελτιστοποίηση κώδικα είναι η βαθιά γνώση και εμπειρία που κατέχει, όσο αφορά την γλώσσα προγραμματισμού που χρησιμοποιεί. Όσο καλύτερα γνωρίζει τα χαρακτηριστικά της συγκεκριμένης γλώσσας προγραμματισμού, τόσο καλύτερα θα πετύχει τον στόχο του αν επιθυμεί την βελτιστοποίηση.

Στη επόμενο υποκεφάλαιο θα δούμε κάποιες εισηγήσεις βελτιστοποιήσεις που ισχύουν για την γλώσσα Java (μιας και είναι αυτή η γλώσσα που χρησιμοποιήθηκε για την εκπόνηση αυτής της εργασίας). Για την ώρα ας δούμε κάποιες από τις πιο σημαντικές εισηγήσεις σε γλώσσες προγραμματισμού που χρησιμοποιούν παρόμοιο συντακτικό με την Java (C, C++, C#, κτλ).

- Αν μία έκφραση επαναλαμβάνεται σε περισσότερα από ένα σημεία, την

εκτελούμε μόνο μια φορά και αποθηκεύουμε το αποτέλεσμα. Αυτό είναι ιδιαίτερα χρήσιμο σε περιπτώσεις όπου έχουμε loops (επαναληπτικούς βρόγχους). Έτσι, δεν θα χρειάζεται κάθε φορά να υπολογίζεται το αποτέλεσμα μιας έκφρασης μέσα στο loop. Για παράδειγμα, ο κώδικας:

```
int i;
for (i = 0; i < array.length; ++i)
```

Εικόνα 3.3: Παράδειγμα κομματιού κώδικα

μπορεί να αντικατασταθεί από τον

```
int i;
int length = array.length;
for (i = 0; i < length; ++i)
```

Εικόνα 3.4: Παράδειγμα κομματιού κώδικα

Παρόμοια, είναι καλά να αποφεύγουμε την δήλωση διαφόρων μεταβλητών μέσα στο σώμα ενός loop, διότι για κάθε επανάληψη θα δεσμεύεται ξανά μνήμη για αυτές τις μεταβλητές.

- Οι έλεγχοι που γίνονται στον κώδικα μας είναι συνετό να διακόπτονται όσο το δυνατό πιο γρήγορα. Για παράδειγμα :

```
if ( complicateCalculations(x) && (x > 0) )
```

Εικόνα 3.5: Παράδειγμα κομματιού κώδικα

μπορεί να γραφτεί ως :

```
if ( (x > 0) && (complicateCalculations(x)) )
```

Εικόνα 3.6: Παράδειγμα κομματιού κώδικα

Δηλαδή, μπορούμε να εκμεταλλευτούμε το short-circuiting (lazy evaluation) [7] που προσφέρουν οι περισσότερες γλώσσες προγραμματισμού. Στο πιο πάνω παράδειγμα μπορούμε να αναβάλουμε την κλήση της συνάρτησης *complicateCalculations* και να ελέγξουμε πρώτα αν x > 0. Αν αυτό αποτιμηθεί σε *false* τότε λόγω lazy evaluation δεν θα καλεστεί η συνάρτηση/μέθοδος *complicateCalculations*. Έτσι, γλιτώνουμε από το κάλεσμα αυτής της συνάρτησης/μεθόδου που μπορεί να έπαιρνε αρκετό χρόνο μέχρι να μας επέστρεφε κάποιο αποτέλεσμα.

Παρόμοια, ο πιο κάτω κώδικας :

```
negativeInputFound = false;
for (int i = 0; i < count; i++) {
    if (input[i] < 0) {
        negativeInputFound = true;
    }
}
```

Εικόνα 3.7: Παράδειγμα κομματιού κώδικα

μπορεί να αντικατασταθεί από τον

```
negativeInputFound = false;
for (int i = 0; i < count; i++) {
    if (input[i] < 0) {
        negativeInputFound = true;
        break;
    }
}
```

Εικόνα 3.8: Παράδειγμα κομματιού κώδικα

Δεν είναι απαραίτητο να προσπελαστεί ολόκληρο το loop αν έχουμε βρει αυτό που ψάχνουμε, έτσι μπορούμε να γλιτώσουμε κάποιο χρόνο διακόπτοντας την εκτέλεση του loop με την εντολή *break*.

- Μερικές φορές είναι δυνατή η επιτάχυνση ενός loop ελαττώνοντας τις επαναλήψεις του. Αυτό μπορεί να γίνει με το λεγόμενο loop unrolling [1,9,17,18,19,21] (ξετύλιγμα βρόγχων). Για παράδειγμα, ο κώδικας :

```
for (int i = 0; i < 65536; i++) {
    sum = sum + i;
}
```

Εικόνα 3.9: Παράδειγμα κομματιού κώδικα

μπορεί να ξαναγραφτεί ως εξής :

```
for (int i = 0; i < 65536; i = i + 8) {
    sum = sum + i;
    sum = sum + i + 1;
    sum = sum + i + 2;
    sum = sum + i + 3;
    sum = sum + i + 4;
    sum = sum + i + 5;
    sum = sum + i + 6;
    sum = sum + i + 7;

}
```

Εικόνα 3.10: Παράδειγμα κομματιού κώδικα

όπου το loop θα εκτελεστεί 65536/8 φορές. Όμως κάτι τέτοιες τεχνικές τις εκτελούν αυτόματα οι σημερινοί compilers. Παλαιότερα όμως, θα κέρδιζαν κάποιο χρόνο, έστω και αμελητέο.

- Σε κάποιες γλώσσες προγραμματισμού ο τρόπος αποθήκευσης πινάκων είναι κατά σειρά (row-major). Δηλαδή, αν έχουμε ένα δυσδιάστατο πίνακα a, τότε τα στοιχεία του θα φυλάγονταν ως εξής : a[0][0], a[0][1], a[0][2], κτλ και όχι σαν a[0][0], a[1][0], a[2][0] (column-major). Επομένως, έχοντας αυτό υπόψη μας, δεν πρέπει να κάνουμε άσκοπες εναλλαγές των loops, ασχέτως αν μας δίδουν το ίδιο αποτέλεσμα. Για παράδειγμα :

```
for (int i = 0; i < 10000; i++) {
    for (int j = 0; j < 100; j++) {
        sum = sum + array[i][j];
    }
}
```

Εικόνα 3.11: Παράδειγμα κομματιού κώδικα

δεν πρέπει να γραφτεί ως :

```
for (int j = 0; j < 100; j++) {
    for (int i = 0; i < 10000; i++) {
        sum = sum + array[i][j];
    }
}
```

Εικόνα 3.12: Παράδειγμα κομματιού κώδικα

αν χρησιμοποιούμε μια γλώσσα προγραμματισμού που αποθηκεύει σε row-major τους πινάκες της, διότι αυτό θα ήταν πιο χρονοβόρο. Ο λόγος είναι ότι στην πρώτη περίπτωση τα στοιχεία θα προσπελαστούν με τον τρόπο που αυτά είναι διαδοχικά αποθηκευμένα στην μνήμη, ενώ στην δεύτερη όχι.

- Αν γνωρίζουμε από τα χαρακτηριστικά μιας γλώσσας προγραμματισμού, ποιες εντολές ή έτοιμες μεθόδοι/συναρτήσεις είναι πιο φθηνές σε κόστος, σε σύγκριση με κάποιες άλλες που επιτελούν παρόμοια η την ίδια εργασία, τότε είναι καλύτερα να χρησιμοποιούμε αυτές που κοστίζουν λιγότερο.

- Μια άλλη καλή τακτική για βελτιστοποίηση είναι να αποθηκεύουμε κάποιες τιμές, που χρησιμοποιούνται ξανά και ξανά κατά την διάρκεια του προγράμματος μας. Για παράδειγμα, στις αντικειμενοστραφείς γλώσσες υπάρχουν έτοιμες μεθόδοι που υπολογίζουν το μέγεθος ενός αρχείου. Η κλήση αυτών των μεθόδων κρύβει κάποιο κόστος. Επομένως, σίγουρα είναι καλύτερα να καλέσουμε μια φορά την συγκεκριμένη μέθοδο και να αποθηκεύσουμε την τιμή της σε μια μεταβλητή και έπειτα να χρησιμοποιήσουμε αυτή την μεταβλητή, παρά να καλούμε κάθε φορά την ίδια την μέθοδο.

- Μια τεχνική βελτιστοποίησης είναι το λεγόμενο loop hoisting [18]. Αυτή είναι μια τεχνική όπου κάποια επιπρόσθετα-πλεονάζοντα στοιχεία από ένα εσωτερικό loop μεταφέρονται στο εξωτερικό loop. Βέβαια, δεν είναι πάντοτε απαραίτητο να έχουμε εξωτερικό και εσωτερικό loop. Για παράδειγμα, ο κώδικας :

```
for (int i = 0; i < MAX; i++) {
    if (firstFlag && secondFlag) {
        doOneThing(i);
    } else if (firstFlag && thirdFlag) {
        doSomethingElse(i);
    } else {
        doYetAnotherThing(i);
    }
}
```

Εικόνα 3.13: Παράδειγμα κομματιού κώδικα

χρησιμοποιώντας την ιδέα του “hoisting” (ανύψωσης) μπορεί να γραφτεί ως εξής :

```
if (firstFlag && secondFlag) {
    for (int i = 0; i < MAX; i++) {
        doOneThing(i);
    }
} else if (firstFlag && thirdFlag) {
    for (int i = 0; i < MAX; i++) {
        doSomethingElse(i);
    }
} else {
    for (int i = 0; i < MAX; i++) {
        doYetAnotherThing(i);
    }
}
```

Εικόνα 3.14: Παράδειγμα κομματιού κώδικα

Στο πιο πάνω κώδικα συγκεκριμένα έχουμε κάνει "hoist" (ανύψωση) τα "if" statements (δηλώσεις) έξω από το loop και τοποθετήσαμε σε κάθε "if" statement το συγκεκριμένο loop statement. Στο πρώτο παράδειγμα κώδικα (εικόνα 3.13) κάθε φορά που εκτελείται το loop statement πρέπει να αποφασιστεί ποιο από τα "if" statements πρέπει να εκτελεστεί, δίδοντας κάποιο επιπρόσθετο κόστος αυτή η απόφαση σε κάθε επανάληψη. Στο δεύτερο παράδειγμα κώδικα (εικόνα 3.14) αυτό το κόστος δεν υπάρχει πλέον αφού εξασφαλίζουμε πρώτα ποιο "if" statement θα εκτελεστεί και έπειτα εκτελείται το loop statement.

- Παρά τη χρήση των τελεστών της διαίρεσης και του πολλαπλασιασμού (που κοστίζουν περισσότερο από τους τελεστές της πρόσθεσης και της αφαίρεσης) είναι πιο βέλτιστο να χρησιμοποιήσουμε τους αντίστοιχους shift operators (αριστερής και δεξιάς ολίσθησης των bits) για ακέραιους που είναι δυνάμεις του 2. Για παράδειγμα, x >> 2 μπορεί να χρησιμοποιηθεί στη θέση του x / 4 και το x << 1, μπορεί να αντικαταστήσει το x * 2. Υπάρχουν και περιπτώσεις που μπορούμε να αποφύγουμε την χρήση του πολλαπλασιασμού και της διαίρεσης και να χρησιμοποιήσουμε τελεστές που είναι πιο βέλτιστοι όπως η πρόσθεση και η αφαίρεση. Για παράδειγμα, ο κώδικας :

```
for(i=0; i<10; i++) {
    printf("%d\n",i*10);
}
```

Εικόνα 3.15: Παράδειγμα κομματιού κώδικα

μπορεί να γραφτεί ως εξής :

```
for(i=0; i<100; i+=10) {
    printf("%d\n",i);
}
```

Εικόνα 3.16: Παράδειγμα κομματιού κώδικα

3.5 Εισηγήσεις βελτιστοποίησης κώδικα σε Java

Ας δούμε τώρα και κάποιες εισηγήσεις βελτιστοποίησης κώδικα σε Java. Να σημειώσουμε ότι κάποιες από τις εισηγήσεις που προτείνονται εδώ θα τις δούμε αναλυτικότερα στο κεφάλαιο 4, όπου θα διεξαχθούν και πειραματικές μετρήσεις με την χρήση ενός profiler.

- Ένα πολύ σημαντικό σημείο όσο αφορά την επίδοση στην Java είναι το γεγονός ότι τα strings (συμβολοσειρές) είναι αμετάβλητα [7,10,20] (immutable). Αυτό σημαίνει ότι έπειτα από την δημιουργία τους δεν αλλάζουν ποτέ. Για παράδειγμα :

```
String str = "Hello";
str = str + " world!";
```

το string “Hello”, μόλις δημιουργηθεί δεν αλλάζει, όμως η αναφορά (reference) στο string μπορεί να αλλάξει. Αρχικά, το *str* δείχνει στο “Hello”, όμως έπειτα δείχνει σε ένα καινούργιο string που δημιουργείται από την συνένωση του *str* με το “ world!”. Η συνένωση αυτή γίνεται με τον υπερφορτωμένο (overloaded) τελεστή “+” [7]. Κάθε φορά που χρησιμοποιείται ο τελεστής “+” για συνένωση strings, ο compiler της Java για να κάνει την συγκεκριμένη συνένωση, δημιουργεί ένα αντικείμενο (object) *StringBuilder*. Ο *StringBuilder* χρησιμοποιείται για δημιουργήσει (“κτίσει”) το string (στο παράδειγμα, το *str*). Κάθε φορά που χρησιμοποιείται ο τελεστής “+” γίνεται append (προσαρτίζεται) στο string, το string που βρίσκεται δεξιά του τελεστή “+” (στο παράδειγμα το “ world!”). Όταν γίνουν όλες οι προσαρτήσεις, έπειτα ο *StringBuilder* καλεί την μέθοδο *toString()* για να παραχθεί το νέο string. Ακολουθεί ένα παράδειγμα κώδικα όπου δημιουργείται ένα string με δυο τρόπους. Ο ένας τρόπος είναι με τον τελεστή “+” και ο άλλος με τον *StringBuillder*.

```
public class WhitherStringBuilder {
  public String implicit(String[] fields) {
    String result = "";
    for(int i = 0; i < fields.length; i++)
      result += fields[i];
    return result;
  }
  public String explicit(String[] fields) {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < fields.length; i++)
      result.append(fields[i]);
    return result.toString();
  }
} ///:~
```

Εικόνα 3.17: Παράδειγμα κώδικα [7]

Αν κάνουμε decompile τον πιο πάνω κώδικα με το εργαλείο *javap* που είναι μέρος του JDK [7] της Java, δίδοντας την εντολή *javap –c WitherStringBuilder*, τότε θα παραχθούν τα JVM (Java Virtual Machine) bytecodes [7,10] . Τα bytecodes της Java είναι κάτι ισοδύναμο της γλώσσας assembly. Ας δούμε λοιπόν τα bytecodes για την μέθοδο *implicit()* (εικόνα 3.17) στην οποία το string δημιουργείται με τον τελεστή "+".

```
public java.lang.String implicit(java.lang.String[]);
  Code:
   0:   ldc #2; //String
   2:   astore_2
   3:   iconst_0
   4:   istore_3
   5:   iload_3
   6:   aload_1
   7:   arraylength
   8:   if_icmpge   38
   11:  new #3; //class java/lang/StringBuilder
   14:  dup
   15:  invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V
   18:  aload_2
   19:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   22:  aload_1
   23:  iload_3
   24:  aaload
   25:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   28:  invokevirtual   #6; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
   31:  astore_2
   32:  iinc    3, 1
   35:  goto    5
   38:  aload_2
   39:  areturn
```

Εικόνα 3.18: Bytecodes μεθόδου implicit() [7]

Οι γραμμές 8: και 35: (εικόνα 3.18) αποτελούν ένα loop. Το σημαντικό που πρέπει να προσέξουμε εδώ είναι ότι η δημιουργία ενός αντικειμένου *StringBuilder* (εικόνα 3.18, γραμμή 15:) γίνεται μέσα στο loop, που σημαίνει ότι κάθε φορά που γίνεται προσπέλαση του loop θα παίρνουμε ένα καινούργιο αντικείμενο *StringBuilder*. Από την άλλη, η μέθοδος *explicit()* (εικόνα 3.17) κάνει χρήση του *StringBuilder*. Ας δούμε λοιπόν και τα bytecodes της μεθόδου *explicit()* :

```
public java.lang.String explicit(java.lang.String[]);
  Code:
    0:   new #3; //class java/lang/StringBuilder
    3:   dup
    4:   invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V
    7:   astore_2
    8:   iconst_0
    9:   istore_3
    10:  iload_3
    11:  aload_1
    12:  arraylength
    13:  if_icmpge   30
    16:  aload_2
    17:  aload_1
    18:  iload_3
    19:  aaload
    20:  invokevirtual   #5; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
    23:  pop
    24:  iinc    3, 1
    27:  goto    10
    30:  aload_2
    31:  invokevirtual   #6; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
    34:  areturn
```

Εικόνα 3.19: Bytecodes μεθόδου explicit() [7]

Εδώ παρατηρούμε ότι ο κώδικας του loop είναι πιο μικρός (εικόνα 3.19, γραμμές 13: με 27:) σε σύγκριση με πριν. Επίσης, η δημιουργία του αντικειμένου *StringBuilder* γίνεται μια φορά (εικόνα 3.19, γραμμή 4:) και όχι μέσα στο σώμα (body) του loop. Έτσι, σαφώς γλιτώνουμε από την αλόγιστη δημιουργία αντικειμένων, που είναι σπατάλη χώρου, μνήμης και κόστους στον garbage collector (“σκουπιδιάρη”) [7] που θα έρθει έπειτα να συλλέξει αυτά τα αντικείμενα. Επομένως, ο πιο βέλτιστος τρόπος για να δημιουργούμε string μέσα σε loops είναι με την χρήση του *StringBuilder*.

Δημιουργώντας ένα αντικείμενο *StringBuilder*, αν ξέρουμε από πριν το μέγεθος του string που θα δημιουργηθεί, καλά είναι να το χρησιμοποιήσουμε κατά την κατασκευή του. Για παράδειγμα αν ξέρουμε ότι θα έχει 40 χαρακτήρες, μπορούμε να γράψουμε το εξής : `StringBuilder sb = new StringBuilder(40);`
Έτσι δεν θα χρειάζεται να γίνεται αναπροσαρμογή του μεγέθους του *StringBuilder* κάθε φορά και σαν αποτέλεσμα η χρήση του θα γίνει ακόμα πιο γρήγορη.

- Στην String class (κλάση) είναι προτιμότερο η χρήση της έτοιμης μεθόδου *charAt()* έναντι της *startWith()* [20]. Η *startWith()* παρέχει την δυνατότητα επιστροφής true (αληθές) ή false (ψευδές), αν ένα string ξεκινά με μια συγκεκριμένη υποακολουθία string (substring). Από πλευράς επίδοσης η *startWith()* κάνει αρκετές συγκρίσεις προτού ετοιμαστεί να συγκρίνει το πρόθεμα (prefix) της με ένα άλλο string. Έτσι, επομένως αντί για το πιο κάτω :

```
if (s.startsWith("a") ) {
  // Do something here
}
```

Εικόνα 3.20: Παράδειγμα κομματιού κώδικα [20]

είναι προτιμότερο το εξής :

```
if ("a" == s.charAt(0) ) {
  // Do something here
}
```

Εικόνα 3.21: Παράδειγμα κομματιού κώδικα [20]

όπου η *charAt()* επιστρέφει την τιμή του χαρακτήρα στην θέση που θα της οριστεί.

- Είναι προτιμότερο να αποφεύγεται η δημιουργία αντικειμένων εκτός και αν απαιτείται [20]. Για παράδειγμα, αντί για την δήλωση :

```
Date myDate =new Date();
if (requiredCondition) {
    // usemyDate
}
```

Εικόνα 3.22: Παράδειγμα κομματιού κώδικα [20]

μπορούμε να δηλώσουμε το αντικείμενο *Date* μέσα στο "if" statement, έτσι ώστε να αποφύγουμε την δημιουργία του σε περίπτωση που το *requiredCondition* είναι false. Επομένως, θα έχουμε το εξής :

```
if (requiredCondition){
   Date myDate =new Date();
   // use myDate
}
```

Εικόνα 3.23: Παράδειγμα κομματιού κώδικα [20]

- Είναι καλά να αποφεύγεται η άσκοπη δημιουργία του ίδιου αντικειμένου δυο φορές [20]. Για παράδειγμα, στο ακόλουθο κομμάτι κώδικα :

```
public class x{
   privateVector v = new Vector();
   public x() {
     v = new Vector();
   }
}
```

Εικόνα 3.24: Παράδειγμα κομματιού κώδικα [20]

ο compiler δημιουργεί τον ακόλουθο κώδικα για τον constructor (κατασκευαστή) :

```
public x() {
   v = new Vector();
   v = new Vector();
}
```

Εικόνα 3.25: Παράδειγμα κομματιού κώδικα [20]

Οτιδήποτε έχει δημιουργηθεί σαν field (πεδίο) μιας κλάσης, ο κώδικας της δημιουργίας του αυτόματα (by default) μεταφέρεται στον constructor. Επομένως, αν έχει γίνει ήδη η δημιουργία ενός αντικειμένου σαν field μιας κλάσης, τότε πρέπει να αποφύγουμε την εκ νέου δημιουργία του στον constructor. Επομένως, ο κώδικας της εικόνας 3.24 μπορεί να γραφτεί ως εξής :

```
public classx {
   privateVector v;
   public x() {
     v = new Vector();
   }
}
```

Εικόνα 3.26: Παράδειγμα κομματιού κώδικα [20]

- Αν ξέρουμε εκ των προτέρων πόσο είναι το μέγεθος μιας έτοιμης δομής δεδομένων που θέλουμε να χρησιμοποιήσουμε (πχ. μιας λίστας, ενός πίνακα κατακερματισμού [12]) τότε είναι πιο βέλτιστο να χρησιμοποιήσουμε αυτή την πληροφορία. Ο λόγος είναι για αποφύγουμε την αυτόματη επέκταση και αναπροσαρμογή της συγκεκριμένης δομής δεδομένων, με στόχο να μπορεί να δεχθεί και άλλα στοιχεία. Αυτό έχει κάποιο κόστος και αν μπορούμε να το αποφύγουμε θα ήταν προς όφελος μας, όπως έχει αναφερθεί και πιο πάνω για τον *StringBuilder*.

- Οι *synchronized* (συγχρονισμένες) μεθόδοι χρησιμοποιούνται στον προγραμματισμό thread (νημάτων) [7,10]. Όταν θέλουμε να δώσουμε σε μια μέθοδο αποκλειστική πρόσβαση σε ένα στιγμιότυπο (instance) αντικειμένου τότε χρησιμοποιούμε προγραμματισμό με threads, έτσι ώστε η μέθοδος να μπορεί να εκτελέσει τις κατάλληλες ενημερώσεις δομών δεδομένων, χωρίς την παρέμβαση από άλλα threads. Τέτοιες μεθόδοι είναι πιο αργές από τις συνηθισμένες (μη-συγχρονισμένες), λόγω του κόστους λήψης ενός lock (κλειδώματος) στο αντικείμενο της μεθόδου. Επομένως, είναι καλά να το έχουμε υπόψη μας όταν προγραμματίζουμε χωρίς την χρήση thread (single-thread) ή όταν χρησιμοποιούμε Collection classes [7] (LinkedList, ArrayList, Vector κτλ.). Για παράδειγμα, το Vector class είναι thread-safe [7,10] ενώ το ArrayList δεν είναι. Όμως, το γεγονός ότι το Vector είναι thread-safe του προσδίδει επιπρόσθετο κόστος στην επίδοση. Επομένως, αν δεν προγραμματίζουμε με σκοπό την δημιουργία ενός προγράμματος που να κάνει χρήση threads, μπορούμε να χρησιμοποιήσουμε το ArrayList αντί του Vector.

- Όπως έχει αναφερθεί πιο πάνω το ArrayList έχει λιγότερο κόστος από ότι το Vector. Αυτό ισχύει γενικώς μεταξύ των έτοιμων δομών δεδομένων που προσφέρουν οι αντικειμενοστραφείς γλώσσες. Επομένως, κάποιος/κάποια που θέλει να ασχοληθεί με την βελτιστοποίηση κώδικα, καλό θα ήταν να μάθει περισσότερες λεπτομέρειες για

κάθε μια από αυτές και να δει ποια τον συμφέρει. Στο κεφάλαιο 4 θα δούμε πιο συγκεκριμένα, διαφορές στην επίδοση μεταξύ ArrayList και LinkedList.

- Επιπλέον, είναι καλά επίσης να γνωρίζουμε και τι είδους μεθόδους προσφέρει η κάθε κλάση. Για παράδειγμα η String class, έχει την μέθοδο *charAt()*, που έχουμε δει και πιο πάνω. Αν έχουμε ένα μεγάλο string, αντί να χρησιμοποιήσουμε την *charAt()* για να βρούμε την τιμή ενός χαρακτήρα μέσα στο string, είναι καλύτερα να το μετατρέψουμε σε στατικό πίνακα από χαρακτήρες (char[]). Έτσι, μπορούμε να βρούμε την αντίστοιχη θέση που θα αναζητούσαμε με τον *charAt()* μέσα στον πίνακα από χαρακτήρες και να βρούμε την επιθυμητή τιμή που αναζητούμε. Ο πίνακας αυτός μπορεί να δημιουργηθεί με την έτοιμη μέθοδο της String class, την *toCharArray()*.

Άλλο παράδειγμα, είναι η χρήση του *System.arraycopy()* [7,10]. Αυτή είναι μια μέθοδος που προσφέρει ένα πιο αποδοτικό τρόπο για αντιγραφή των στοιχείων ενός πίνακα σε ένα άλλο, έναντι άλλων τεχνικών όπως είναι η χρήση ενός "for" loop. Για παράδειγμα, αν έχουμε δυο πίνακες, array1 και array2, μήκους N και θέλουμε να αντιγράψουμε τα στοιχεία του array1 στο array2, γράφουμε το εξής : `System.arraycopy(array1, 0, array2, 0);`
Για πολύ μικρούς πίνακες όμως, η χρήση αυτής της μεθόδου μπορεί να μην ενδείκνυται λόγω του ίδιου του κόστους που υπάρχει στο να γίνει η κλήση της μεθόδου και να ελεγχθούν και οι παράμετροι της.

- Ο καλύτερος τρόπος με τον οποίο μπορεί να γίνει βελτιστοποίηση της επίδοσης των εισόδων και εξόδων (input/output) στην Java είναι με το λεγόμενο *buffering* [7,10]. Το *buffering* που λειτουργεί βασιζόμενο στην ιδέα της ενδιάμεσης μνήμη, επιτρέπει την αποθήκευση των εισόδων και εξόδων σε μεγαλύτερα κομμάτια (chunks) αντί για ένα χαρακτήρα ή byte (ψηφιολέξη) κάθε φορά. Αυτό μπορεί να παίξει σημαντικό ρόλο στην επίδοση. Λεπτομέρειες για αυτό θα δούμε στο κεφάλαιο 4.

- Όταν δημιουργείται ένα στιγμιότυπο μιας κλάσης, γίνεται αρχικοποίηση των μεταβλητών [10]. Οι μεταβλητές όμως που είναι δηλωμένες σαν *static* (στατικές) χρειάζονται να αρχικοποιηθούν μόνο μια φορά και έπειτα χρησιμοποιούνται χωρίς να επαναληφθεί ξανά η αρχικοποίηση τους. Η διαφορά μεταξύ των δυο ειδών

αρχικοποίησης (στατικών και μη-στατικών), μπορεί να παίξει καθοριστικό ρόλο στην επίδοση. Αυτό θα γίνει πιο ξεκάθαρο στο κεφάλαιο 4.

Κεφάλαιο 4

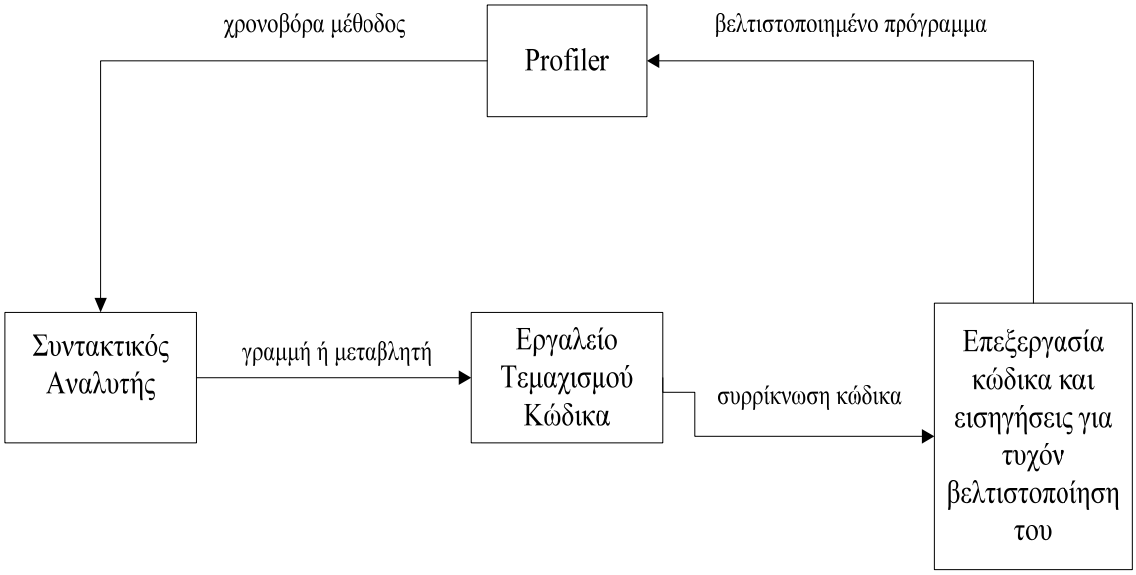
Παρουσίαση Μεθοδολογίας

4.1 Εισαγωγή

Σ' αυτό το κεφάλαιο θα γίνει η παρουσίαση της μεθοδολογίας που χρησιμοποιήθηκε για την αποπεράτωση αυτής της εργασίας. Πιο συγκεκριμένα, θα δούμε περισσότερες λεπτομέρειες για το πως λειτουργεί ο profiler του Netbeans IDE 6.5.1 [15] και παρουσίαση της λειτουργίας του συντακτικού αναλυτή καθώς και περιγραφή της υλοποίησης του. Επιπλέον, θα γίνει παρουσίαση και ενός εργαλείου τεμαχισμού που δυστυχώς δεν πραγματοποιήθηκε η συνένωση του σ' αυτή την εργασία. Τέλος, θα διεξαχθούν πειραματικές μετρήσεις (με την χρήση του profiler) βάση των πιο σημαντικών εισηγήσεων που έγιναν στο κεφάλαιο 3 για βελτιστοποίηση κώδικα. Οι συγκρίσεις θα γίνονται ανά ζεύγος αρχείων, που λύνουν το ίδιο πρόβλημα με βέλτιστο και με μη-βέλτιστο τρόπο και θα δούμε τις διαφορές στον χρόνο εκτέλεσης και των δύο.

Πρέπει να αναφέρουμε ότι η γλώσσα προγραμματισμού που έχει χρησιμοποιηθεί για την υλοποίηση του συντακτικού αναλυτή και τα προγράμματα που έχουν χρησιμοποιηθεί για ανάλυση της ταχύτητας τους με τον profiler, έχουν γραφτεί σε Java

[7]. Ο λόγος επιλογής της Java είναι διότι διαθέτει έτοιμη βιβλιοθήκη κανονικών εκφράσεων (regex) που έχει χρησιμοποιηθεί για την υλοποίηση του συντακτικού αναλυτή. Επιπρόσθετα, το Netbeans IDE, μπορεί να μας προσφέρει ένα έτοιμο built-in (ενσωματωμένο) profiler για profiling προγραμμάτων γραμμένα σε Java. Επιπλέον, από παλαιότερες διπλωματικές εργασίες έχουν γίνει προσπάθειες δημιουργίας ενός slicing tool (εργαλείου τεμαχισμού κώδικα) κάνοντας πάλι χρήση της γλώσσας Java. Ίσως μια μελλοντική προσθήκη σ' αυτό το εργαλείο, να είναι η ενσωμάτωση του συντακτικού αναλυτή που έχει παραχθεί για τις ανάγκες αυτής της διπλωματικής εργασίας. Ακολουθεί ένα σχεδιάγραμμα της μεθοδολογίας :



Σχήμα 4.1: Σχεδιάγραμμα Μεθοδολογίας

Στο σχήμα 4.1 βλέπουμε το σχεδιάγραμμα της μεθοδολογίας που ακολουθήθηκε. Αρχικά, μπορεί να γίνει η χρήση του profiler πάνω σε ένα εκτελέσιμο πρόγραμμα Java και να μας εντοπίσει τις πιο χρονοβόρες μεθόδους κάποιου προγράμματος. Έπειτα με την χρήση του συντακτικού αναλυτή μπορούν να βρεθούν για την συγκεκριμένη χρονοβόρα μέθοδο, οι γραμμές όπου υπάρχουν μεταβλητές, μαζί με τα αντίστοιχα ονόματα των μεταβλητών που υπάρχουν σε κάθε γραμμή.

Στη συνέχεια, μια από αυτές τις γραμμές ή μεταβλητές (αναλόγως τι παίρνει σαν είσοδο το slicing tool), μπορεί να δοθεί στο slicing tool. Να σημειώσουμε ότι δεν είναι απαραίτητο αρχικά να χρησιμοποιηθεί ο profiler. Αν ο/η χρήστης δεν επιθυμεί να

στηριχθεί στις μεθόδους που είναι πιο χρονοβόρες, ο συντακτικός αναλυτής μπορεί να προσφέρει και άλλες επιλογές. Αυτές οι επιπλέον επιλογές (εκτός από την επιλογή κάποιας μεθόδου), στηρίζονται στην εύρεση κομματιών κώδικα που θεωρούνται χρονοβόρα, όπως είναι τα loops (βρόγχοι), recursions (αναδρομές) και η χρήση του τελεστή modulus (υπόλοιπο διαίρεσης – '%'). Μέσα στην εμβέλεια αυτών των χρονοβόρων κομματιών, ο συντακτικός αναλυτής για ακόμα μια φορά, μπορεί να βρει τις γραμμές με τις αντίστοιχες μεταβλητές.

Ακολούθως, αφού δοθεί η γραμμή ή μεταβλητή στο slicing tool, αυτό με την σειρά του αναλαμβάνει τον τεμαχισμό του προγράμματος. Με τον τεμαχισμό, όπως αναφέραμε και στο κεφάλαιο 3, ο κώδικας μικραίνει (συρρικνώνεται) και απομένουν μόνο οι γραμμές του κώδικα που σχετίζονται με το κριτήριο που έχουμε θέσει. Αυτές πολύ πιθανόν να σχετίζονται με κάποιο χρονοβόρο κομμάτι, που μπορεί να τύχει βελτιστοποίησης.

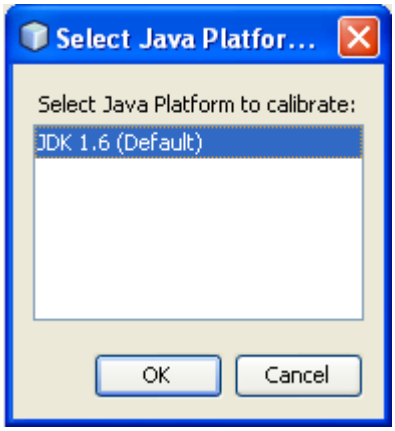
Στη συνέχεια, αφού μας δοθεί το αποτέλεσμα του slicing tool, το εξετάζουμε για εντοπισμό πιθανών βελτιστοποιήσεων βάση των εισηγήσεων για βελτιστοποίηση προγράμματος που έχουμε αναφέρει στο κεφάλαιο 3. Αν δεν εντοπίσουμε κάτι, δοκιμάζουμε και άλλες γραμμές ή άλλες επιλογές του συντακτικού αναλυτή μέχρι να εξαντλήσουμε κάθε πιθανή περίπτωση απομόνωσης κώδικα που μπορεί να μας βοηθήσει στο να εισηγηθούμε κάτι για βελτιστοποίηση του.

Τέλος, αν έχουν εντοπιστεί σημεία που μπορούν να τύχουν βελτιστοποίησης τότε κάνουμε τις απαραίτητες τροποποιήσεις στο κώδικα μας. Έπειτα, το βελτιστοποιημένο πρόγραμμα (σχήμα 4.1), μπορεί να επαναχρησιμοποιηθεί από τον profiler για να διαπιστωθεί αν έχουμε πραγματικά πετύχει τον στόχο μας. Ο profiler μπορεί να μας δείξει αν όντως έχουμε βελτιώσει τον χρόνο εκτέλεσης του συγκεκριμένου προγράμματος.

4.2 Profiler Netbeans IDE

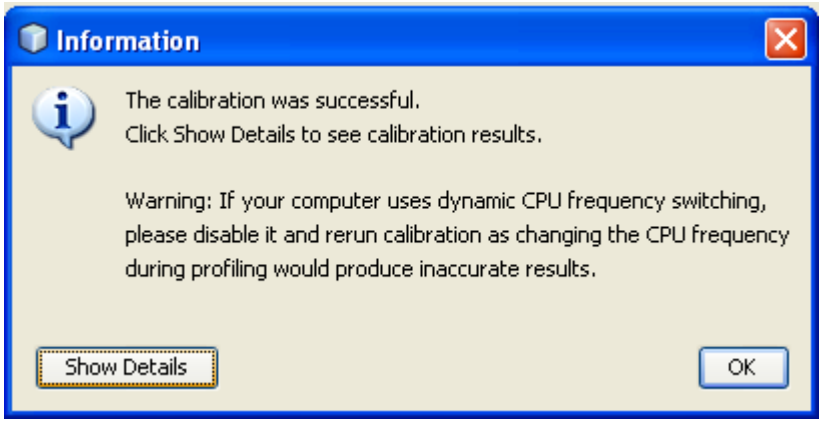
Ο profiler που έχει χρησιμοποιηθεί για τις ανάγκες αυτής της μεθοδολογίας, όπως έχει ήδη αναφερθεί, είναι αυτός του Netbeans IDE 6.5.1. Το NetBeans IDE περιλαμβάνει ένα ισχυρό εργαλείο για profiling που μπορεί να μας τροφοδοτήσει (γενικά μιλώντας) με τις σημαντικότερες πληροφορίες όσο αφορά τον χρόνο εκτέλεσης της εφαρμογής μας. Ο profiler του NetBeans μας επιτρέπει εύκολα να ελέγξουμε τις καταστάσεις των threads, την επίδοση του CPU (Κεντρική Μονάδα Επεξεργασίας) καθώς και την χρήση της μνήμης (memory usage) στην εφαρμογή μας.

Για να πετύχουμε πιο ακριβή αποτελέσματα με τον profiler, θα πρέπει πρώτα να κάνουμε ένα calibration (προσαρμογή μετρήσεων) βάσει του λειτουργικού συστήματος που χρησιμοποιούμε. Αν χρησιμοποιήσουμε για πρώτη φορά τον profiler, το ίδιο το IDE θα μας προτρέψει στο να κάνουμε το calibration. Το calibration χρειάζεται να γίνει μόνο μια φορά. Όμως, αν έχουμε πραγματοποιήσει κάποιες ουσιώδεις αλλαγές από πλευράς υλικού (hardware) που μπορούν να αλλάξουν την επίδοση της μηχανής μας, τότε αν θέλουμε να πάρουμε πιο ακριβή αποτελέσματα πρέπει να γίνει ξανά το calibration. Μπορούμε να επαναλάβουμε το calibration οποιαδήποτε στιγμή πηγαίνοντας στο menu του calibration και επιλέγοντας Advance commands > Run Profiler Calibration. Ακολούθως, θα εμφανιστεί η πιο κάτω οθόνη :



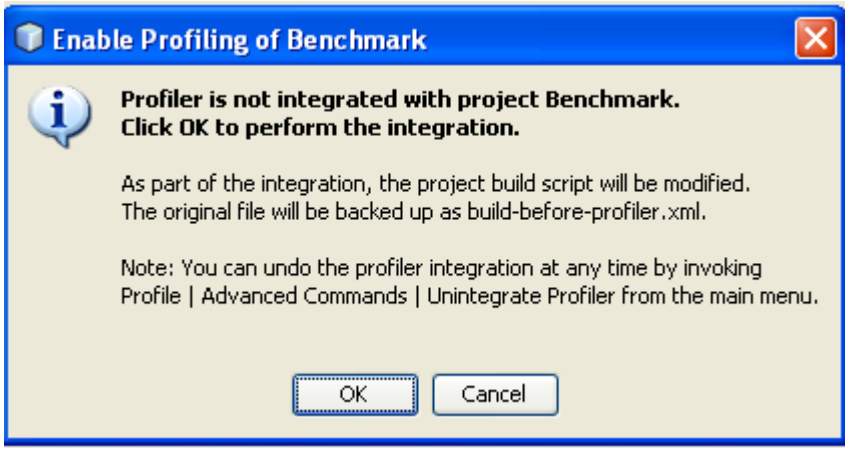
Εικόνα 4.1: Επιλογή εκτέλεσης του calibration

Πατώντας OK, θα εμφανιστεί μια άλλη οθόνη που θα μας επιβεβαιώσει πως έχει γίνει το calibration (εικόνα 4.2)



Εικόνα 4.2: Επιβεβαίωση ότι το calibration έχει πραγματοποιηθεί επιτυχώς

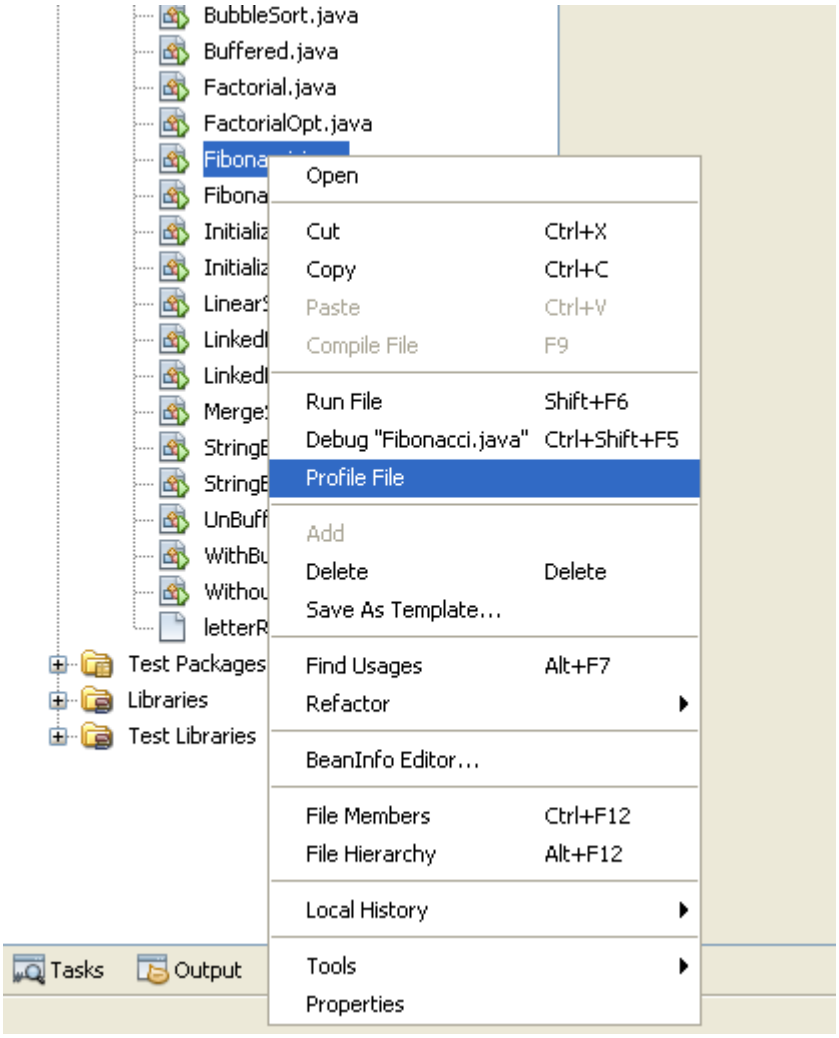
Την πρώτη φορά που θα επιχειρήσουμε να κάνουμε profile κάποιο project μέσα στο Netbeans θα εμφανιστεί η πιο κάτω οθόνη (εικόνα 4.3) που μας πληροφορεί ότι θα γίνει ενσωμάτωση (integration) του profiler με το συγκεκριμένο project που χρησιμοποιούμε.



Εικόνα 4.3: Προτροπή για ενσωμάτωση του profiler

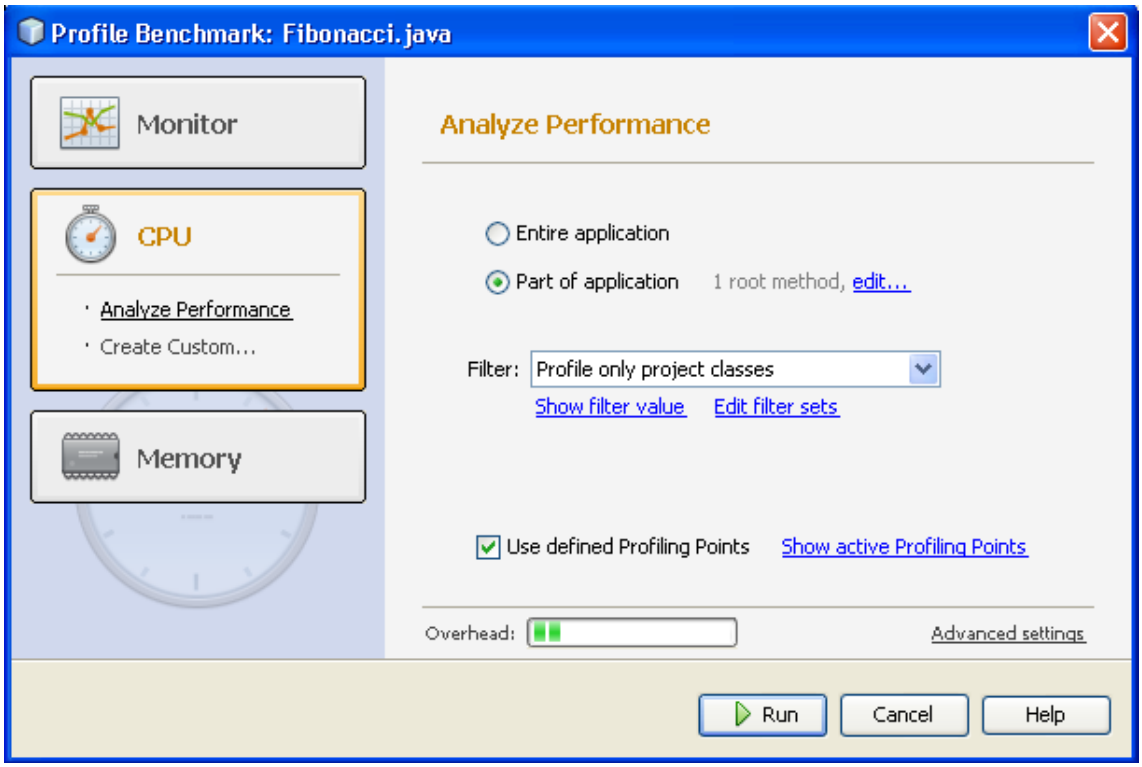
Έπειτα, μπορούμε να ξεκινήσουμε την διαδικασία για το profiling ενός αρχείου ή και ολόκληρου project. Από τις εργασίες (profiling task) που μας προσφέρει ο profiler για έλεγχο (έχουν αναφερθεί πιο πάνω), για τις ανάγκες της μεθοδολογίας μας χρησιμοποιήθηκε μόνο αυτή που έχει να κάνει με την επίδοση του CPU.

Για να αναλύσουμε την επίδοση του CPU κάποιου συγκεκριμένου αρχείου (στο παράδειγμα μας το Fibonacci.java), ο πιο απλός τρόπος είναι να κάνουμε δεξί click (πάτημα) στο αρχείο που επιθυμούμε και να επιλέξουμε “Profile File” (εικόνα 4.4).



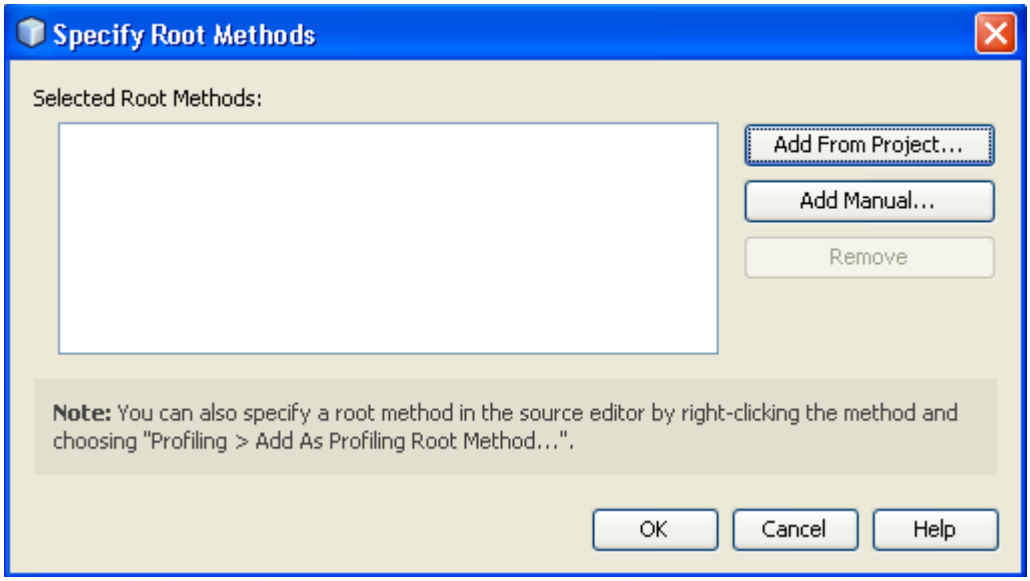
Εικόνα 4.4: Profiling ένα αρχείο

Έπειτα στην οθόνη που θα εμφανιστεί (εικόνα 4.5), κάνουμε click στο εικονίδιο του CPU και έπειτα επιλέγουμε το “Part of application” και ακολούθως το “edit…”.



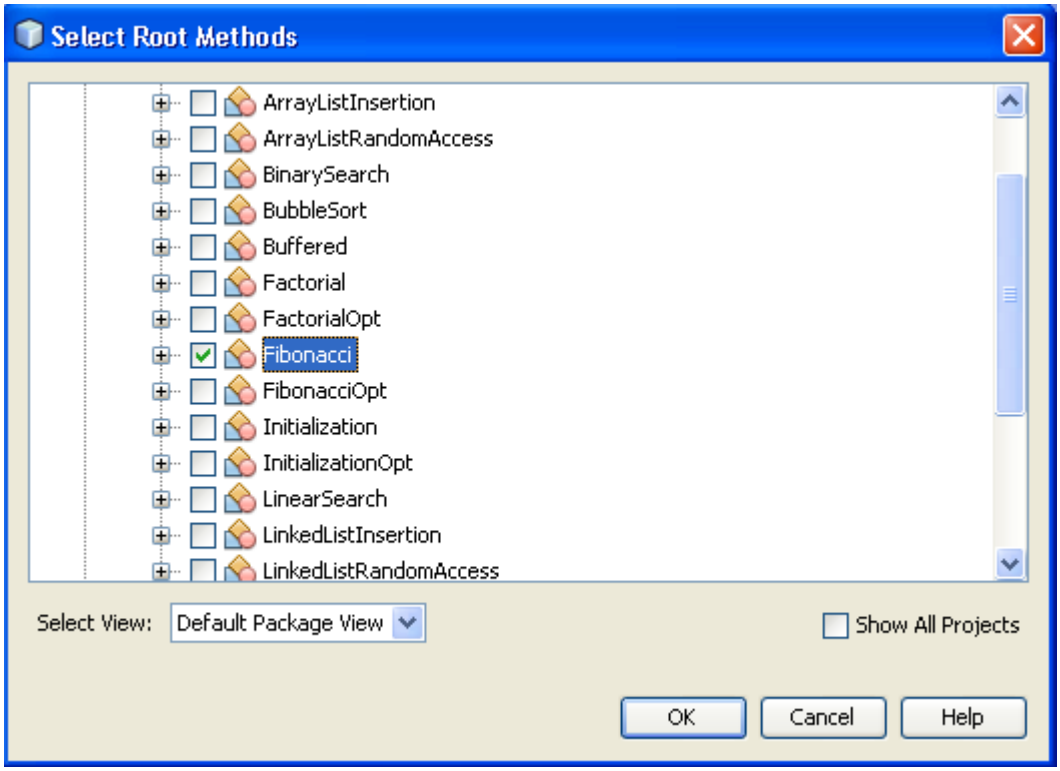
Εικόνα 4.5: Profiling του αρχείου Fibonacci.java

Μετά, στην οθόνη (εικόνα 4.6) που θα εμφανιστεί επιλέγουμε "Add From Project…"



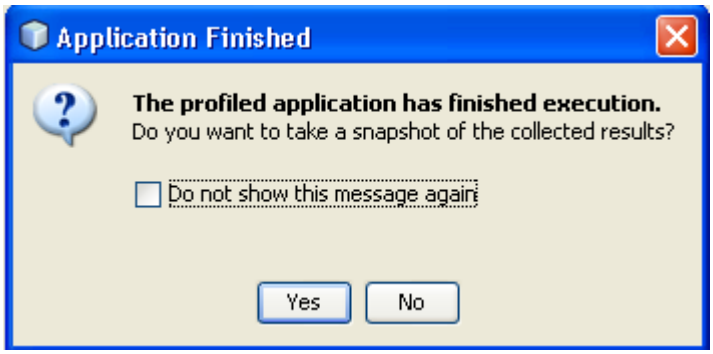
Εικόνα 4.6: Καθορισμός αρχείου

Στη συνέχεια, επιλέγουμε το αρχείο που θέλουμε να αναλύσουμε την επίδοση του (στο παράδειγμα μας, το Fibonacci.java, εικόνα 4.7) κάνοντας click στο κουτάκι που βρίσκεται στα αριστερά του.



Εικόνα 4.7: Επιλογή αρχείου

Τελειώνοντας, πατούμε "OK" σε όλα τα παράθυρα που έχουν ανοιχθεί (εικόνας 4.6 και 4.7) και στο αρχικό παράθυρο (εικόνα 4.5) κάνουμε click στο "Run". Στη συνέχεια, γίνεται ανάλυση της επίδοσης του αρχείου που μας ενδιαφέρει και όταν τελειώσει αυτή η ανάλυση, μπορούμε αν επιθυμούμε να δούμε ένα φωτογραφικό στιγμιότυπο (snapshot, εικόνα 4.8) των αποτελεσμάτων. Τέτοιου είδους snapshot αποτελεσμάτων θα δούμε προς το τέλος αυτού του κεφαλαίου.



Εικόνα 4.8: Επιλογή φωτογραφικού στιγμιότυπου (snapshot) αποτελεσμάτων

4.3 Περιγραφή υλοποίησης συντακτικού αναλυτή

Όπως είναι ήδη γνωστό, για την υλοποίηση αυτής της διπλωματικής εργασίας χρειάστηκε η δημιουργία ενός εργαλείου συντακτικής ανάλυσης (parser tool), με σκοπό την εύρεση κομματιών κώδικα που θεωρούνται χρονοβόρα, όπως είναι τα loops (επαναληπτικοί βρόγχοι), αναδρομικές μεθόδοι και η χρήση του τελεστή modulus. Επιπλέον, όπως αναφέραμε με την συνεργασία του profiler του Netbeans IDE 6.5.1 [15], μπορούν να βρεθούν οι πιο χρονοβόρες μεθόδοι (methods) και να τύχουν και αυτές επεξεργασίας. Έπειτα, θα μπορεί να γίνει η συλλογή των μεταβλητών που περικλείονται μέσα στην εμβέλεια του συγκεκριμένου κομματιού κώδικα (του χρονοβόρου) και των αντίστοιχων γραμμών όπου βρίσκονται αυτές οι μεταβλητές (μαζί με τις μεταβλητές συλλέγονται και οι σταθερές και τα χειριστήρια αντικειμένων (object handles), αλλά από εδώ και μπρος μιλώντας για μεταβλητές, θα υπονοούνται και τα υπόλοιπα, διότι ως επί το πλείστο τα περισσότερα είναι μεταβλητές). Η γραμμή είτε ή μεταβλητή αυτή, θα χρησιμοποιούνται ακολούθως από ένα άλλο εργαλείο που θα κάνει τεμαχισμό κώδικα (slicing tool) με αποτέλεσμα ο κώδικας να μικραίνει και να απομένουν μέσα σ' αυτόν μόνο οι γραμμές του κώδικα που σχετίζονται με το πιθανό χρονοβόρο κομμάτι, που μπορεί να τύχει βελτιστοποίησης.

Για την υλοποίηση, όπως έχει αναφερθεί και πιο πάνω, χρησιμοποιήθηκε η γλώσσα προγραμματισμού Java [7] και το περιβάλλον (IDE) του Eclipse 3.4.2 Ganymede [16]. Έχουν δημιουργηθεί τα εξής αρχεία: *Constants.java*, *DirectoryList.java*, *Formatter.java*, *HighlightLines.java*, *Loops.java*, *Methods.java*, *Modulus.java*, *ParserTool.java*, *Recursions.java*, *Scope.java*, *Variables.java*, *ReadInput.java*, *TextFile.java* [Παράρτημα Α].

Το αρχεία *ReadInput.java* και *TextFile.java*, είναι αρχεία που θα μπορούσαν να αποτελούν κομμάτι οποιουδήποτε προγράμματος Java. Το *ReadInput.java* διαβάζει από την οθόνη ένα ακέραιο αριθμό και ελέγχει αν είναι μεταξύ κάποιων επιτρεπόμενων ορίων (επιλογή του χρήστη στα menu), πχ. ακέραιος μεταξύ 1-5, διαφορετικά προτρέπεται ο χρήστης να ξαναδώσει ως είσοδο ένα αριθμό μεταξύ των σωστών ορίων. Το *TextFile.java* μπορεί να χρησιμοποιηθεί για το διάβασμα και γράψιμο αρχείων (οτιδήποτε αρχείων που περιέχουν μέσα κείμενο(text)).

Η λειτουργία του συντακτικού αναλυτή ξεκινά από το αρχείο *ParserTool.java*. Γίνεται χρήση του *DirectoryList.java* που σκοπό έχει ν' ανακτήσει τα αρχεία που βρίσκονται στην περιοχή (package) του Eclipse με ονομασία "testing", που θα χρησιμοποιηθούν αργότερα για έλεγχο αν περιέχουν χρονοβόρα κομμάτια κώδικα. Αν η περιοχή αυτή δεν περιέχει αρχεία τότε η λειτουργία του συντακτικού αναλυτή τερματίζεται και προτρέπεται ο χρήστης να εισάγει (import) αρχεία στην περιοχή αυτή, κάνοντας χρήση του "import" που προσφέρει ο Eclipse. Αρχικά, εμφανίζεται το menu και ζητείται από τον χρήστη να επιλέξει ποιο αρχείο επιθυμεί να εξετάσει.

Το αρχείο που επιλέγει ο χρήστης περνά από ένα είδος μορφοποίησης και μετασχηματισμού του κώδικα, έτσι ώστε να διευκολύνεται η διαδικασία της συντακτικής ανάλυσης (parsing), με την χρήση κανονικών εκφράσεων [7,14]. Υπεύθυνο για αυτού του είδους την μορφοποίηση είναι το αρχείο *Formatter.java*.

Η βασική ιδέα της λειτουργίας του formatter, είναι να διαβάζει ένα αρχείο Java, ν' απαλείφει όλων των ειδών τα σχόλια που υπάρχουν στο αρχείο (είτε με /* ... */, είτε με //...) και να εξασφαλίζει ώστε κάθε γραμμή να τερματίζεται με '{', ή '}', ή ';'. Για να γίνει αυτό, πρώτα διαβάζεται το αρχείο και αποθηκεύεται όλο σαν ένα string (συμβολοσειρά). Έπειτα, αυτό το string-αρχείο μετασχηματίζεται μέχρι να φτάσει στην τελική του μορφή όπως περιγράφτηκε πιο πάνω.

Αρχικά, εντοπίζονται όλα τα 'for' loops που υπάρχουν μέσα στον κώδικα, διαβάζοντας γραμμή προς γραμμή το string-αρχείο και αποθηκεύονται σαν μια λίστα από strings. Ο τρόπος με τον οποίο εντοπίζονται τα 'for' loops γίνεται με την βοήθεια μιας κανονικής έκφρασης που λαμβάνει υπόψη την σύνταξη ενός 'for' loop:
`for(`*`initialization`*`; `*`termination`*`; `*`increment`*`)`. Ο λόγος που γίνεται εντοπισμός των 'for' loops είναι διότι αργότερα θα αντικατασταθούν προσωρινά από ένα μοναδικό string, έτσι ώστε, στο τέλος όταν θα ξεκινήσει να πραγματοποιείται ο χωρισμός βάσει των χαρακτήρων ';', '{', '}' και να εισάγεται το new line (χαρακτήρας αλλαγής γραμμής) μετά από αυτούς τους χαρακτήρες, να μην υπάρχουν προβλήματα χωρισμού του 'for' loop με new line, λόγω του ότι περιέχει ';' στην σύνταξη του. Ταυτόχρονα, εξετάζεται αν το 'for' loop περικλείεται μέσα σε σχόλια (όπου η σύνταξη των σχολίων φαίνεται πιο πάνω). Αυτό πάλι επιτυγχάνεται χρησιμοποιώντας μια κανονική έκφραση

για σχόλια. Αν υπάρχουν ‘for’ loop μέσα σε σχόλια τότε αυτά αγνοούνται και δεν αποθηκεύονται. Αυτό, όπως θα φανεί και παρακάτω, επιβάλλεται διότι στο τέλος οτιδήποτε έχει αντικατασταθεί προσωρινά από ένα μοναδικό string, θα επανέλθει στην αρχική μορφή που είχε. Επειδή όμως τα σχόλια στο τέλος θα απαλειφθούν, αν αποθηκεύονταν και τα ‘for’ loops που υπήρχαν μέσα σε σχόλια τότε θα παρουσιαζόταν πρόβλημα στην τελική αντικατάσταση. Θα παρουσιαζόταν πρόβλημα εξαιτίας του γεγονός, ότι η αντικατάσταση πραγματοποιείται με την σειρά εμφάνισης τους στον κώδικα και λόγω της απαλοιφής των σχολίων και των ‘for’ loops που θα χάνονταν μαζί τους, στο τέλος θα υπήρχαν πιο πολλά αποθηκευμένα, ενώ στην ουσία θα είχαν απομείνει πιο λίγα.

Έπειτα, διαβάζεται γραμμή προς γραμμή το string-αρχείο και πραγματοποιείται η αντικατάσταση των ‘for’ loops με το string, “for (myForPatent)X” (όπου X ένας αύξων αριθμός, ξεκινώντας από το μηδέν) και αποθηκεύεται σαν το νέο string-αρχείο.

Ακολούθως, μετά τα ‘for’ loops πρέπει να εντοπιστούν όλα τα string quotes (εισαγωγικά συμβολοσειρών) πχ. “...” και όλα τα character quotes πχ. ‘.’ (εισαγωγικά χαρακτήρων). Ο σκοπός που χρειάζεται να εντοπιστούν και να αποθηκευτούν τα character quotes είναι διότι μέσα σ’ αυτά μπορεί να περικλείονται οι χαρακτήρες ‘;’, ‘{‘, ‘}’, που αργότερα μετά από αυτούς θα εισαχθεί ο χαρακτήρας new line (αλλαγής γραμμής). Αν αυτό δεν είχε επιτευχθεί, θα καταστρεφόταν η δομή του formatter, διότι το ένα quote (‘) θα βρισκόταν μόνο του σε μια γραμμή και πρέπει κάθε γραμμή να τελειώνει με ‘;’, ‘{‘, ‘}’, έτσι ώστε να διευκολύνεται αργότερα το parsing για εύρεση των μεταβλητών.

Ο σκοπός που χρειάζεται να εντοπιστούν και να αποθηκευτούν τα string quotes είναι ο ίδιος με αυτό των character quotes και επιπλέον ότι μέσα στα string quotes μπορεί να περικλείεται οτιδήποτε, όπως πχ.

```
System.out.println(" { } ; /*inside println*/")
```

που μπορεί να ξεγελάσει την κανονική έκφραση των σχολίων, ότι το `/*inside println*/`, είναι σχόλιο ενώ στην ουσία περικλείεται μέσα στο string quote του `System.out.println` και δεν είναι σχόλιο.

Ταυτόχρονα, εξετάζεται αν αυτά τα quotes (string & character) βρίσκονται μέσα σε σχόλια, έτσι ώστε να μην αποθηκευτούν, για τους ίδιους λόγους που ισχύουν πιο πάνω για τα 'for' loops. Έπειτα, διαβάζεται γραμμή προς γραμμή το string-αρχείο και πραγματοποιείται η αντικατάσταση των string quotes με "myPatentX" και των character quotes με 'X' και ακολούθως αποθηκεύεται το νέο string-αρχείο (όπου X ένας αύξων αριθμός, ξεκινώντας από το μηδέν).

Στη συνέχεια, απαλείφονται όλα τα σχόλια με την βοήθεια της κανονικής έκφρασης για τα σχόλια. Ακολούθως, όλοι οι χαρακτήρες new line (αλλαγής γραμμής) και tab επίσης απαλείφονται και σαν αποτέλεσμα οι εντολές του string-αρχείου, βρίσκονται τώρα όλες σε μια γραμμή. Μετά, διαβάζεται ξανά το string-αρχείο και όπου εντοπιστούν χαρακτήρες ';', '{', '}', τοποθετείται στη συνέχεια ο χαρακτήρας new line και αποθηκεύεται το νέο string-αρχείο.

Στο τέλος, διαβάζεται για ακόμη μια φορά το string-αρχείο και αναλόγως με την κανονική έκφραση που αντιστοιχεί στα 'for' loops, string και character quotes επαναφέρονται όλα τα αρχικά 'for' loops, string και character quotes από την λίστα όπου είχαν αποθηκευτεί και δημιουργείται το τελικό string-αρχείο. Το τελικό string-αρχείο αποθηκεύεται στην περιοχή "testing/files" του Eclipse.

Αφού γίνει η μορφοποίηση του αρχείου με την βοήθεια του *Formatter.java*, σειρά έχει το *Variables.java*. Το *Variables.java* βρίσκει από την περιοχή "testing/files" του Eclipse, το αρχείο που έχει δημιουργηθεί και συλλέγει με την βοήθεια κανονικών εκφράσεων όλες τις μεταβλητές που υπάρχουν σε αυτό το string-αρχείο. Τα σημεία σ' ένα κώδικα Java, όπου μπορούν να υπάρχουν δηλώσεις μεταβλητών είναι στα fields (πεδία) μιας κλάσης (class), στις local variables (τοπικές μεταβλητές) μιας μεθόδου (method), μέσα στις παραμέτρους μιας μεθόδου, μέσα στις παραμέτρους ενός constructor (κατασκευαστή), μέσα στη δήλωση ενός catch clause (δήλωση ενός try/catch block για exceptions) και μέσα στη δήλωση ενός 'for' loop ή ενός 'foreach' loop.

Αρχικά, διαβάζεται το αρχείο και αποθηκεύεται σαν ένα string. Πριν ξεκινήσει η εύρεση των μεταβλητών πρέπει από το string-αρχείο ν' αντικατασταθούν όλα τα string

quotes και τα character quotes, όπως παρόμοια έγινε και στον *Formatter* και να αποθηκευτεί σαν το καινούργιο string-αρχείο με όλες τις αλλαγές. Ο σκοπός που γίνεται αυτό είναι διότι τα quotes μπορούν να περικλείουν οποιαδήποτε έκφραση μέσα τους και να ξεγελάσουν την κανονική έκφραση πως έχει βρεθεί αυτό που αναζητείται.

Έτσι, αναγκαστικά έπρεπε να βρεθεί ένας τρόπος ουδετεροποίησης των quotes. Πχ. αν έχουμε μια κανονική έκφραση που ψάχνει να βρει δηλώσεις μεταβλητών και έχουμε την ακόλουθη εντολή:

```
System.out.println("integer declaration example :int a;");
```

το 'int a;' στο τέλος του string quote μέσα στο println θα ξεγελούσε την κανονική έκφραση και θα το συμπεριλάμβανε σαν δήλωση μεταβλητής. Όλα τα string quotes αντικαθιστούνται με την λέξη "myPatent" και όλα τα char quotes με τον χαρακτήρα '!'.

Ακολούθως, εντοπίζονται τα ονόματα των constructor και αποθηκεύονται σε μια λίστα. Ο τρόπος με τον οποίο γίνεται η εύρεση των constructor, είναι με την χρήση ενός μηχανισμού της Java, του reflection mechanism [7] (μηχανισμού αντικατοπτρισμού). Ο reflection mechanism, μπορεί την ώρα του runtime (εκτέλεσης) ενός προγράμματος Java να εντοπίσει τα ονόματα των constructors, methods και fields καθώς και άλλες λειτουργίες που δεν χρειάστηκαν για την υλοποίηση του συντακτικού αναλυτή. Τα ονόματα των constructors επιστρέφονται σε μορφή qualified name, δηλαδή αν η κλάση βρίσκεται σ' ένα package (πακέτο) πχ. 'testing' και το όνομα του constructor είναι πχ. Test, τότε το qualified name που θα επιστραφεί σαν όνομα του constructor είναι το 'testing.Test'. Όμως, αυτό δεν θα εξυπηρετούσε αργότερα όταν θα γινόταν αναζήτηση των δηλώσεων των μεταβλητών μέσα στις παραμέτρους του constructor, διότι η αντίστοιχη κανονική έκφραση δεν θα έβρισκε ταίρι. Πχ. αν είχαμε `public Test(String str)` αυτό θα εμφανιζόταν σαν `public testing.Test(String str)` και το γεγονός ότι του ονόματος του constructor προηγείται η τελεία ('.'), θα εμπόδιζε την κανονική έκφραση από το να ταιριάξει. Γι' αυτό με την βοήθεια κανονικής έκφρασης που βρίσκει ταίρι μόνο αν υπάρχει [A-Za-z_0-9] που μετά ακολουθείται από τελεία ('.'), εντοπίζονται και απαλείφονται τα qualified name και απομένει μόνο το τελικό όνομα (όπου στο πιο πάνω παράδειγμα είναι το Test).

Αφού έχουν εντοπιστεί τα ονόματα των constructor έπειτα είναι δυνατό με την βοήθεια μιας άλλης κανονικής έκφρασης να εντοπιστούν και οι παράμετροι του constructor. Όταν εντοπιστούν και οι παράμετροι, τότε γίνεται συλλογή των ονομάτων των παραμέτρων-μεταβλητών, όπου αποθηκεύονται σε μια λίστα που διατηρεί την ιδιότητα του συνόλου. Δηλαδή, αν ξαναβρεθεί το ίδιο όνομα μεταβλητής, δεν θα εισαχθεί στην λίστα. Το περιεχόμενο των παραμέτρων ενός constructor και γενικότερα των μεθόδων είναι της μορφής : *modifier var_type var_name comma* όπου *modifier* μπορεί να είναι μόνο *final* αν υπάρχει και πρέπει να ακολουθείται υποχρεωτικά από *var_type* και *var_name* όπου είναι της μορφής αλφαριθμητικού ([A-Za-z_0-9]) και μετά από ένα ή κανένα κόμμα και ξανά η δήλωση της κανονικής έκφρασης από την αρχή,
modifier var_type var_name comma modifier var_type var_name comma, κτλ. Έτσι, κάνοντας χρήση μιας κανονικής έκφρασης που λαμβάνει υπόψη της την πιο πάνω σύνταξη, εντοπίζονται και αποθηκεύονται τα ονόματα των παραμέτρων-μεταβλητών (*var_name* στην συγκεκριμένη περίπτωση).

Δυστυχώς, η χρήση του reflection αν και μπορεί να χρησιμοποιηθεί για την εύρεση ονομάτων των fields, δεν μπορεί να χρησιμοποιηθεί για την εύρεση των ονομάτων των local variables που υπάρχουν μέσα σε μεθόδους. Έτσι, προτιμότερο είναι η χρησιμοποίηση κανονικής έκφρασης και για τα δυο μαζί, επειδή η σύνταξη των fields και local variables είναι η ίδια. Η χρήση του reflection δεν μπορεί να χρησιμοποιηθεί ούτε για την εύρεση των ονομάτων των μεθόδων διότι με την χρήση του reflection αν ο τύπος επιστροφής της μεθόδου είναι generic πχ. *List<?>* η πληροφορία αυτή χάνεται και μένει μόνο το *List*. Έτσι, η κανονική έκφραση που θα είχε χρησιμοποιηθεί για την εύρεση των παραμέτρων των μεθόδων δεν θα έβρισκε ταίρι και θα χάνονταν οι παράμετροι-μεταβλητές από όσες μεθόδους επιστρέφουν τύπο generic. Πχ. αν έχουμε μια δήλωση μεθόδου : *List<?> listReturned (List<?> list)* με την χρήση reflection θα γίνει : *List listReturned (List list)* και έτσι η κανονική έκφραση που θα χρησιμοποιηθεί αργότερα για να βρει τα ονόματα των παραμέτρων-μεταβλητών δεν θα βρίσκει ταίρι.

Έτσι, αντί να χρησιμοποιείται το reflection για εύρεση των ονομάτων των μεθόδων χρησιμοποιείται από την αρχή μια κανονική έκφραση που βρίσκει τόσο το όνομα όσο και τις παραμέτρους.

Στη συνέχεια, εντοπίζονται οι μεταβλητές-παράμετροι ή καλύτερα object handles, των catch clause των exceptions, χρησιμοποιώντας κανονική έκφραση που εκμεταλλεύεται την ίδια σύνταξη που χρησιμοποιείται και για τις παραμέτρους των constructor, με την μόνη διαφορά ότι δεν υπάρχει το κόμμα πλέον και είναι μόνο *modifier var_type var_name*. Αυτό γίνεται διότι μόνο μια παράμετρος μπορεί να υπάρχει μέσα στο catch clause ενός try/catch block. Μετά, αποθηκεύονται οι μεταβλητές που βρέθηκαν, στην ίδια λίστα με αυτή που αποθηκεύτηκαν οι μεταβλητές που πάρθηκαν από τους constructors.

Ακολούθως, γίνεται συλλογή των μεταβλητών που υπάρχουν μέσα στις δηλώσεις των 'for' loops. Ένα 'for' loop μπορεί να έχει την ακόλουθη μορφή :
`for (int j = 0, k; j < 10 ;j++).` Τώρα όμως, η κανονική έκφραση χρειάζεται να λάβει υπόψη της και το '=' και όχι μόνο το κόμμα που υπήρχε στην περίπτωση των constructors. Έτσι, τώρα πρέπει να έχει την ακόλουθη σύνταξη : *modifier var_type var_name more_vars* όπου modifier, var_type και var_name ισχύουν τα ίδια με πιο πάνω. Όπου more_var, σημαίνει ύπαρξη '=' και κάποιας αρχικοποίησης μετά όπου μπορεί να ακολουθείται από κόμμα (',') και να τελειώνει όμως αναγκαστικά με ερωτηματικό (';'). Όταν βρεθούν οι μεταβλητές, αποθηκεύονται στη λίστα όπου νωρίτερα αποθηκεύτηκαν οι μεταβλητές που πάρθηκαν από τους constructors και τα catch clause και όπου γενικά θα αποθηκεύονται όσες μεταβλητές εντοπίζονται.

Στη συνέχεια, εντοπίζονται τα 'foreach' loops με την βοήθεια κανονικής έκφρασης που κάνει χρήση της σύνταξης ενός 'foreach' loop : `for (type var : arr)`
Αμέσως μετά, εντοπίζονται οι μεταβλητές του 'foreach' με την κανονική έκφραση που χρησιμοποιήθηκε για το catch clause των try/catch block. Όταν εντοπιστούν οι μεταβλητές, αποθηκεύονται στην λίστα για τις μεταβλητές.

Έπειτα, πρέπει να εντοπιστούν όλες οι μεθόδοι, έτσι ώστε μετά να παρθούν οι παράμετροι- μεταβλητές από αυτές. Οι μεθόδοι βρίσκονται χρησιμοποιώντας κανονική έκφραση που εκμεταλλεύεται την ακόλουθη σύνταξη :
modifier returned_type generics brackets method_name parameters
Όπου *modifier* αυτή την φορά μπορεί να είναι ένας συνδυασμός των public, private, protected, final, static ή και κανένα από όλα αυτά. Ακολουθείται μετά από το

returned_type που είναι το ίδιο με το var_type που αναφέρθηκε πιο πάνω. Στη συνέχεια, μπορεί να υπάρχουν *generics* ή *brackets*, ή και τα δύο, είτε και κανένα από τα δύο. Τα *generics* ξεκινούν με '<' και κλείνουν με '>' και μεταξύ τους μπορεί να υπάρχει οποιαδήποτε αλφαριθμητικό ([A-Za-z_0-9]) ή ο χαρακτήρας '?' ή ξανά να περικλείονται μέσα σε άλλα "<…>". Τα *brackets* ξεκινούν με '[' και κλείνουν με ']' και μετά μπορεί να ακολουθούνται και από αλλά "[]", σε περίπτωση πολυδιάστατου πίνακα. Το *method_name* είναι το ίδιο με το *var_name* που χρησιμοποιήθηκε πιο πάνω. Το *parameters* είναι οτιδήποτε περικλείεται μέσα σε παρενθέσεις "(…)" και ακολουθείτε από δεξιά αγκυλωτή παρένθεση "{".

Στη συνέχεια, εντοπίζονται οι παράμετροι-μεταβλητές κάνοντας χρήση της ίδιας κανονικής έκφρασης που χρησιμοποιήθηκε για την εύρεση των μεταβλητών στην δήλωση ενός 'for' loop και αποθηκεύονται στην λίστα με τις μεταβλητές.
Τέλος, σειρά έχουν τα fields μιας κλάσης και τα local variables. Τα fields και τα local variables εντοπίζονται χρησιμοποιώντας κανονική έκφραση που έχει την ακόλουθη σύνταξη : *modifier var_type generics brackets var_name more_vars*.
Όπου *modifier* έχει την ίδια ερμηνεία με αυτή που χρησιμοποιήθηκε πιο πάνω για τις μεθόδους. Τα υπόλοιπα έχουν επίσης χρησιμοποιηθεί και στις υπόλοιπες κανονικές εκφράσεις πιο πάνω. Έτσι, εντοπίζονται πάλι οι μεταβλητές και αποθηκεύονται στην λίστα τους.

Αφού έχουν εντοπιστεί οι μεταβλητές του αρχείου που επέλεξε ο χρήστης, έπειτα προτρέπεται (ο χρήστης) να διαλέξει τι πιθανό χρονοβόρο κομμάτι θέλει να εξετάσει έχοντας τις ακόλουθες επιλογές : 'for loops', 'while loops', 'recursions', 'modulus operator' και 'choose a method'. Αν επιλέξει 'for loops' τότε εμφανίζονται στην οθόνη οι γραμμές όπου υπάρχουν μεταβλητές και τα αντίστοιχα ονόματα τους, που περικλείονται στην εμβέλεια των 'for' και των 'foreach' loops. Αν επιλέξει 'while loops' τότε εμφανίζονται στην οθόνη οι γραμμές όπου υπάρχουν μεταβλητές και τα αντίστοιχα ονόματα τους, που περικλείονται στην εμβέλεια των 'while' και των 'do-while' loops. Αν επιλέξει 'recursions', τότε εντοπίζονται οι αναδρομικές (recursive) μεθόδοι και εμφανίζονται στην οθόνη οι γραμμές όπου υπάρχουν μεταβλητές και τα αντίστοιχα ονόματα τους, που περικλείονται στην εμβέλεια αυτής της αναδρομικής μεθόδου. Αν επιλέξει 'modulus operator' τότε εμφανίζονται στην οθόνη οι γραμμές που

συμπεριλαμβάνουν τον τελεστή ‘%’ και οι αντίστοιχες μεταβλητές που συμβάλουν στην πράξη, κάνοντας χρήση αυτού του τελεστή.

Αν η επιλογή του χρήστη ήταν είτε ‘for loops’ είτε ‘while loops’ τότε χρησιμοποιείται το *Loops.java*. Σκοπός του είναι να εντοπίσει τις μεταβλητές που χρησιμοποιούνται για ένα συγκεκριμένο είδος loop και να τις αποθηκεύσει σ’ ένα hash map (πίνακα κατακερματισμού) όπου γίνεται αποθήκευση σε μορφή ζευγαριών (pairs) , κλειδιού και τιμής (key,value). Σαν κλειδί (key) χρησιμοποιείται η γραμμή που εντοπίστηκαν οι μεταβλητές και σαν τιμή (value), ένα string που περιέχει τα ονόματα μεταβλητών στην συγκεκριμένη γραμμή.

Η γλώσσα προγραμματισμού Java, έχει δύο είδη ‘while loops’, όπως έχει και δύο είδη ‘for loops’. Επομένως, για την εύρεση των ‘while loops’ χρειάζονται επιπρόσθετα δύο κανονικές εκφράσεις που να λαμβάνουν υπόψη τους την σύνταξη αυτού του είδους loop. Η σύνταξη ενός ‘while loop’ είναι η ακόλουθη :

```
while (expression) {
     statement(s)
}
```

και η σύνταξη ενός ‘do-while loop’ η εξής :

```
do {
     statement(s)
} while (expression);
```

Επομένως, το *Loops.java* είναι υπεύθυνο αρχικά να εντοπίσει περί τίνος loop πρόκειται και έπειτα να βρει την εμβέλεια αυτών των loops και τις αντίστοιχες μεταβλητές που περικλείονται σ’ αυτή την εμβέλεια Πριν γίνει αυτό, κάθε φορά γίνεται το κόλπο της ουδετεροποίησης των string και character quotes, για τους ίδιους ακριβώς λόγους όπως χρησιμοποιήθηκε πιο πάνω. Για την εύρεση των μεταβλητών χρησιμοποιείται η κανονική έκφραση του *var_name*.

Για την εύρεση της εμβέλειας ενός loop και γενικότερα μιας μεθόδου, είναι υπεύθυνο το *Scope.java*. Η εμβέλεια καθορίζεται από την αριστερή αγκυλωτή παρένθεση (‘{’), όπου δηλώνει την αρχή της εμβέλειας και την δεξιά αγκυλωτή παρένθεση (‘}’), που δηλώνει το τέλος. Όσο αφορά τα loops που περιέχουν μια γραμμή κώδικα και δεν

χρειάζεται να περικλείονται από ‘{…}’, στα δικά μας “test” αρχεία αυτό απαγορεύεται διότι χάνεται ο τρόπος εύρεσης της εμβέλειας του loop. Ο τρόπος με τον οποίο ανιχνεύεται η εμβέλεια, είναι κάθε φορά που εντοπίζεται αριστερή αγκυλωτή παρένθεση ‘{’ αυξάνεται ένας μετρητής κατά ένα και κάθε φορά που βρίσκεται δεξιά αγκυλωτή παρένθεση (‘}’), αυτός ο μετρητής μειώνεται. Έτσι, ελέγχεται η τιμή του μετρητή και υπολογίζεται κατά πόσο το loop ή μέθοδος βρίσκονται εντός ή εκτός εμβέλειας.

Στη συνέχεια, αφού εντοπιστούν και αποθηκευτούν οι μεταβλητές στο hash map για το συγκεκριμένο είδος loop, αυτό ταξινομείται σε αύξουσα σειρά με βάση την γραμμή που βρέθηκε η συγκεκριμένη μεταβλητή.

Να σημειώσουμε, πως κατά την διάρκεια που εντοπίζεται ένα είδος loop τυπώνονται στην οθόνη τα αποτελέσματα του (γραμμές και αντίστοιχες μεταβλητές που βρίσκονται σε κάθε γραμμή). Τα loops τυπώνονται με την σειρά που αυτά βρέθηκαν στον κώδικα. Πχ. αν έχει βρεθεί το πρώτο ‘for’ loop, τότε στην οθόνη θα εμφανιστεί ‘For loop 1’, αν έχει βρεθεί το δεύτερο, τότε θα εμφανιστεί ‘For loop 2’ και ούτω καθεξής. Αν αυτό το loop περικλείει ένα ή περισσότερα εσωτερικά loop του ίδιου είδους τότε αυτά απαριθμούνται σαν υποτμήμα του εξωτερικού loop που τα περικλείει. Πχ. αν το εξωτερικό loop είναι το 1, τότε τα εσωτερικά loop απαριθμούνται σαν 1.1, 1.2 και τα λοιπά. Επομένως, αν ο χρήστης επέλεξε ’for’ loops τότε θα απαριθμηθούν όλα τα ‘for’ loops με την σειρά με την οποία βρέθηκαν μαζί με τις αντίστοιχες πληροφορίες (γραμμές – μεταβλητές) και έπειτα θα απαριθμηθούν όλα τα ‘foreach’ loops με τον ίδιο τρόπο. Αν ο χρήστης επέλεξε ’while’ loops τότε θα απαριθμηθούν όλα τα ‘while’ loops με τον ίδιο τρόπο όπως έγινε και με τα ‘for’ loops και έπειτα θα απαριθμηθούν όλα τα ‘do-while’ loops.

Αν ο χρήστης επιλέξει ‘recursions’ τότε γίνεται χρήση του *Recursions.java*. Πάλι και εδώ στην αρχή διαβάζεται το αρχείο που επέλεξε ο χρήστης και γίνεται ουδετεροποίηση των string και character quotes. Μετά, βρίσκονται οι μεθόδοι με την βοήθεια της αντίστοιχης κανονικής έκφρασης που αναφέρθηκε πιο πάνω. Κάθε φορά που βρίσκεται μια μέθοδος ελέγχεται αν μέσα στην εμβέλεια της (με την βοήθεια του *Scope.java*), υπάρχει ξανά κλήση του εαυτού της. Ο εντοπισμός της ύπαρξης κλήσης του εαυτού της

μεθόδου γίνεται με την κανονική έκφραση : *methodName parenthesis*. Όπου *methodName*, το όνομα της μεθόδου και *parenthesis*, οτιδήποτε περικλείετε μέσα σε αριστερή και δεξιά παρένθεση (‘(...)’) Αν εντοπιστεί ξανά κλήση του εαυτού της μεθόδου, ελέγχοντας αν το όνομα της μεθόδου (*method_name*) της μιας κανονικής έκφρασης είναι το ίδιο με το όνομα της μεθόδου της άλλης κανονικής έκφρασης (*methodName*), τότε οι μεταβλητές που έχουν βρεθεί με την βοήθεια πάλι της κανονικής έκφρασης *var_name*, αποθηκεύονται στο hash map. Αν δεν έχει εντοπιστεί ξανά κλήση του εαυτού της μεθόδου δεν αποθηκεύεται τίποτα στο hash map. Σε περίπτωση που έχει βρεθεί αναδρομική μέθοδος τότε τυπώνεται στην οθόνη το όνομα της αναδρομικής μεθόδου και ακολούθως όλες οι γραμμές όπου υπάρχουν μεταβλητές, μαζί με τις αντίστοιχες μεταβλητές, στην εμβέλεια εκείνης της αναδρομικής μεθόδου.

Αν ο χρήστης επιλέξει ‘modulus operator’, τότε γίνεται χρήση του *Modulus.java*. Αναζητείται ο τελεστής modulus μέσα στο αρχείο που επέλεξε ο χρήστης και στις γραμμές που βρεθεί, αποθηκεύονται οι μεταβλητές στο hash map, όπως έχει γίνει και πιο πάνω. Να σημειώσουμε, ότι στην αρχή γίνεται και πάλι η ουδετεροποίηση των σχολίων. Έπειτα τυπώνονται στην οθόνη τα αποτελέσματα (γραμμές-μεταβλητές) βάση των στοιχείων του hash map που έχουν αποθηκευτεί.

Αν πάλι ο χρήστης επιλέξει ‘choose a method’ τότε εντοπίζονται και αποθηκεύονται όλα τα ονόματα μεθόδων που υπάρχουν στο συγκεκριμένο αρχείο που επέλεξε ο χρήστης. Έπειτα, βάση των ονομάτων που έχουν εντοπιστεί δημιουργείται ένα menu το οποίο απαριθμεί τα ονόματα των μεθόδων που έχουν εντοπιστεί και προτρέπεται ο χρήστης να επιλέξει μια από αυτές. Να σημειώσουμε πως η επιλογή της μεθόδου που θα διαλέξει ο χρήστης μπορεί να υποβοηθηθεί από ένα profiler (στην περίπτωση μας αυτός του Netbeans IDE) που μπορεί να υποδείξει στον χρήστη ποιες μεθόδοι είναι οι πιο χρονοβόρες. Αν έχει επιλεχθεί ένα αρχείο το οποίο δεν περιέχει καθόλου μεθόδους, τότε ο χρήστης θα ενημερωθεί με το κατάλληλο μήνυμα. Διαφορετικά, αν υπάρχουν μεθόδοι, τότε θα γίνει η ίδια διαδικασία που έγινε για την εύρεση γραμμών-μεταβλητών των αναδρομικών μεθόδων. Η μόνη διάφορα είναι ότι τώρα δεν θα αναζητείται η εύρεση αναδρομικής μεθόδου, αλλά η εύρεση μεθόδου που έχει το ίδιο όνομα με αυτό που επέλεξε ο χρήστης. Αυτό πάλι γίνεται με την βοήθεια της αντίστοιχης κανονικής έκφρασης για μεθόδους που έχει χρησιμοποιηθεί πιο πάνω. Στο τέλος, πάλι τυπώνονται

στην οθόνη τα αποτελέσματα (γραμμές-μεταβλητές) βάση των στοιχείων του hash map που έχουν αποθηκευτεί.

Είναι σημαντικό να αναφερθεί ότι σε περίπτωση που ο χρήστης επιλέξει μια από τις πιο πάνω επιλογές : 'for loops', 'while loops', 'recursions', 'modulus operator', 'choose a method' και δεν βρεθούν μεταβλητές μέσα στην εμβέλεια της αναζήτησης, τότε στην οθόνη θα εμφανιστεί και το ανάλογο μήνυμα.

Πρέπει να αναφερθεί επίσης πως όταν ο χρήστης έχει κάνει μια από τις πιο πάνω επιλογές και έχουν εντοπιστεί μεταβλητές, τότε δημιουργείται στην περιοχή "/testing/htmlview" ένα αντίτυπο html αρχείο, το ίδιο μ' αυτό που δημιουργήθηκε στην περιοχή "/testing/files", που φωσφορίζει (highlight) τις γραμμές με τις αντίστοιχες μεταβλητές. Αυτό επιτυγχάνεται διαβάζοντας γραμμή προς γραμμή και περικλείοντας σε html markup tags τα περιεχόμενα του αντίστοιχου αρχείου από την περιοχή "/testing/files". Με την βοήθεια του hash map, όπου έχουν αποθηκευτεί οι αριθμοί των γραμμών, γίνεται ο φωσφορισμός των σωστών γραμμών (χρησιμοποιώντας τα αντίστοιχα html markup tags). Υπεύθυνο για την δημιουργία του html αρχείου είναι το *HighlightLines.java*.

Τελειώνοντας, διαγράφονται τα στοιχεία του hash map που έχει χρησιμοποιηθεί για την αποθήκευση των γραμμών-μεταβλητών, καθώς και των υπόλοιπων λιστών που έχουν χρησιμοποιηθεί, ώστε να είναι έτοιμα για επαναχρησιμοποίηση.

Το *Constants.java*, είναι υπεύθυνο για την αποθήκευση όλων των σταθερών της υλοποίησης του συντακτικού αναλυτή. Αυτές οι σταθερές, περιλαμβάνουν κανονικές εκφράσεις, πρότυπα κανονικών εκφράσεων και strings.

4.4 Παρουσίαση συντακτικού αναλυτή

Στο κομμάτι αυτό θα γίνει μια παρουσίαση της λειτουργίας του συντακτικού αναλυτή. Μόλις ξεκινήσουμε τον συντακτικό αναλυτή, εμφανίζεται ένα menu, που προτρέπει τον/την χρήστη να επιλέξει ένα αρχείο για συντακτική ανάλυση (εικόνα 4.9).

```
======================================
Select the test file you want to use:
======================================

1. Load file Factorial.java
2. Load file Fibonacci.java
3. Load file ForLoops.java
4. Load file Input.java
5. Load file ModulusOperatorExample.java
6. Load file RecursionSample.java
7. Load file WhileLoops.java
8. Exit

Enter your choice (Must be from 1 to 8) :
```

Εικόνα 4.9: Αρχικό menu parser

Έστω ότι επιλέγουμε το αρχείο "*ForLoops.java*" [Παράρτημα Β-26], γράφοντας 3 και πατώντας enter. Τότε, θα εμφανιστεί στην οθόνη μας ένα άλλο menu, προτρέποντας μας να επιλέξουμε πιο "χρονοβόρο" κομμάτι θέλουμε να επιλέξουμε (εικόνα 4.10).

```
============================================================
Select what time consuming code parts you want to check out:
============================================================

1. For loops
2. While loops
3. Recursions
4. Modulus Operator
5. Choose a method
6. Return to load files menu

Enter your choice (Must be from 1 to 6) :
```

Εικόνα 4.10: Επιλογή "χρονοβόρου" κομματιού

Αν επιλέξουμε το 1, θα δούμε τα σημεία όπου υπάρχουν "for" loops μαζί με τις αντίστοιχες μεταβλητές που περικλείονται στην εμβέλεια τους. Τα αποτελέσματα αυτής της επιλογής φαίνονται στην πιο κάτω εικόνα (εικόνα 4.11).

```
========================================================================

===== For Loop 1 =====
Line: 8 Vars: i,i,i

===== For Loop 1.1 =====
Line: 9 Vars: j,j,i,j
Line: 10 Vars: i

===== For Loop 2 =====
Line: 14 Vars: i,i,i

===== For Loop 2.1 =====
Line: 15 Vars: j,j,j

===== For Loop 2.2 =====
Line: 16 Vars: m,m,m

===== For Loop 2.3 =====
Line: 17 Vars: n,n,n
Line: 18 Vars: n,m,j,i

===== For Loop 3 =====
Line: 23 Vars: i,i,i
Line: 24 Vars: f,i,rand

===== For Loop 4 =====
Line: 33 Vars: k

===== ForEach Loop 1 =====
Line: 26 Vars: x,f
Line: 27 Vars: x

===== ForEach Loop 1.1 =====
Line: 28 Vars: g2,f
Line: 29 Vars: g2

========================================================================
```

Εικόνα 4.11: Αποτελέσματα επιλογής

Το "μορφοποιημένο" αρχείο που αυτόματα παράχθηκε στην περιοχή "testing/files" φαίνεται στην πιο κάτω εικόνα (εικόνα 4.12).

```
1  package testing.files;
2   import java.util.Random;
3  public class ForLoops {
4  public static void main(String[] args) {
5  Random rand = new Random(47);
6  float[] f = new float[10];
7  int k = 0;
8  for (int i = 1; i <= 5; i++) {
9  for (int j = 1; j <= i; j++) {
10 System.out.print(i);
11 }
12 System.out.println();
13 }
14 for (int i = 0; i < 10; i++) {
15 for (int j = 0; j < 10; j++) {
16 for (int m = 0; m < 10; m++) {
17 for (int n = 0; n < 10; n++) {
18 System.out.println(n*m*j*i);
19 }
20 }
21 }
22 }
23 for (int i = 0; i < 10; i++) {
24 f[i] = rand.nextFloat();
25 }
26 for (float x : f) {
27 System.out.println(x);
28 for (float g2 : f) {
29 System.out.println(g2);
30 }
31 }
32 for (;;) {
33 System.out.println(k++);
34 }
35 }
36 }
```

Εικόνα 4.12: Αρχείο "*ForLoops.java*" μετά την μορφοποίηση του

Το highlighted html αρχείο που αυτόματα παράχθηκε στην περιοχή "testing/html" φαίνεται στην πιο κάτω εικόνα (εικόνα 4.13).

```
1       package testing.files;
2        import java.util.Random;
3       public class ForLoops {
4       public static void main(String[] args) {
5       Random rand = new Random(47);
6       float[] f = new float[10];
7       int k = 0;
8       for (int i = 1; i <= 5; i++) {
9       for (int j = 1; j <= i; j++) {
10      System.out.print(i);
11      }
12      System.out.println();
13      }
14      for (int i = 0; i < 10; i++) {
15      for (int j = 0; j < 10; j++) {
16      for (int m = 0; m < 10; m++) {
17      for (int n = 0; n < 10; n++) {
18      System.out.println(n*m*j*i);
19      }
20      }
21      }
22      }
23      for (int i = 0; i < 10; i++) {
24      f[i] = rand.nextFloat();
25      }
26      for (float x : f) {
27      System.out.println(x);
28      for (float g2 : f) {
29      System.out.println(g2);
30      }
31      }
32      for (;;) {
33      System.out.println(k++);
34      }
35      }
36      }
```

Εικόνα 4.13: Αρχείο "*ForLoops.java*" με highlighted τις γραμμές-μεταβλητές των "for" loops

Αν ο/η χρήστης επέλεγε στο menu της εικόνας 4.10 (μιλώντας για το συγκεκριμένο αρχείο, "*ForLoops.java*"), την επιλογή 2, "while" loops, λόγω του ότι δεν υπάρχουν στο συγκεκριμένο αρχείο γραμμές με "while" loops που να περικλείουν κάποιες μεταβλητές, θα εμφανιστεί το πιο κάτω μήνυμα (εικόνα 4.14)

```
========================================================================

No variables were found in "while" loops' scope of this file!

========================================================================
```

Εικόνα 4.14: Εμφάνιση μηνύματος

Παρόμοια, αν ο/η χρήστης είχε επιλέξει 3 ή 4 (recursion ή modulo), λόγω του ότι πάλι δεν βρέθηκαν κάποιες μεταβλητές μέσα στις συγκεκριμένες εμβέλειες, θα εμφανιστούν τα αντίστοιχα μηνύματα στις εικόνες 4.15 και 4.16.

```
==========================================================================

No recursion methods' variables were found in the scope of this file!

==========================================================================
```

Εικόνα 4.15: Εμφάνιση μηνύματος

```
==========================================================================

No modulus operator was found in this file, in order to
get the involving variables in the equation!

==========================================================================
```

Εικόνα 4.16: Εμφάνιση μηνύματος

Πρέπει να αναφέρουμε ότι σε περίπτωση που δεν έχουν βρεθεί μεταβλητές είναι λογικό πως δεν θα γίνει highlight κάτι στο html αρχείο που παράγεται.

Αν τώρα στο αρχικό menu (εικόνα 4.9) επιλέγαμε το αρχείο "*WhileLoops.java*" [Παράρτημα Β-29] και έπειτα (εικόνα 4.10) διαλέγαμε να εξετάσουμε τα "while" loops τότε θα παίρναμε τα αποτελέσματα που φαίνονται στην εικόνα 4.17.

```
========================================================================

===== While Loop 1 =====
Line: 18 Vars: n
Line: 19 Vars: n

===== While Loop 1.1 =====
Line: 20 Vars: m
Line: 21 Vars: m

===== While Loop 2 =====
Line: 26 Vars: j
Line: 28 Vars: j

===== Do-While Loop 1 =====
Line: 11 Vars: i
Line: 12 Vars: i
Line: 14 Vars: i

========================================================================
```

Εικόνα 4.17: Αποτελέσματα επιλογής

Το αντίστοιχο highlighted html αρχείο φαίνεται πιο κάτω στην εικόνα 4.18.

```
1       package testing.files;
2       public class WhileLoops {
3       static boolean condition() {
4       boolean result = Math.random() < 0.99;
5       System.out.print(result + ", ");
6       return result;
7       }
8       public static void main(String[] args) {
9       int i = 0;
10       do {
11      System.out.println("i is : " + i);
12      i++;
13      }
14       while (i < 5);
15      int n = 0;
16      int m = 0;
17      i = 0;
18      while (n < 5) {
19      System.out.println(++n);
20      while (m < 5) {
21      System.out.println(++m);
22      }
23      }
24      System.out.println("Exited 'while'");
25      while(true) {
26      int j = 0;
27      System.out.println("Infinite While");
28      System.out.println(j);
29      }
30      }
31      }
```

Εικόνα 4.18: Αρχείο "*WhileLoops.java*" με highlighted τις γραμμές-μεταβλητές των "while" loops

Αν διαλέγαμε για το συγκεκριμένο αρχείο, να εξετάσουμε τα “for” loops που δεν υπάρχουν, τότε θα εμφανιζόταν αντίστοιχα το ακόλουθο μήνυμα (εικόνα 4.19).

```
========================================================================

No variables were found in "for" loops' scope of this file!

========================================================================
```

Εικόνα 4.19: Εμφάνιση μηνύματος

Στην συνέχεια, μετά την εμφάνιση των αποτελεσμάτων επιστρέφουμε πίσω στο αρχικό menu (εικόνα 4.9). Αν τώρα επιλέγαμε το αρχείο “*RecursionSample.java*” [Παράρτημα Β-28] και έπειτα (εικόνα 4.10) διαλέγαμε να εξετάσουμε τα “recursions” τότε θα παίρναμε τα αποτελέσματα που φαίνονται στην εικόνα 4.20.

```
========================================================================

===== Found Recursive Method : factorial =====
Line: 4 Vars: n
Line: 8 Vars: n,n

===== Found Recursive Method : fib =====
Line: 23 Vars: n
Line: 24 Vars: n
Line: 27 Vars: n,n

========================================================================
```

Εικόνα 4.20: Αποτελέσματα επιλογής

Το αντίστοιχο highlighted html αρχείο φαίνεται πιο κάτω, στην εικόνα 4.21.

```
1       package testing.files;
2       public class RecursionSample {
3           public int factorial (int n)    {
4               if (n == 0) {
5                   return 1;
6               }
7        else {
8                   return n * factorial (n - 1);
9               }
10          }
11                  public int fibonacci (int n)    {
12              int previous = -1;
13              int result = 1;
14              for (int i = 0; i <= n; )        {
15                  int sum = result + previous;
16                  previous = result;
17                  result = sum;
18                  i = i + 1;
19              }
20              return result;
21          }
22                  public static long fib(int n) {
23              if (n <= 1) {
24              return n;
25              }
26              else {
27              return fib(n-1) + fib(n-2);
28              }
29          }
30          public static void main(String[] args) {
31                  int num = Integer.parseInt(args[0]);
32              for (int i = 1; i <= num; i++) {
33              System.out.println(i + ": " + fib(i));
34              }
35          }
36      }
```

Εικόνα 4.21: Αρχείο "*RecursionSample.java*" με highlighted τις γραμμές-μεταβλητές των recursive methods

Αν τώρα για το συγκεκριμένο αρχείο, στο δεύτερο menu (εικόνα 4.10), επιλέγαμε την επιλογή 5, "Choose a method", τότε θα εμφανιζόταν το ακόλουθο menu (εικόνα 4.22).

```
======================================
Select the method you want to use:
======================================

1. Method name : factorial
2. Method name : fibonacci
3. Method name : fib
4. Method name : main
5. Return to load files menu

Enter your choice (Must be from 1 to 5) :
```

Εικόνα 4.22: Επιλογή από μεθόδους του αρχείου

Με την βοήθεια του profiler, θα μπορούσαμε να δούμε ότι η πιο χρονοβόρα μέθοδος στην εικόνα 4.22, είναι η "fib". Αν στη συνέχεια επιθυμούμε να εξετάσουμε από το συγκεκριμένο αρχείο την μέθοδο "fib" και επιλέγαμε το 3, τότε θα παίρναμε τα αποτελέσματα που φαίνονται στην εικόνα 4.23.

```
=========================================================================

===== Found Method : fib =====
Line: 23 Vars: n
Line: 24 Vars: n
Line: 27 Vars: n,n

=========================================================================
```

Εικόνα 4.23: Αποτελέσματα επιλογής

Το αντίστοιχο highlighted html αρχείο για τα πιο πάνω αποτελέσματα φαίνεται πιο κάτω, στην εικόνα 4.24.

```
1       package testing.files;
2       public class RecursionSample {
3           public int factorial (int n)    {
4               if (n == 0) {
5                   return 1;
6               }
7        else {
8                   return n * factorial (n - 1);
9               }
10          }
11                  public int fibonacci (int n)    {
12              int previous = -1;
13              int result = 1;
14              for (int i = 0; i <= n; )        {
15                  int sum = result + previous;
16                  previous = result;
17                  result = sum;
18                  i = i + 1;
19              }
20              return result;
21          }
22                  public static long fib(int n) {
23              if (n <= 1) {
24              return n;
25              }
26              else {
27              return fib(n-1) + fib(n-2);
28              }
29          }
30          public static void main(String[] args) {
31                  int num = Integer.parseInt(args[0]);
32              for (int i = 1; i <= num; i++) {
33              System.out.println(i + ": " + fib(i));
34              }
35          }
36      }
```

Εικόνα 4.24: Αρχείο "*RecursionSample.java*" με highlighted τις γραμμές-μεταβλητές της method "fib"

Ακολούθως, μετά την εμφάνιση των αποτελεσμάτων επιστρέφουμε πίσω στο αρχικό menu (εικόνα 4.9). Αν τώρα επιλέγαμε το αρχείο "*ModulusOperatorExample.java*" [Παράρτημα Β-27] και έπειτα (εικόνα 4.10) επιλέγαμε να εξετάσουμε τα "modulus operator" τότε θα παίρναμε τα αποτελέσματα που φαίνονται στην εικόνα 4.25.

```
========================================================================

Line: 8 Vars: i
Line: 9 Vars: d

========================================================================
```

Εικόνα 4.25: Αποτελέσματα επιλογής

Το αντίστοιχο highlighted html αρχείο για τα πιο πάνω αποτελέσματα φαίνεται πιο κάτω, στην εικόνα 4.26.

```
1       package testing.files;
2       public class ModulusOperatorExample {
3       public static void main(String[] args) throws Exception {
4       String str = "Java Modulus Operator(%%) example";
5       System.out.println(str);
6       int i = 50;
7       double d = 32;
8       System.out.println("i mod 10 = " + i % 10);
9       System.out.println("d mod 10 = " + d % 10);
10      }
11      }
```

Εικόνα 4.26: Αρχείο "*ModulusOperatorExample.java*" με highlighted τις γραμμές-μεταβλητές του modulus operator

4.5 Εργαλείο τεμαχισμού προγράμματος

Αφού πραγματοποιηθεί η εργασία του συντακτικού αναλυτή και εντοπιστούν κάποιες γραμμές με τις αντίστοιχες μεταβλητές που επιθυμούμε να εξετάσουμε, σειρά παίρνει τώρα ο τεμαχισμός του συγκεκριμένου προγράμματος με ένα εργαλείο τεμαχισμού κώδικα/προγράμματος (slicing tool).

Από μια έρευνα που έχουμε κάνει, το πιο διαθέσιμο και υποσχόμενο από τα λιγοστά ως καθόλου εργαλεία που υπάρχουν για τεμαχισμό προγραμμάτων σε Java, είναι το JSlice [13]. Το JSlice είναι ένα dynamic slicing tool για προγράμματα Java. Όπως αναφέρεται και στο διαδικτυακό χώρο (webpage) του εργαλείου [13], το JSlice ίσως είναι το μοναδικό *dynamic* slicing tool που υπάρχει διαθέσιμο για προγράμματα σε Java.

Στο JSlice μπορούμε να δώσουμε σαν είσοδο την γραμμή όπου γίνεται αναφορά (reference) της μεταβλητής που μας ενδιαφέρει να δούμε όλες τις αλληλεξαρτήσεις της. Η ιδέα αυτής της μεθοδολογίας όπως έχουμε ήδη αναφέρει είναι να πάρουμε ένα "συρρικνωμένο" κώδικα που θα μας προσφέρει το slicing tool, ώστε να είναι πιο εύκολο να τον μελετήσουμε. Ο συντακτικός αναλυτής θα μας δώσει επομένως κάποιες γραμμές και μεταβλητές που αυτές περικλείονται σε τυχόν χρονοβόρα σημεία του

κώδικα μας. Έπειτα, με την χρήση του slicing tool, χρησιμοποιώντας μια από αυτές τις γραμμές, μπορούμε να πάρουμε ένα μικρότερο κομμάτι (slice) του συγκεκριμένου κώδικα που σίγουρα θα βρίσκεται μέσα στην εμβέλεια κάποιου γενικότερου χρονοβόρου κομματιού (πχ. μιας αναδρομικής μεθόδου, ενός loop). Επομένως, μπορούμε να εξετάσουμε ένα προς ένα αυτά τα slices και να προσπαθήσουμε να εντοπίσουμε κάτι που να χρειάζεται βελτιστοποίηση.

Όπως αναφέρεται και στην ιστοσελίδα του εργαλείου, το JSlice μπορεί να εγκατασταθεί σε Fedora Core 3/4 ή παρόμοια περιβάλλοντα. Επίσης, χρειάζεται gcc version 3.4 για να γίνει compile ο πηγαίος κώδικας (source code) του, JDK 1.4 και Eclipse 3.1. Δυστυχώς, οι προσπάθειες για εγκατάσταση του JSlice σε μια μηχανή που έχει Fedora Core 4, έχουν αποτύχει (σημειώνουμε οτι το Fedora Core, βρίσκεται στην έκδοση 13). Ένας από τους κυριότερους λόγους αυτής της αποτυχίας είναι το γεγονός ότι οι προδιαγραφές του συγκεκριμένου εργαλείου, μας πηγαίνουν πίσω σε εκδόσεις εργαλείων και λειτουργικών συστημάτων που είναι πέραν των 5 χρονών. Ακόμη και η τελευταία έκδοση του JSlice (2008), υποστηρίζει τις ίδιες προδιαγραφές. Το Α και το Ω στην ανάπτυξη ενός λογισμικού, είναι η συντήρηση του. Σαφώς, οι κατασκευαστές του συγκεκριμένου εργαλείου έχουν αποτύχει στον να συντηρήσουν το λογισμικό εργαλείο που έχουν αναπτύξει.

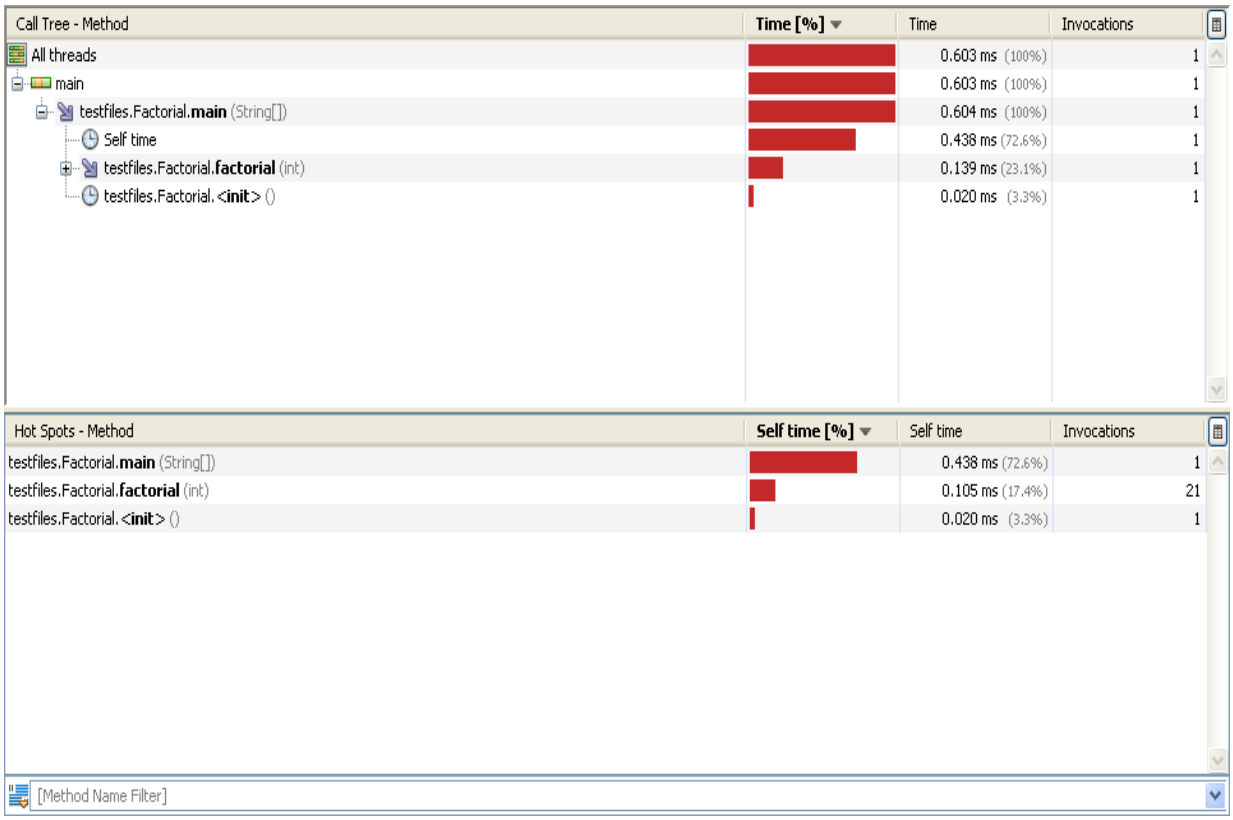
Παρόλα αυτά μπορούμε να συνεχίσουμε την ιδέα αυτής της μεθοδολογίας και χωρίς κάποιο slicing tool, μέχρι στο μέλλον να εμφανιστεί κάποιο που να μπορεί να ενσωματωθεί στις ανάγκες μας. Στην εσχάτη των περιπτώσεων μπορεί για ένα μικρό πρόγραμμα να κάνουμε “manual” (με το χέρι) την εργασία του slicing tool. Όμως, όσο αφορά τις εισηγήσεις για βελτιστοποίηση κώδικα μπορούν να γίνουν και χωρίς το slicing tool, μόνο με την χρήση του συντακτικού αναλυτή. Ο συντακτικός αναλυτής μπορεί να υποδείξει τις γραμμές των πιθανών χρονοβόρων σημείων και επίσης να τις κάνει highlight. Επομένως, μπορούμε να βασιστούμε στον συνδυασμό του συντακτικού αναλυτή και του profiler για να συνεχίσουμε το έργο της μεθοδολογίας μας. Το μόνο πλεονέκτημα που θα υπήρχε αν είχαμε και κάποιο slicing tool, είναι το γεγονός ότι ένα πρόγραμμα/κώδικα θα μπορούσαμε να τον εξετάσουμε από πολύ περισσότερες οπτικές γωνίες, παρά από μια μόνο.

4.6 Πειραματικές μετρήσεις της επίδοσης διαφόρων μικρών προγραμμάτων

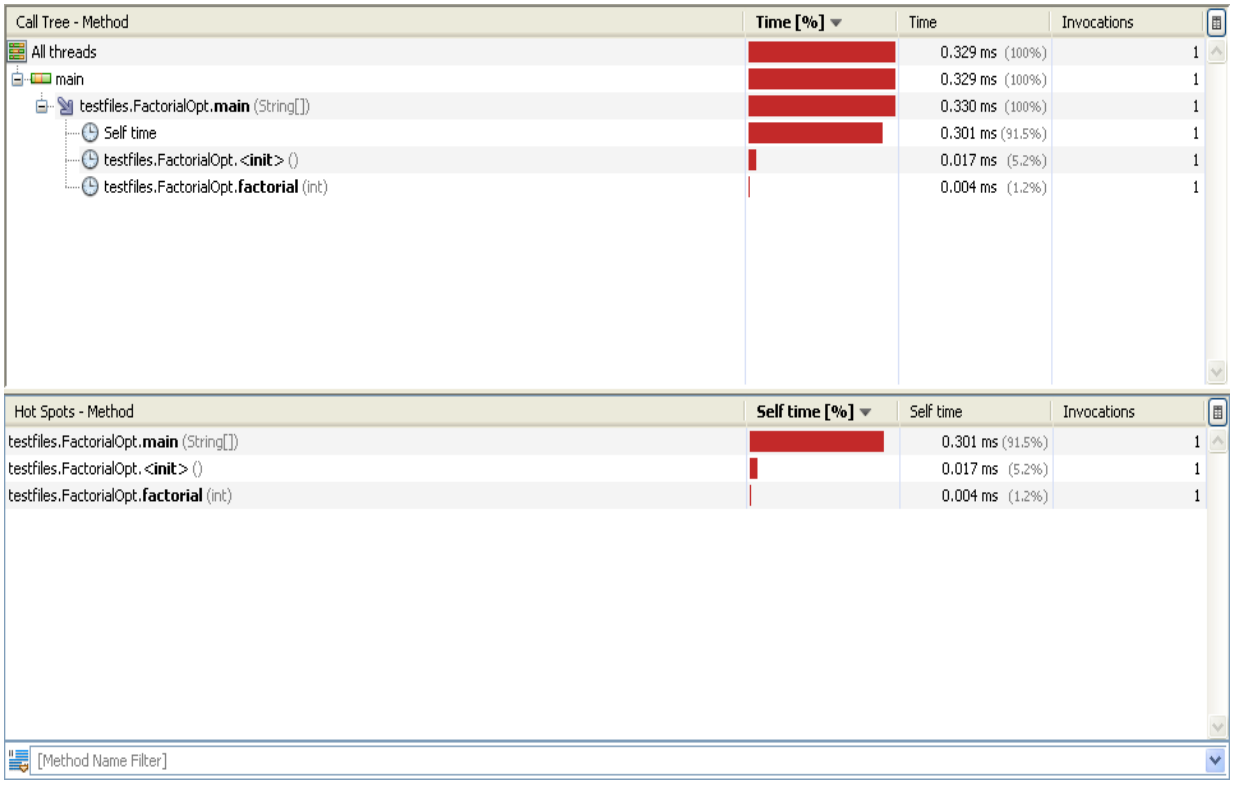
Στο κομμάτι αυτό θα γίνει μια παρουσίαση των διαφόρων πειραματικών μετρήσεων (benchmarks) που έχουν διεξαχθεί με την χρήση του profiler του Netbeans IDE, σε διάφορα μικρά προγράμματα. Στόχος μας είναι να αποδείξουμε και να στηρίξουμε κάποιες από τις σημαντικότερες εισηγήσεις που έχουν αναφερθεί (στο κεφάλαιο 3) για βελτιστοποίηση προγράμματος στην γλώσσα προγραμματισμού Java, καθώς και συγκρίσεις μεταξύ κάποιων κλασσικών αλγορίθμων.

Διαθέτοντας, ζευγάρια αρχείων που επιλύουν το ίδιο πρόβλημα με βέλτιστο και μη-βέλτιστο τρόπο, κάνουμε συγκρίσεις στην επίδοση του κάθε προγράμματος με την βοήθεια του profiler. Να σημειώσουμε ότι για το κάθε ένα πρόγραμμα εκτελούμε την διαδικασία του profiling 10 φορές και παίρνουμε τον μέσο όρο της ολικής του επίδοσης (που φαίνεται στα snapshots που βγάζει ο profiler). Έπειτα, παρατηρούμε ποιο από τα snapshot που μας παρέχει ο profiler, έχει επίδοση κοντά στον μέσο όρο που έχουμε βρει και χρησιμοποιούμε το συγκεκριμένο snapshot για μέτρο σύγκρισης.

Ας ξεκινήσουμε λοιπόν με τη σύγκριση δυο προγραμμάτων που βρίσκουν το παραγοντικό ενός αριθμού. Να σημειώσουμε ότι στα snapshots, στο σημείο που αναγράφεται “Hot Spots -Method”, κάτω από αυτό, απαριθμούνται οι μεθόδοι, με την πιο χρονοβόρα μέθοδο να είναι επικεφαλής και να ακολουθούν οι υπόλοιπες. Το πρώτο πρόγραμμα (Factorial.java) [Παράρτημα Γ-33] κάνει χρήση αναδρομικότητας ενώ το δεύτερο όχι (FactorialOpt.java) [Παράρτημα Γ-34]. (Οι κώδικες όλων το παραδειγμάτων βρίσκονται στο παράρτημα Γ).



Εικόνα 4.27: Profiling αρχείου Factorial.java

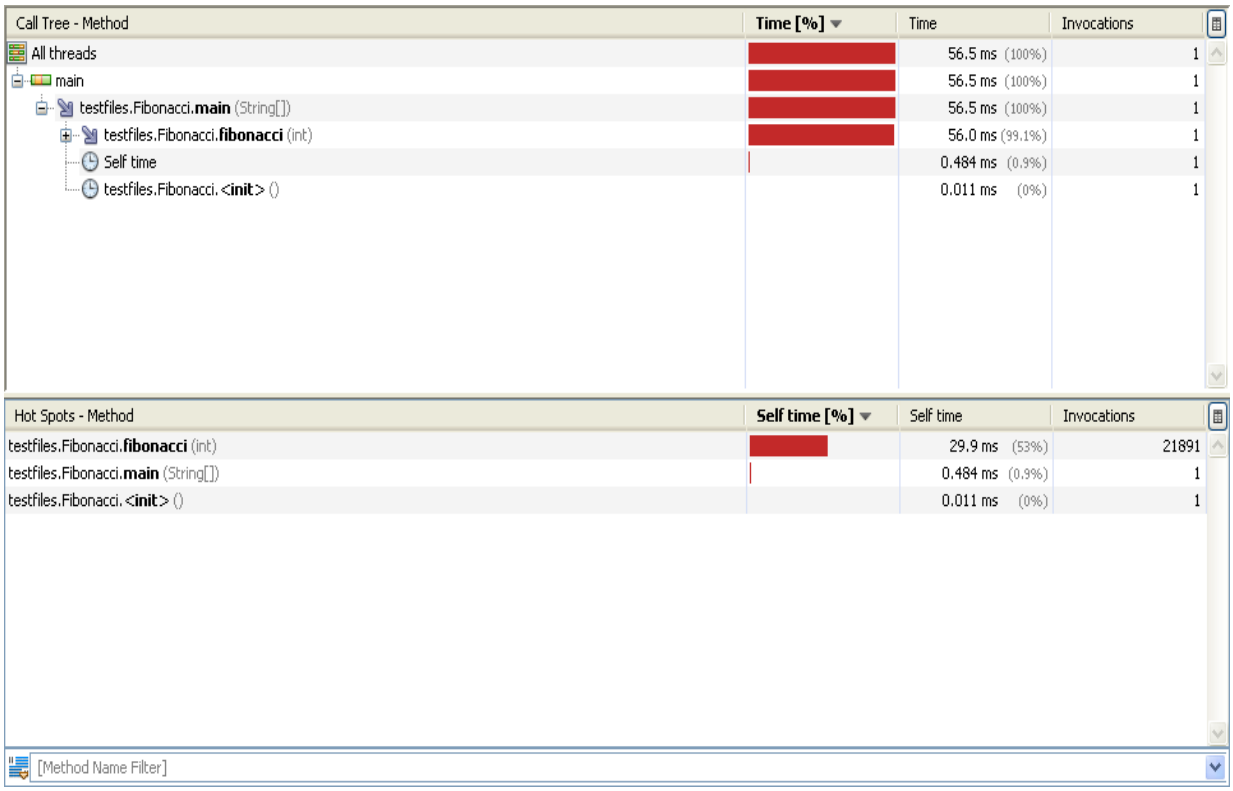


Εικόνα 4.28: Profiling αρχείου FactorialOpt.java

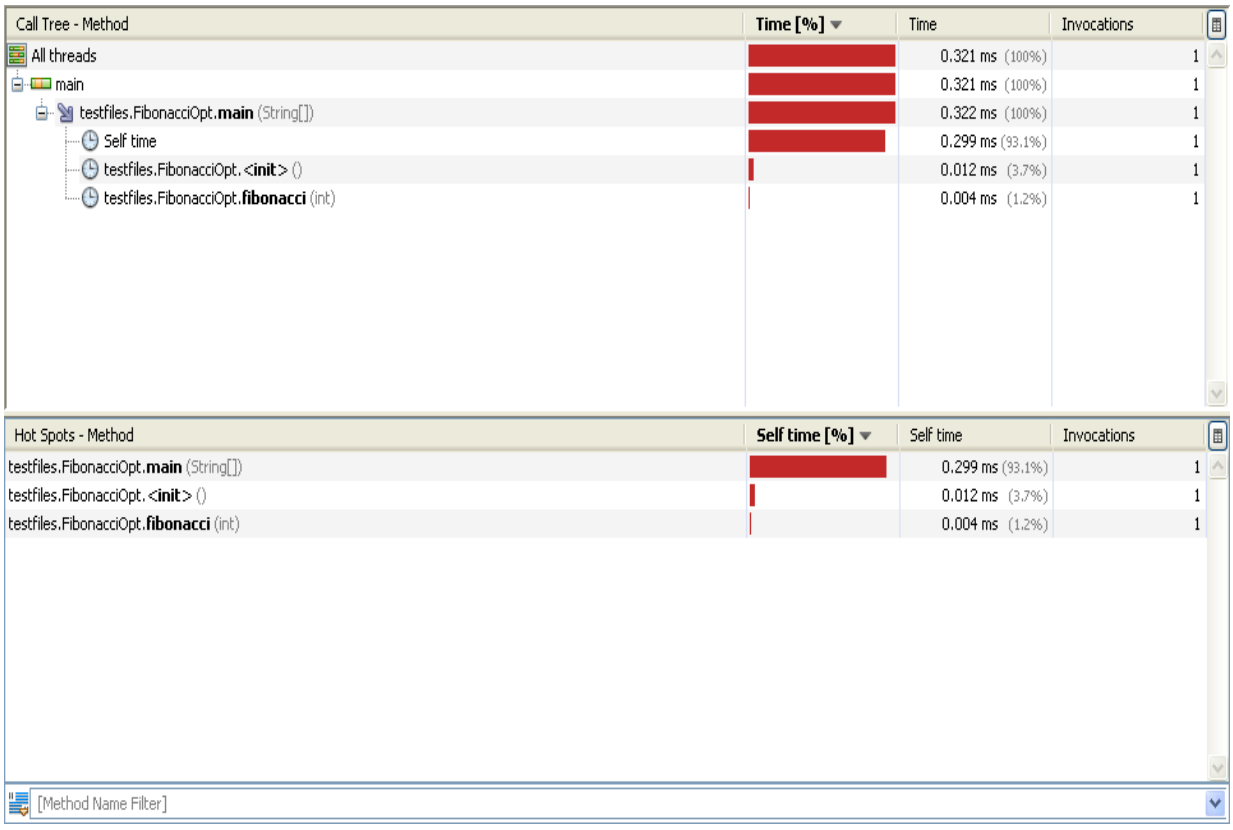
Στο πρόγραμμα Factorial.java γίνεται χρήση της αναδρομικής μεθόδου *factorial* για να βρεθεί το παραγοντικό του αριθμού 20. Στην εικόνα 4.27, παρατηρούμε ότι ο χρόνος εκτέλεσης αυτής της μεθόδου είναι 0.105 ms (milliseconds – χιλιοστά δευτερολέπτου) και γίνονται 21 κλήσεις προς αυτή. Από την άλλη, στο πρόγραμμα FactorialOpt.java γίνεται ο υπολογισμός του ιδίου αριθμού με την χρήση μιας μη-αναδρομικής μεθόδου (μέθοδος *factorial*, εικόνα 4.28). Στην εικόνα 4.28, σε αντίθεση με το προηγούμενο snapshot του profiler, γίνεται μόνο μια φορά κλήση της μεθόδου *factorial* και έχουμε χρόνο 0.004 ms. Ο συνολικός χρόνος επίδοσης (all threads) του Factorial.java είναι 0.603 ms και του Factorial Opt.java είναι 0.329 ms. Έτσι, παρατηρούμε ότι η χρήση αναδρομικότητας κάνει το συγκεκριμένο πρόγραμμα πιο αργό, αν και δεν είναι τόσο αισθητή η διαφορά.

Στο συγκεκριμένο παράδειγμα, βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον συντακτικό αναλυτή στο αρχείο Factorial.java και θα ανακαλύπταμε ότι κάνει χρήση αναδρομικής μεθόδου. Έπειτα αφού δεν έχουμε χρησιμοποιήσει κάποιο slicing tool, θα παρατηρούσαμε στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της αναδρομικής μεθόδου. Στη συνέχεια, θα σκεφτόμασταν τρόπους μετατροπής της αναδρομικής μεθόδου σε μη-αναδρομική. Στην συγκεκριμένη περίπτωση, για το γνωστό πρόβλημα του factorial, θα κάναμε την αντικατάσταση με μια μη-αναδρομική μέθοδο όπως αυτήν που βλέπουμε στο αρχείο FactorialOpt.java.

Τώρα θα δούμε ένα άλλο πρόβλημα που κάνει χρήση αναδρομικότητας, που σε αντίθεση με το πιο πάνω πρόβλημα, η διαφορά μεταξύ αναδρομικής και μη-αναδρομικής επίλυσης του συγκεκριμένου προβλήματος είναι αρκετά αισθητή. Το πρόβλημα είναι η εύρεση του 20^{ου} αριθμού Fibonacci. Το πρόγραμμα που κάνει χρήση της αναδρομικής μεθόδου είναι το Fibonacci.java [Παράρτημα Γ-34] και αυτό που κάνει χρήση της μη-αναδρομικής είναι το FibonacciOpt.java [Παράρτημα Γ-35].



Εικόνα 4.29: Profiling αρχείου Fibonacci.java



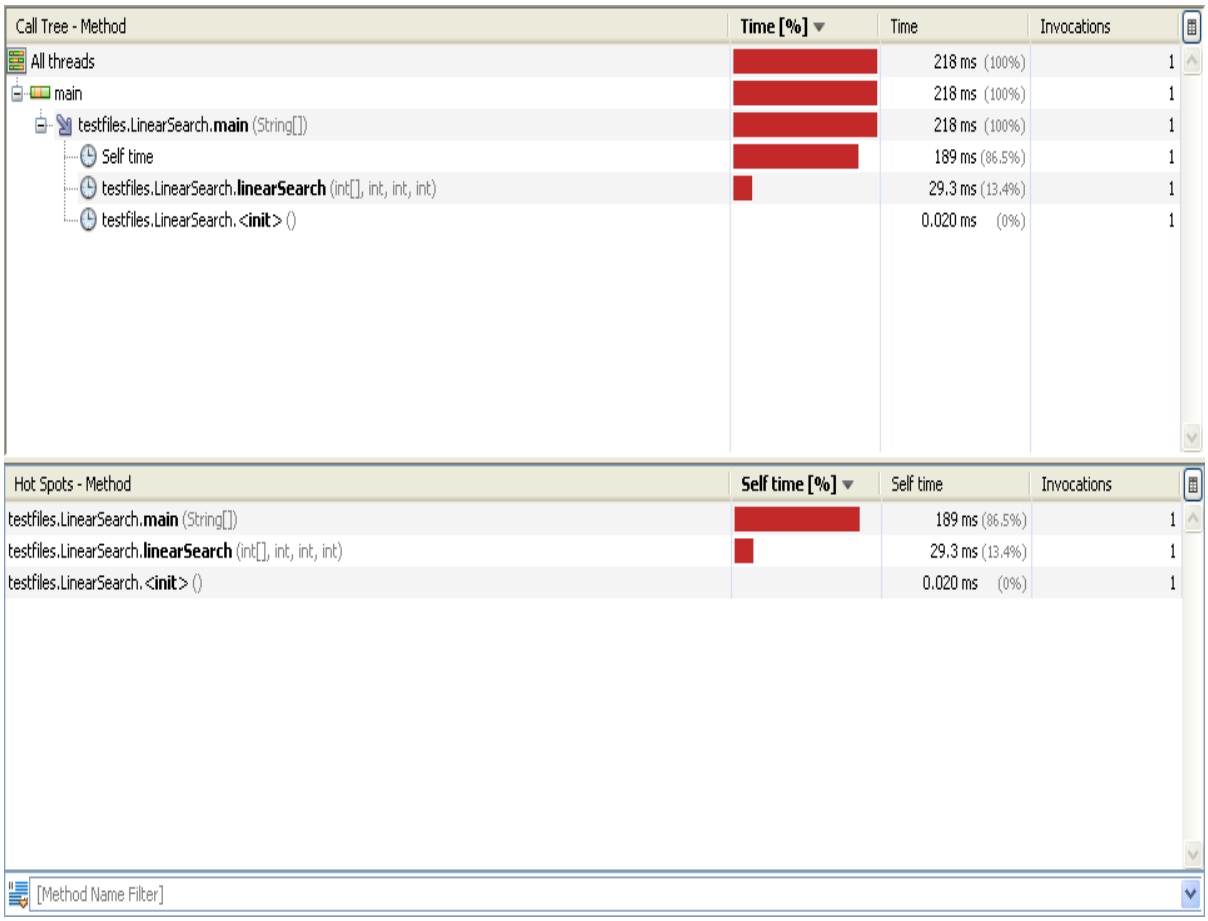
Εικόνα 4.30: Profiling αρχείου FibonacciOpt.java

Στην εικόνα 4.29 παρατηρούμε ότι ο χρόνος εκτέλεσης της αναδρομικής μεθόδου (μέθοδος fibonacci) που υπολογίζει τον $20^{ο}$ αριθμό fibonacci είναι 29.9 ms και γίνονται 21891 κλήσεις της συγκεκριμένης μεθόδου. Από την άλλη, στην εικόνα 4.30, παρατηρούμε ότι ο χρόνος της μη-αναδρομικής μεθόδου (μέθοδος fibonacci) είναι 0.004 ms και γίνεται μόνο μια φορά κλήση της συγκεκριμένης μεθόδου. Ο συνολικός χρόνος επίδοσης (all threads) του Fibonacci.java είναι 56.5 ms και του FibonacciOpt.java είναι 0.321 ms. Επομένως, παρατηρούμε ότι η χρήση αναδρομικότητας εδώ κάνει πιο αισθητή την παρουσία της, αφού έχουμε μεγαλύτερη διαφορά στο χρόνο. Επομένως, σε τέτοιες περιπτώσεις η αποφυγή της αναδρομικότητας μπορεί να μας δώσει ένα πιο βέλτιστο πρόγραμμα. Αυτό, έρχεται σε αντίφαση με το γεγονός ότι η χρήση της αναδρομικότητας στις δυο πιο πάνω περιπτώσεις (factorial και fibonacci), δίνει ένα πιο φυσικό και ευκολονόητο τρόπο σκέψης από αυτό που δίνει η χρήση μη-αναδρομικότητας.

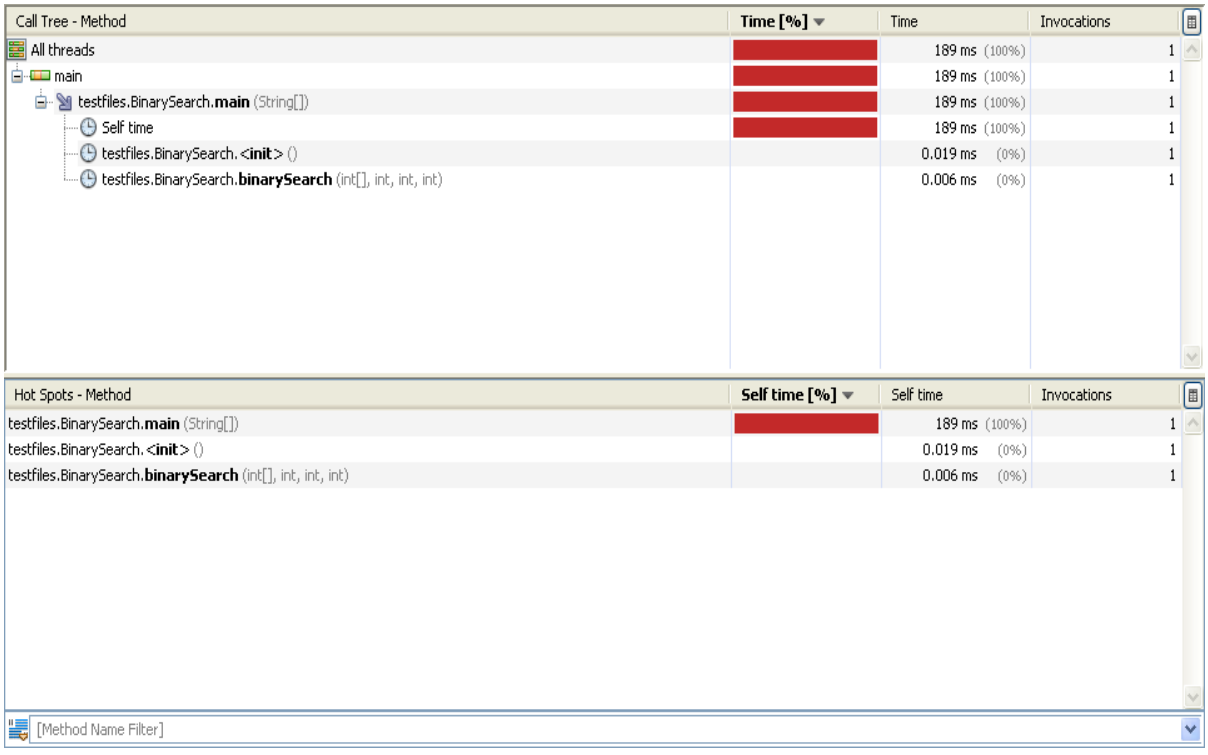
Στο πιο πάνω παράδειγμα, βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον συντακτικό αναλυτή στο αρχείο Fibonacci.java και θα ανακαλύπταμε ότι χρησιμοποιεί μια αναδρομική μέθοδο. Έπειτα θα παρατηρούσαμε στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της αναδρομικής μεθόδου. Στη συνέχεια, θα σκεφτόμασταν τρόπους μετατροπής της αναδρομικής μεθόδου σε μη-αναδρομική. Στην συγκεκριμένη περίπτωση, για το γνωστό πρόβλημα του fibonacci, θα κάναμε την αντικατάσταση με μια μη-αναδρομική μέθοδο όπως αυτήν που βλέπουμε στο αρχείο FibonacciOpt.java.

Ας δούμε τώρα τι διαφορά μπορεί να κάνει σε ένα πρόγραμμα η χρήση διαφορετικών αλγορίθμων. Θα δούμε συγκεκριμένα δύο αλγόριθμους αναζήτησης και δύο αλγόριθμους ταξινόμησης.

Το πρόγραμμα LinearSearch.java [Παράρτημα Γ-37] κάνει χρήση γραμμικής αναζήτησης [12] και το πρόγραμμα BinarySearch.java [Παράρτημα Γ-31] κάνει χρήση δυαδικής αναζήτησης [12]. Τα δύο αυτά προγράμματα, κάνοντας χρήση των δυο διαφορετικών αλγορίθμων αναζήτησης, καλούνται να βρουν την θέση του αριθμού 5000000, μέσα σε ένα πίνακα στοιχείων που απαριθμούνται από το 0 ως το 9999999.



Εικόνα 4.31: Profiling αρχείου LinearSearch.java



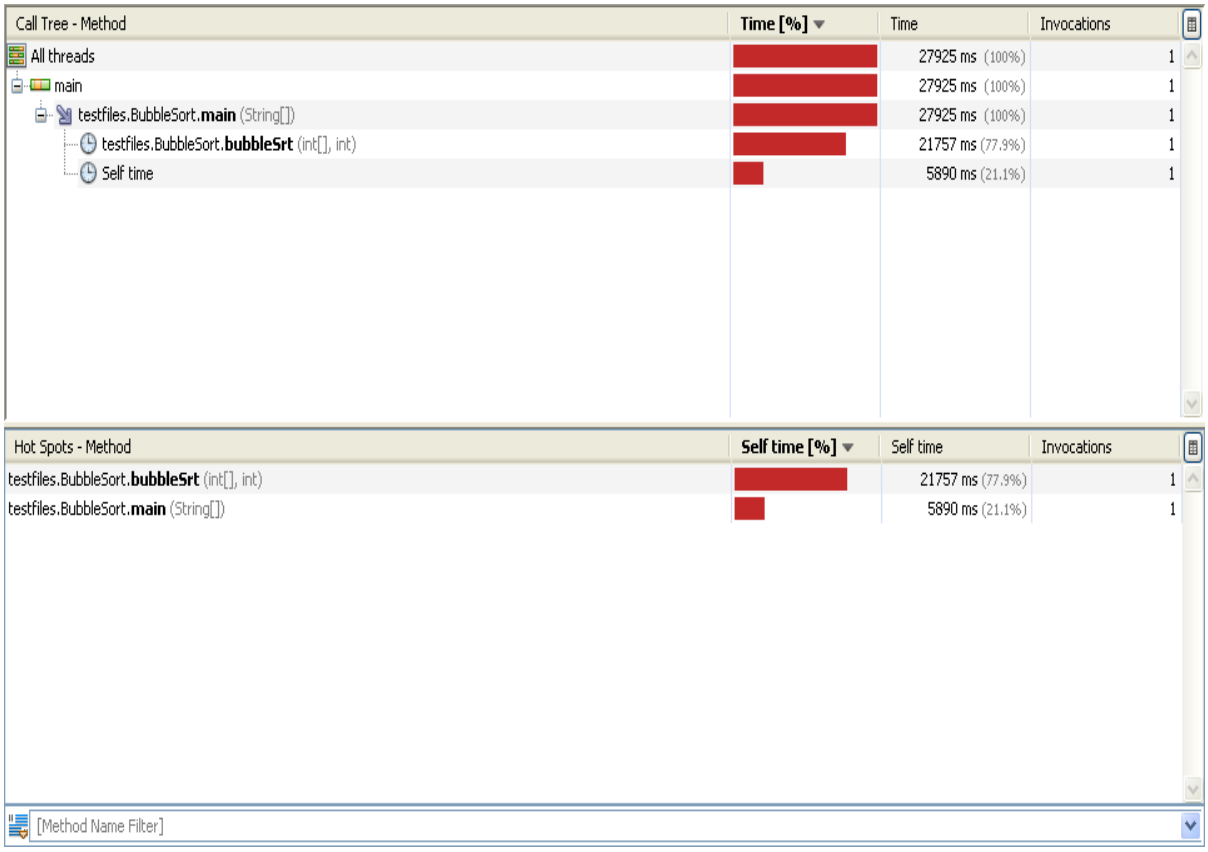
Εικόνα 4.32: Profiling αρχείου BinarySearch.java

Στην εικόνα 4.31 βλέπουμε τους χρόνους εκτέλεσης των μεθόδων του LinearSearch.java και στην εικόνα 4.32, τους χρόνους εκτέλεσης των μεθόδων του BinarySearch.java. Παρατηρούμε ότι ο χρόνος εκτέλεσης της μεθόδου που χρησιμοποιεί γραμμική αναζήτηση (μέθοδος linearSearch) είναι 29.3 ms. Από την άλλη, στην εικόνα 4.32, ο χρόνος της μεθόδου που χρησιμοποιεί δυαδική αναζήτηση (μέθοδος binarySearch) είναι 0.006 ms. Επομένως, η χρήση δυαδικής αναζήτησης σε τέτοιου είδους προβλήματα είναι καλύτερη, αν και η διαφορά δεν είναι και τεράστια.

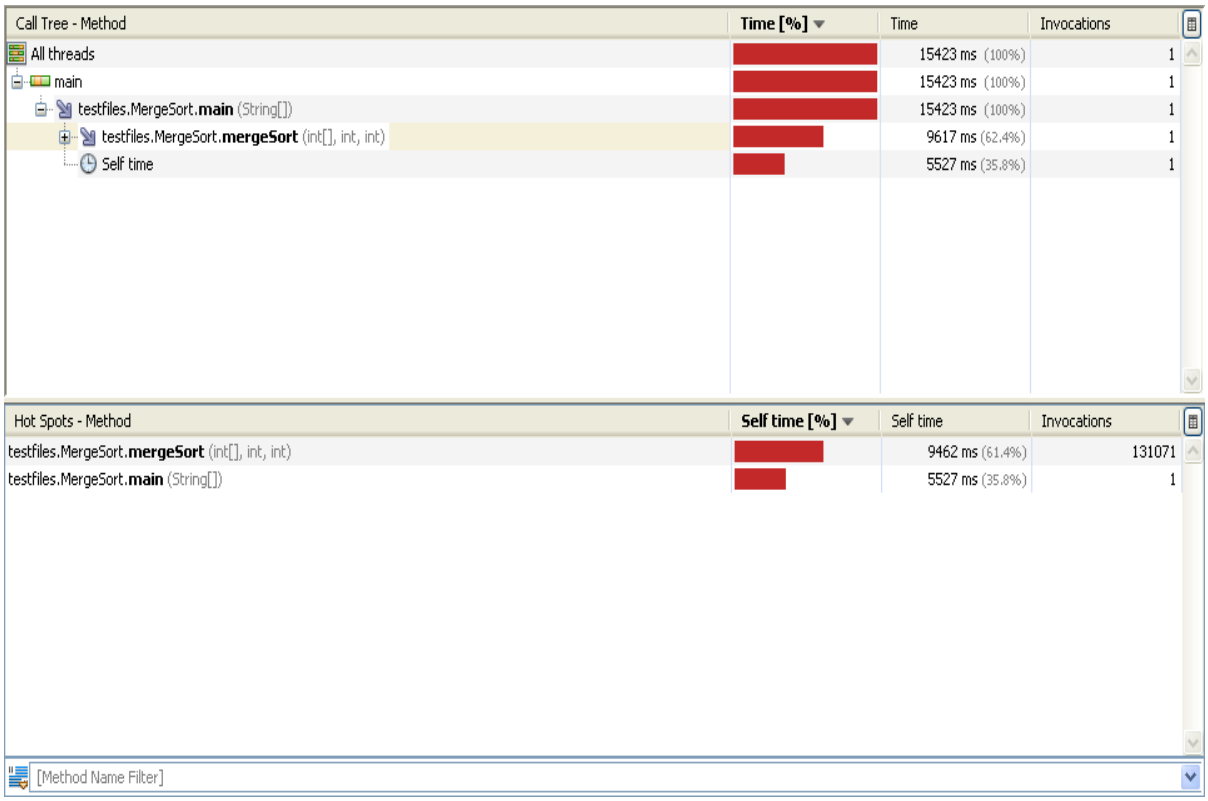
Βάσει της μεθοδολογίας μας, αρχικά θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου LinearSearch.java. Ο profiler θα μας έδινε την main σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.31) και σαν δεύτερη πιο χρονοβόρα μέθοδο, την linearSearch. Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια της μεθόδου main. Θα συμπεραίναμε ότι δεν υπάρχει κάτι που να μπορούμε να βελτιστοποιήσουμε, όμως θα προσέχαμε πως η main καλεί την μέθοδο linearSearch, που όπως έχουμε δει είναι η δεύτερη πιο χρονοβόρα. Επομένως, θα χρησιμοποιούσαμε τον συντακτικό αναλυτή για να εντοπίσουμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια της linearSearch. Στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε ότι η συγκεκριμένη μέθοδος κάνει χρήση ενός “for” loop. Μελετώντας το “for” loop και έχοντας αλγοριθμικές γνώσεις θα παρατηρούσαμε ότι γίνεται μια σειριακή αναζήτηση. Οπόταν, θα σκεφτόμασταν να αντικαταστήσουμε την σειριακή αναζήτηση με μια δυαδική αναζήτηση που είναι πιο γρήγορη. Έτσι θα αντικαθιστούσαμε την συγκεκριμένη μέθοδο, με την μέθοδο binarySearch που βλέπουμε στο αρχείο BinarySearch.java.

Παρόλα αυτά, η χρήση διαφορετικών αλγορίθμων ταξινόμησης σε ένα πρόβλημα, μπορεί να μας δώσει μεγαλύτερη διαφορά στην επίδοση (σε σύγκριση με τους αλγόριθμους αναζήτησης). Ας δούμε λοιπόν την διαφορά μεταξύ του αλγορίθμου *bubblesort* (ταξινόμηση φυσαλίδας) [12] και του *mergesort* (ταξινόμηση συγχώνευσης) [12]. Το πρόγραμμα BubbleSort.java [Παράρτημα Γ-32] κάνει ταξινόμηση *bubblesort* και το πρόγραμμα MergeSort.java [Παράρτημα Γ-39], κάνει ταξινόμηση *mergesort*. Τα δύο αυτά προγράμματα, κάνοντας χρήση των δυο διαφορετικών αλγορίθμων

ταξινόμησης, καλούνται να ταξινομήσουν σε αύξουσα σειρά ένα πίνακα από αριθμούς από το 65536 μέχρι το 1. Δηλαδή, μετά το τέλος της ταξινόμησης οι αριθμοί θα βρίσκονται αποθηκευμένοι στον πίνακα από το 1 μέχρι 65536.



Εικόνα 4.33: Profiling αρχείου BubbleSort.java



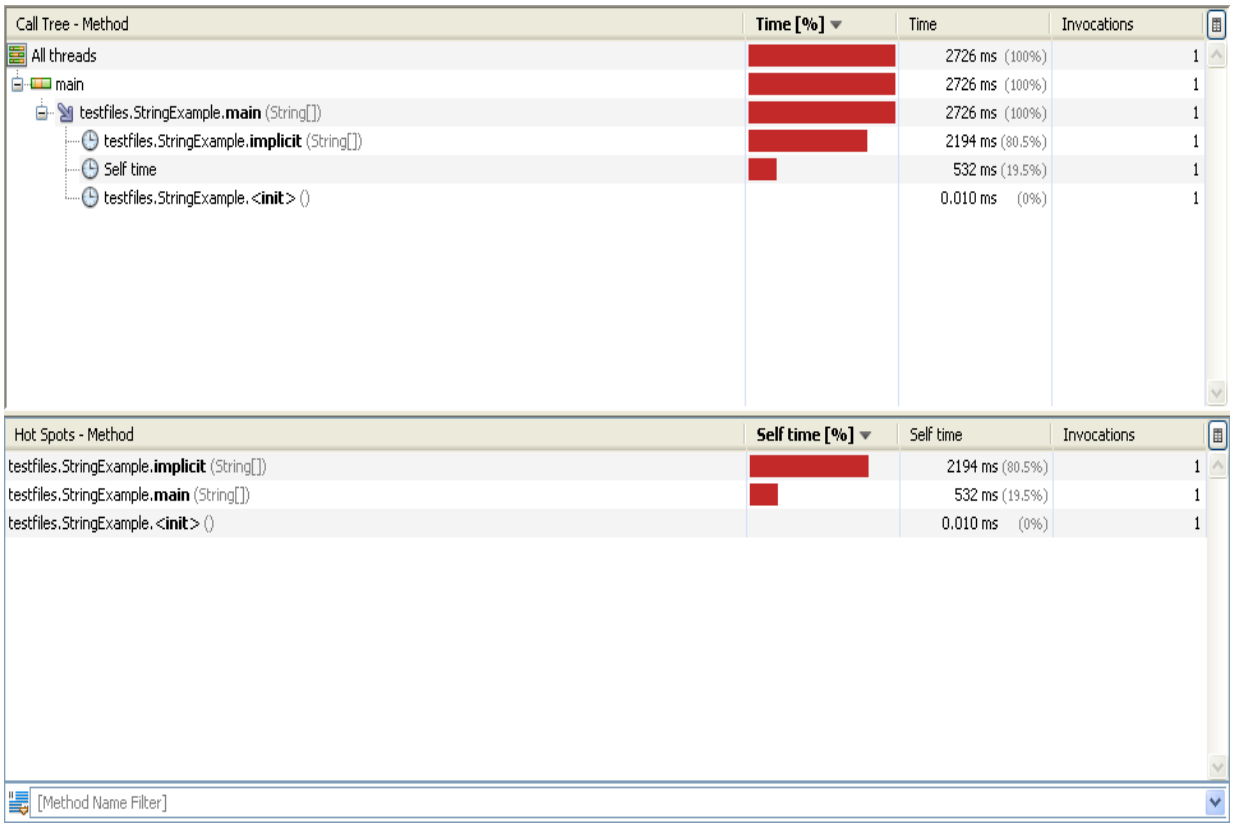
Εικόνα 4.34: Profiling αρχείου MergeSort.java

Στην εικόνα 4.33 μπορούμε να δούμε τους χρόνους εκτέλεσης της μεθόδου του BubbleSort.java και στην εικόνα 4.34, τους χρόνους εκτέλεσης της μεθόδου του MergeSort.java. Παρατηρούμε ότι ο χρόνος εκτέλεσης της μεθόδου που χρησιμοποιεί bubblesort (μέθοδος *bubbleSrt*) είναι 21757 ms. Από την άλλη, στην εικόνα 4.34, ο χρόνος της μεθόδου που κάνει mergesort (μέθοδος *mergeSort*) είναι 9462 ms. Μπορούμε, επίσης να παρατηρήσουμε ότι το MergeSort.java που χρησιμοποιεί αναδρομικότητα, κάνει 131071 κλήσεις της μεθόδου mergeSort ενώ το BubbleSort.java κάνει μόνο μία φορά κλήση στην *bubbleSrt*. Συνεπώς, η χρήση αναδρομικότητας με έξυπνο τρόπο, σε μερικές περιπτώσεις, μπορεί να χρησιμοποιηθεί υπέρ μας. Συμπεραίνουμε λοιπόν, πως η χρήση του mergesort σε τέτοιου είδους προβλήματα ταξινόμησης είναι σαφώς καλύτερη από ότι η χρήση του bubblesort, έχοντας μάλιστα σημαντική διάφορα.

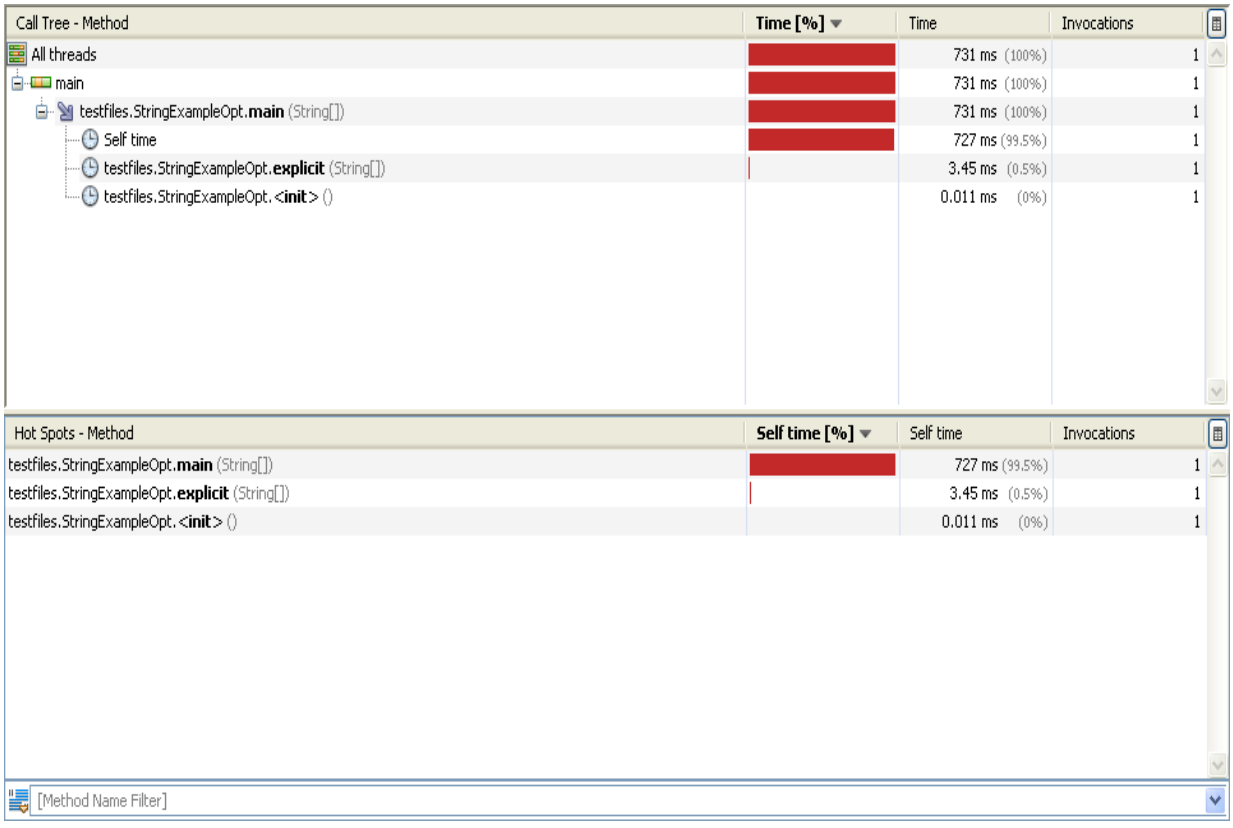
Βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου BubbleSort.java. Ο profiler θα μας έδινε την bubbleSrt σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.33). Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις

αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε ότι η συγκεκριμένη μέθοδος κάνει χρήση φωλιασμένων "for" loops. Μελετώντας τα loops μαζί με αλγοριθμικές γνώσεις που διαθέτουμε, θα παρατηρούσαμε ότι γίνεται χρήση ενός πολύ γνωστού αλγόριθμου για ταξινόμηση, αυτού του bubblesort. Οπόταν, θα σκεφτόμασταν να αντικαταστήσουμε τον συγκεκριμένο αλγόριθμο με κάποιο άλλο που γνωρίζουμε ότι είναι αρκετά πιο γρήγορος. Έτσι, θα αντικαθιστούσαμε την συγκεκριμένη μέθοδο, με την μέθοδο mergeSort (που χρησιμοποιεί τον αλγόριθμο του mergesort που είναι πιο βέλτιστος από ότι αυτός του bubblesort) που βλέπουμε στο αρχείο MergeSort.java.

Ας συνεχίσουμε τώρα να δούμε τις διαφορές μεταξύ της χρήσης του τελεστή "+", για συνένωση string στην Java, έναντι του StringBuilder. Όπως έχουμε αναφέρει και στο κεφάλαιο 3, αν θέλουμε να κάνουμε την συνένωση ενός string μέσα σε ένα loop, τότε ο καλύτερος τρόπος για να το κάνουμε αυτό είναι με την χρήση του StringBuilder. Το πρόγραμμα StringExample.java [Παράρτημα Γ-40], χρησιμοποιεί μέσα στο σώμα ενός "for" loop, τον τελεστή "+" για να κάνει συνολικά 3000 συνενώσεις ενός string. Αντίστοιχα το προγραμμα StringExampleOpt.java [Παράρτημα Γ-40] κάνει τις ίδιες συνενώσεις αλλά με την χρήση του StringBuilder. Πιο κάτω βλέπουμε τα snapshots των δυο προγραμμάτων.



Εικόνα 4.35: Profiling αρχείου StringExample.java



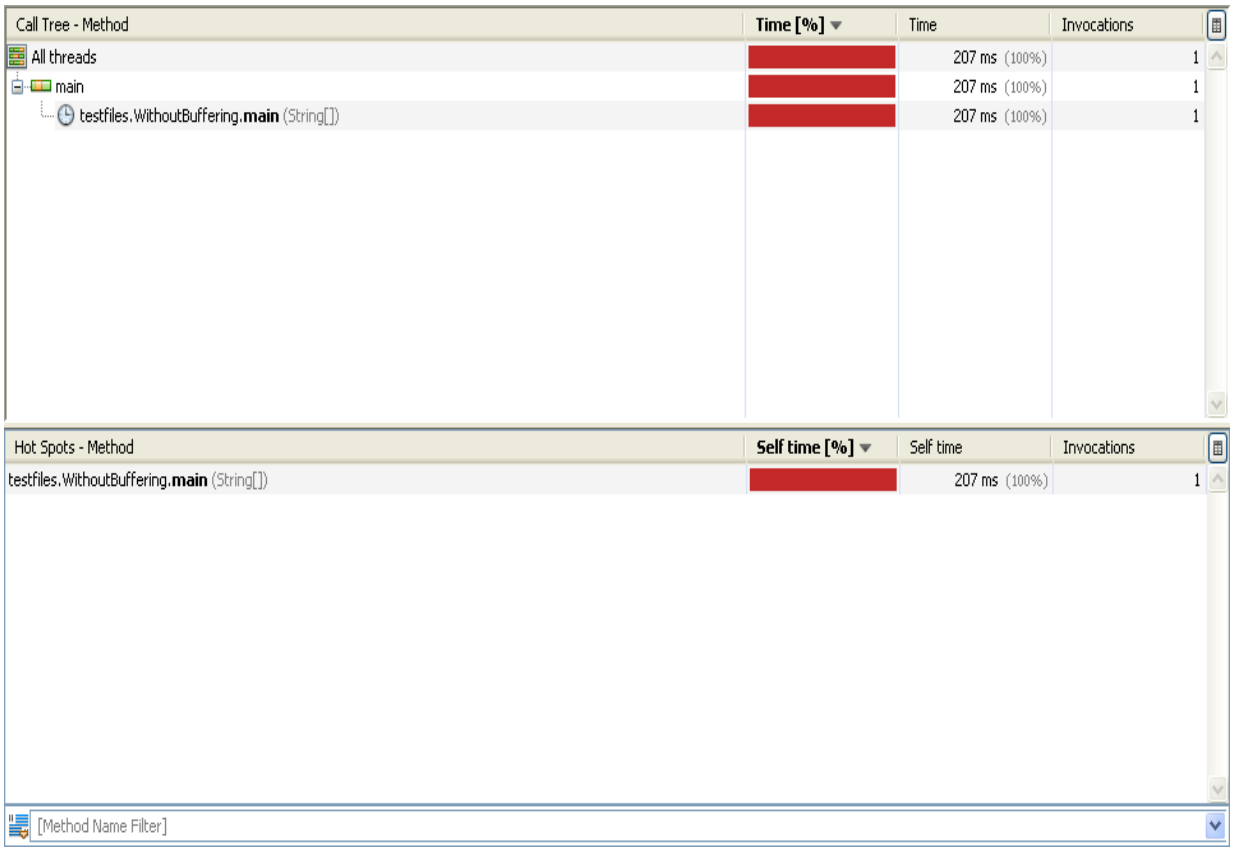
Εικόνα 4.36: Profiling αρχείου StringExampleOpt.java

Στην εικόνα 4.35 η μέθοδος *implicit* που χρησιμοποιεί τον τελεστή "+" ως μέσο για να συνενώσει τα string, χρειάζεται χρόνο 2194 ms. Από την άλλη, στην εικόνα 4.36, η μέθοδος explicit, που χρησιμοποιεί τον StringBuilder, χρειάζεται μόνο 3.45 ms. Επομένως, η χρήση του StringBuilder, είναι σαφώς ο πιο βέλτιστος τρόπος όσο αφορά την συνένωση string, ειδικά όταν πρόκειται για συνένωση μέσα στο σώμα ενός loop.

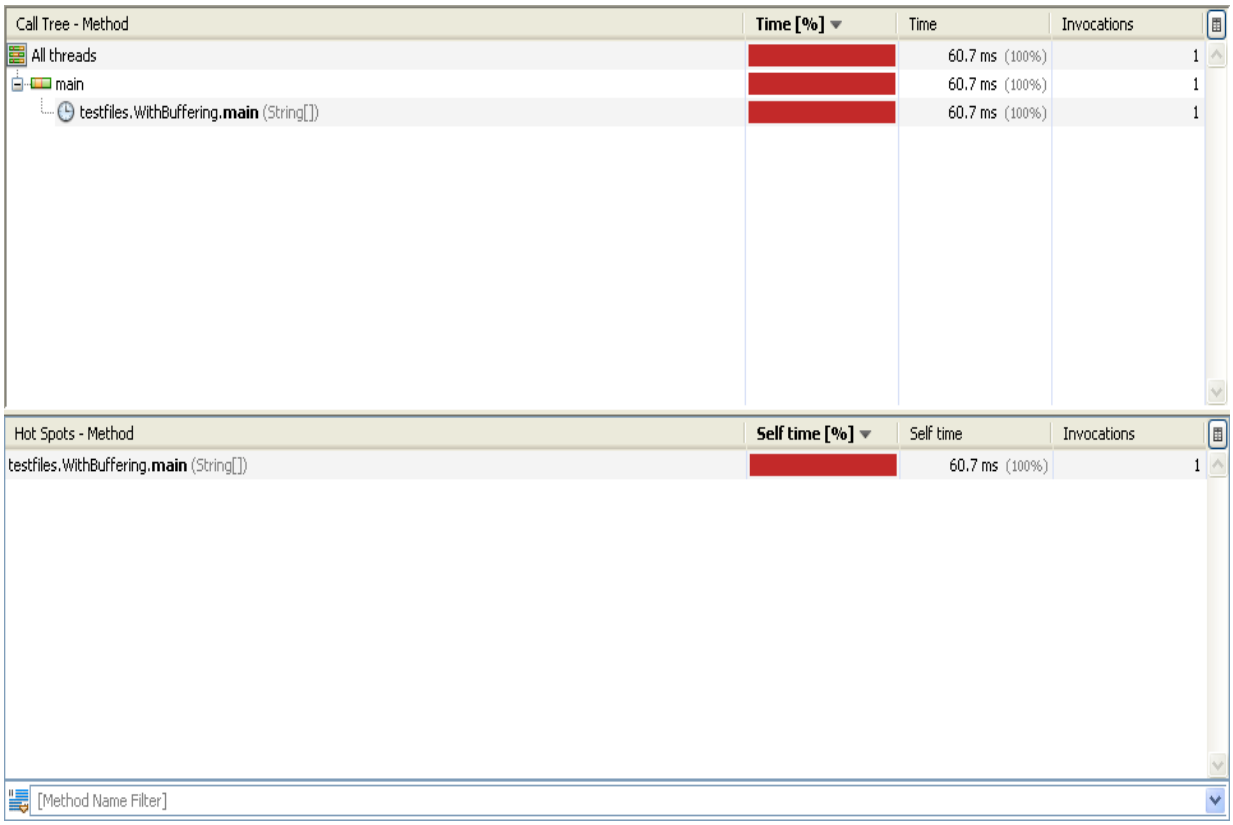
Στο συγκεκριμένο παράδειγμα, βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου StringExample.java. Ο profiler θα μας έδινε την implicit σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.35). Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε ότι η συγκεκριμένη μέθοδος κάνει χρήση του τελεστή "+" μέσα σε ένα "for" loop. Επομένως, θα σκεφτόμασταν να αντικαταστήσουμε τον συγκεκριμένο τελεστή με ένα αντικείμενο τύπου StringBuilder για γίνει το πρόγραμμα μας αρκετά πιο βέλτιστο. Έτσι, θα αντικαθιστούσαμε την συγκεκριμένη μέθοδο, με την μέθοδο explicit που βλέπουμε στο αρχείο StringExampleOpt.java.

Στο κεφάλαιο 3, είχαμε αναφέρει πως η χρήση του *buffering* είναι ο καλύτερος τρόπος με τον οποίο μπορεί να γίνει βελτιστοποίηση της επίδοσης των εισόδων και εξόδων (input/output) στην Java. Η Java μας παρέχει την δυνατότητα να τυλίγουμε (wrapping) το ένα αντικείμενο μέσα στο άλλο. Αυτό μπορεί να μας δώσει καλύτερη επίδοση ειδικά όταν εφαρμοστεί σε περιπτώσεις αξιοποίησης των εισόδων/εξόδων. Για παράδειγμα ένα αντικείμενο FileReader [7] μπορούμε να το κάνουμε "wrapped" μέσα σε ένα αντικείμενο BufferedReader [7].

Το πρόγραμμα WithoutBuffering.java [Παράρτημα Γ-42] διαβάζει όλους τους χαρακτήρες από ένα αρχείο κειμένου *txt* (20000 γραμμών) χρησιμοποιώντας ένα αντικείμενο FileReader. Το πρόγραμμα WithBuffering.java [Παράρτημα Γ-41] διαβάζει όλους τους χαρακτήρες από το ίδιο αρχείο, κάνοντας χρήση ενός BufferedReader, έχοντας κάνει "wrapped" μέσα στον BufferedReader ένα αντικείμενο FileReader.



Εικόνα 4.37: Profiling αρχείου WihoutBuffering.java

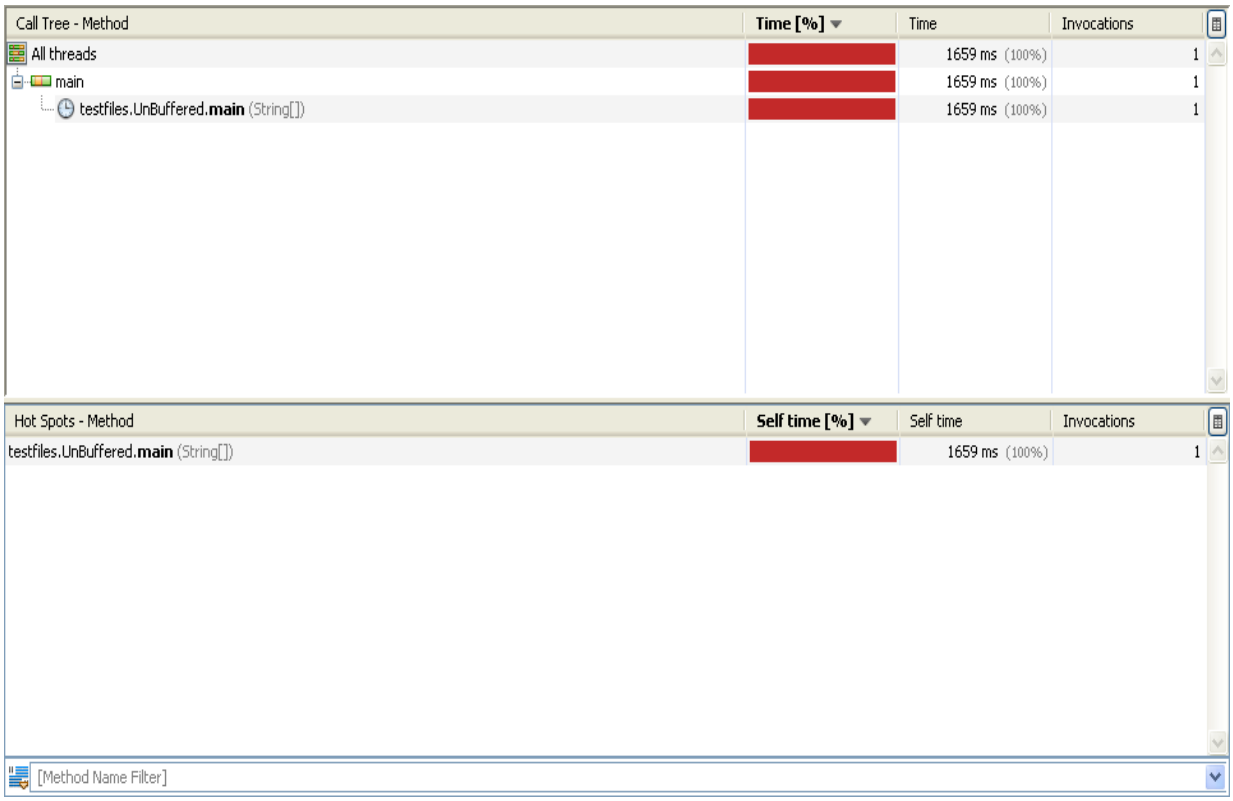


Εικόνα 4.38: Profiling αρχείου WihBuffering.java

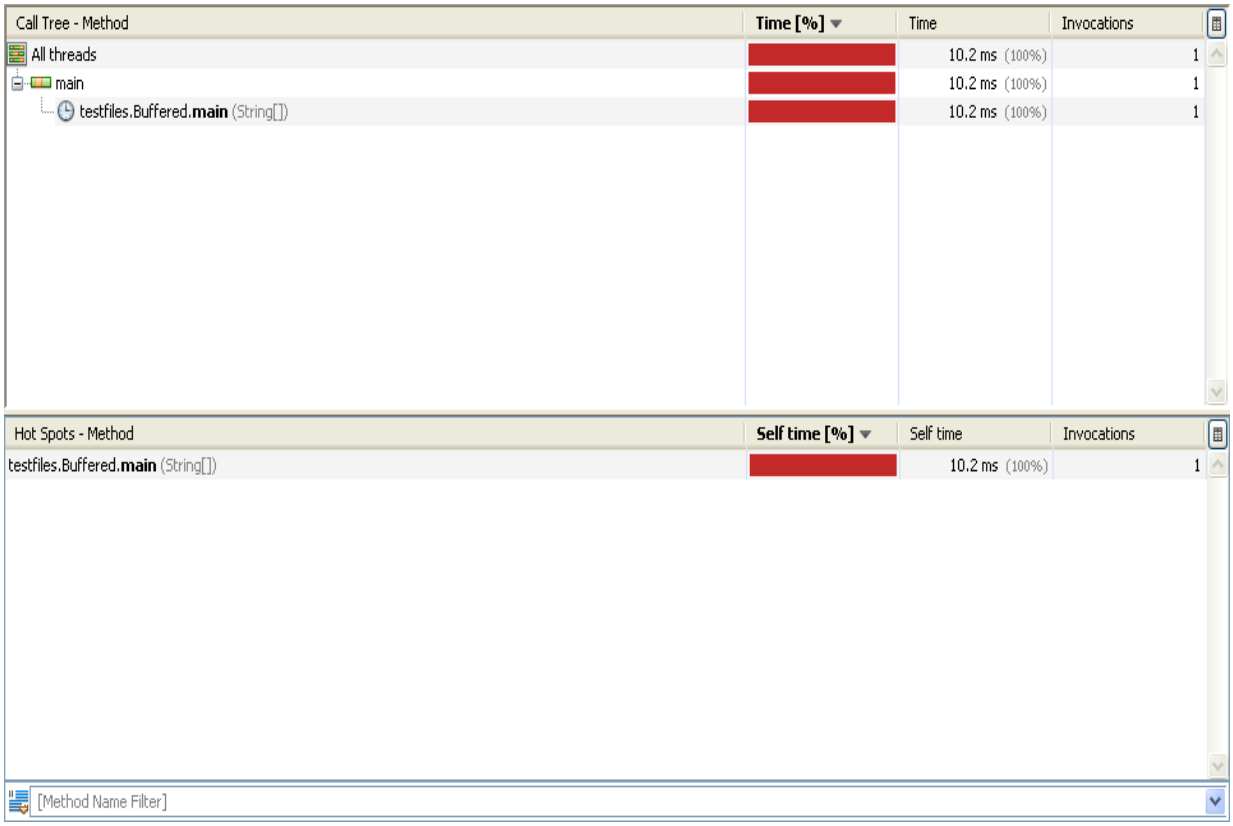
Στην εικόνα 4.37 παρατηρούμε ότι ο ολικός χρόνος εκτέλεσης του προγράμματος WithoutBuffering.java, που δεν κάνει χρήση *buffering*, είναι 207 ms. Στην εικόνα 4.38 παρατηρούμε ότι το πρόγραμμα WithBuffering.java, το οποίο κάνει χρήση *buffering*, ο ολικός χρόνος εκτέλεσης του είναι 60.7 ms. Συνεπώς, η χρήση του buffering μπορεί και αυτή με την σειρά της να μας βοηθήσει στην βελτιστοποίηση προγραμμάτων Java.

Στο πιο πάνω παράδειγμα, βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου WithoutBuffering.java. Ο profiler θα μας έδινε την main σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.37). Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε από τις γραμμές με τις αντίστοιχες μεταβλητές, ότι διαβάζονται οι χαρακτήρες από ένα αρχείο. Οι χαρακτήρες αυτοί διαβάζονται χρησιμοποιώντας ένα αντικείμενο τύπου FileReader. Όμως, δεν γίνεται "wrapped" ο FileReader μέσα σε ένα αντικείμενο τύπου BufferedReader, έτσι ώστε να χρησιμοποιείται το buffering και να γίνει πιο βέλτιστο το συγκεκριμένο πρόγραμμα. Επομένως, κάνουμε αυτήν την αλλαγή και παίρνουμε το αρχείο WithBuffering.java.

Ας δούμε ακόμα ένα παράδειγμα όπου η χρήση του *buffering* κάνει πιο αισθητή διαφορά στον χρόνο εκτέλεσης ενός προγράμματος. Το πρόγραμμα UnBuffered.java [Παράρτημα Γ-41] μετρά την εμφάνιση του χαρακτήρα "x" στο αρχείο κειμένου txt που χρησιμοποιήθηκε στα δυο πιο πάνω προγράμματα. Αυτό γίνεται διαβάζοντας χαρακτήρα προς χαρακτήρα το αρχείο. Το πρόγραμμα Buffered.java [Παράρτημα Γ-33], μετρά και αυτό την εμφάνιση του χαρακτήρα "x" με την μόνη διαφορά, ότι χωρίζει σε μερίδια των 1024 χαρακτήρων (bytes) κάθε φορά το αρχείο. Αυτό επιτυγχάνεται με την αποθήκευση κάθε φορά 1024 χαρακτήρων σε ένα πίνακα από bytes [7], όπου ακολούθως διαβάζεται ο πίνακας αντί το αρχείο, για εύρεση εμφάνισης του συγκεκριμένου χαρακτήρα. Πιο κάτω βλέπουμε τα snapshots των δυο προγραμμάτων.



Εικόνα 4.39: Profiling αρχείου UnBuffered.java



Εικόνα 4.40: Profiling αρχείου Buffered.java

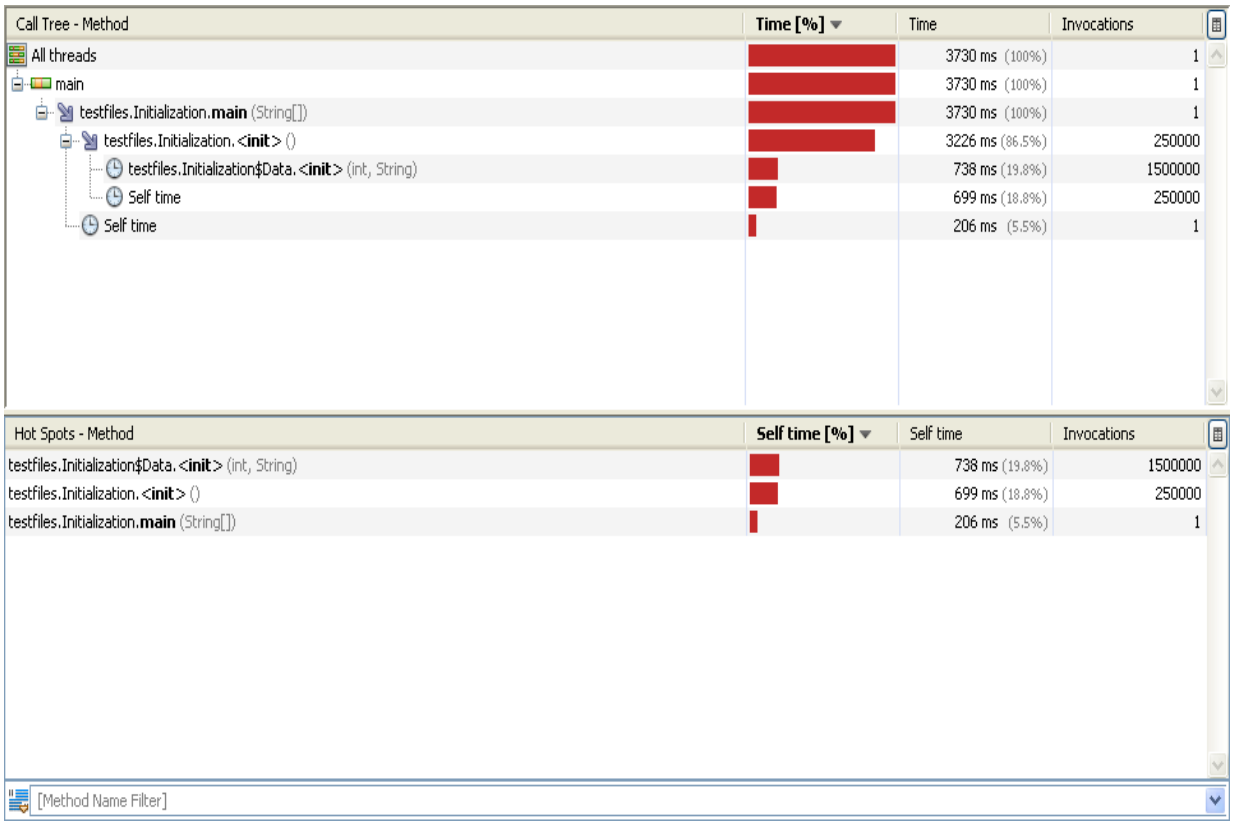
Στην εικόνα 4.39 παρατηρούμε ότι ο ολικός χρόνος εκτέλεσης του προγράμματος UnBuffered.java είναι 1659 ms. Στην εικόνα 4.40, ο αντίστοιχος ολικός χρόνος εκτέλεσης του προγράμματος Buffered.java, είναι 10.2 ms. Συνεπώς, η χρήση του buffering σε αυτού του είδους το πρόβλημα, μας έχει δώσει μεγαλύτερη διαφορά στους χρόνους εκτέλεσης και είναι πιο εμφανές πλέον, πως η χρήση του *buffering*, πραγματικά μπορεί να προσφέρει στην βελτιστοποίηση προγραμμάτων Java.

Βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου UnBuffered.java. Ο profiler θα μας έδινε την main σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.39). Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε από τις γραμμές με τις αντίστοιχες μεταβλητές, ότι διαβάζεται χαρακτήρα προς χαρακτήρα ένα αρχείο, μετρώντας την εμφάνιση του χαρακτήρα ‘x’. Θα σκεφτόμασταν πως θα ήταν καλύτερα και πιο βέλτιστο, να χωρίζαμε το αρχείο σε μερίδια από bytes, έτσι ώστε να κερδίζαμε κάποιο χρόνο χρησιμοποιώντας την τεχνική του buffering. Επομένως, θα πραγματοποιούσαμε τις αλλαγές και θα παίρναμε τον κώδικα του αρχείου Buffered.java.

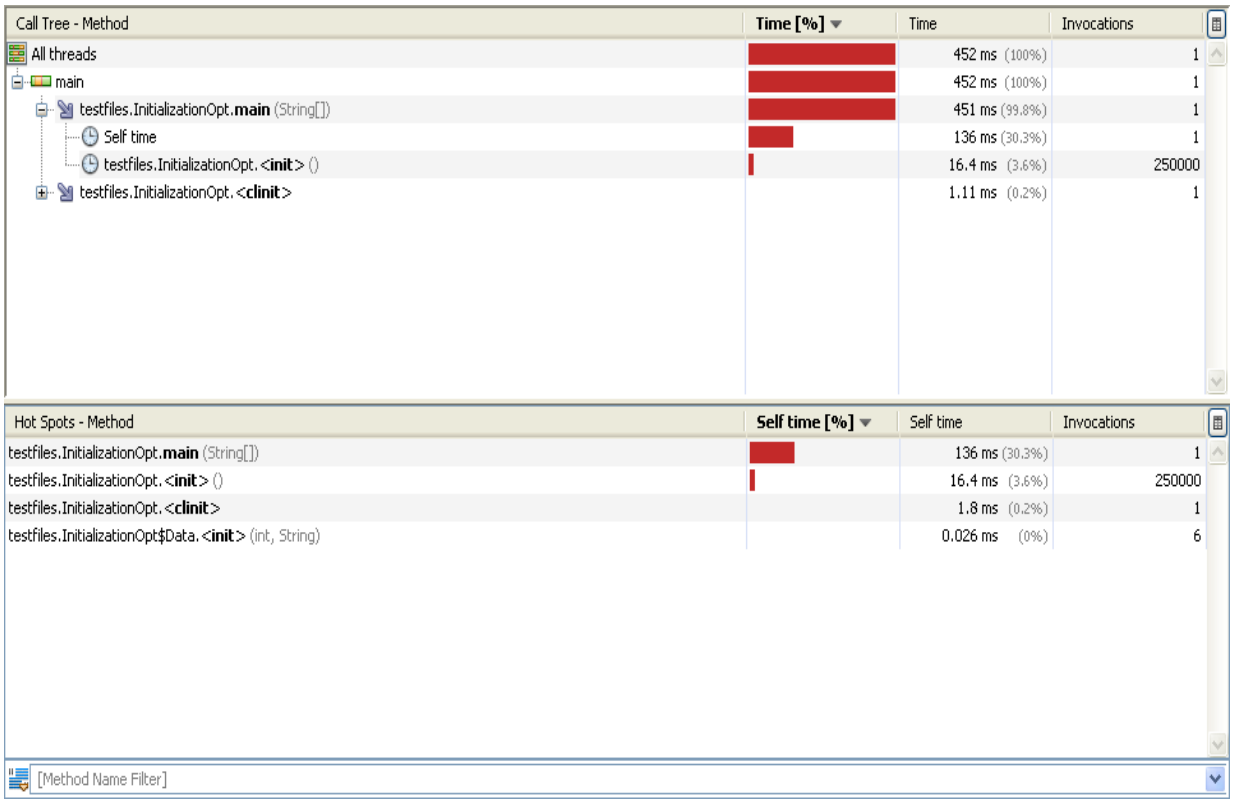
Στο κεφάλαιο 3 είχαμε πει πως η διαφορά μεταξύ των δυο ειδών αρχικοποίησης, στατικών και μη-στατικών, μπορεί να παίξει καθοριστικό ρόλο στην επίδοση. Τώρα θα δούμε δυο προγράμματα, που το ένα κάνει χρήση μη-στατικής αρχικοποίησης αντικειμένου ενώ το άλλο στατικής.

Το πρόγραμμα Initialization.java [Παράρτημα Γ-35] δημιουργεί 250000 στιγμιότυπα της κλάσης Initialization. Δημιουργώντας αυτά τα στιγμιότυπα, αρχικοποιείται αχρείαστα κάθε φορά μαζί και ο πίνακας months με αντικείμενα τύπου Data. Αυτά τα αντικείμενα παραμένουν τα ίδια για κάθε επανάληψη. Συνεπώς, θα ήταν καλύτερα η αρχικοποίηση αυτού του πίνακα να γινόταν μόνο μια φορά στην αρχή και όχι να επαναλαμβάνεται με την δημιουργία στιγμιότυπων της κλάσης Initialization. Αυτό επιτυγχάνεται στο πρόγραμμα InitializationOpt.java [Παράρτημα Γ-36]. Η μόνη διαφορά που υπάρχει είναι ότι κάναμε τον πίνακα months, static (στατικό). Όπως,

έχουμε πει και στο κεφάλαιο 3, οι static μεταβλητές χρειάζονται να αρχικοποιηθούν μόνο μια φορά και έπειτα χρησιμοποιούνται χωρίς να επαναληφθεί ξανά η αρχικοποίηση τους. Η διαφορά στους χρόνους εκτέλεσης των δυο προγραμμάτων, φαίνεται στα πιο κάτω snapshots.



Εικόνα 4.41: Profiling αρχείου Initialization.java



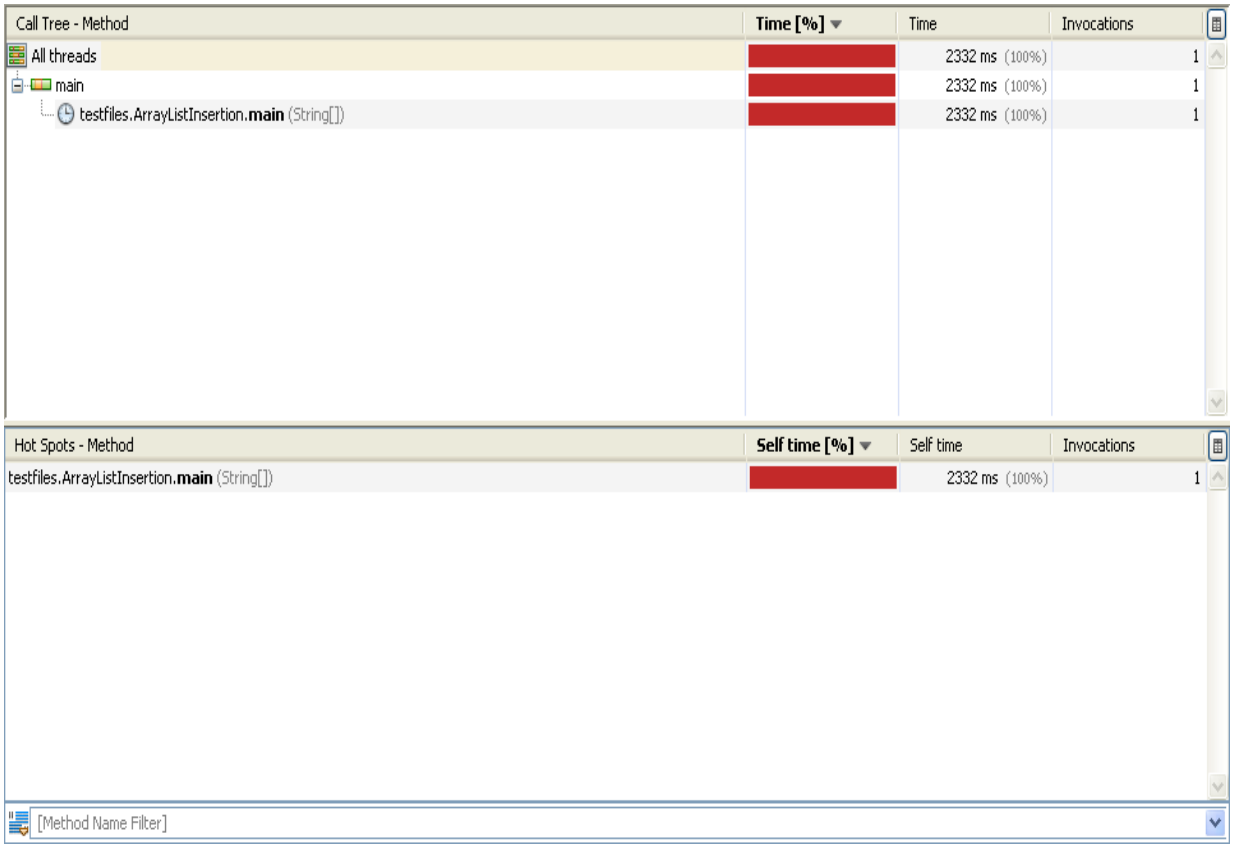
Εικόνα 4.42: Profiling αρχείου InitializationOpt.java

Στην εικόνα 4.41 παρατηρούμε ότι ο ολικός χρόνος εκτέλεσης του προγράμματος Initialization.java είναι 3730 ms. Στην εικόνα 4.42, ο αντίστοιχος ολικός χρόνος εκτέλεσης του προγράμματος InitializationOpt.java, είναι 452 ms. Είναι προφανές λοιπόν, ότι η αποφυγή αχρείαστων αρχικοποιήσεων με την βοήθεια του static, μας έδωσε ένα πιο βελτιστοποιημένο πρόγραμμα.

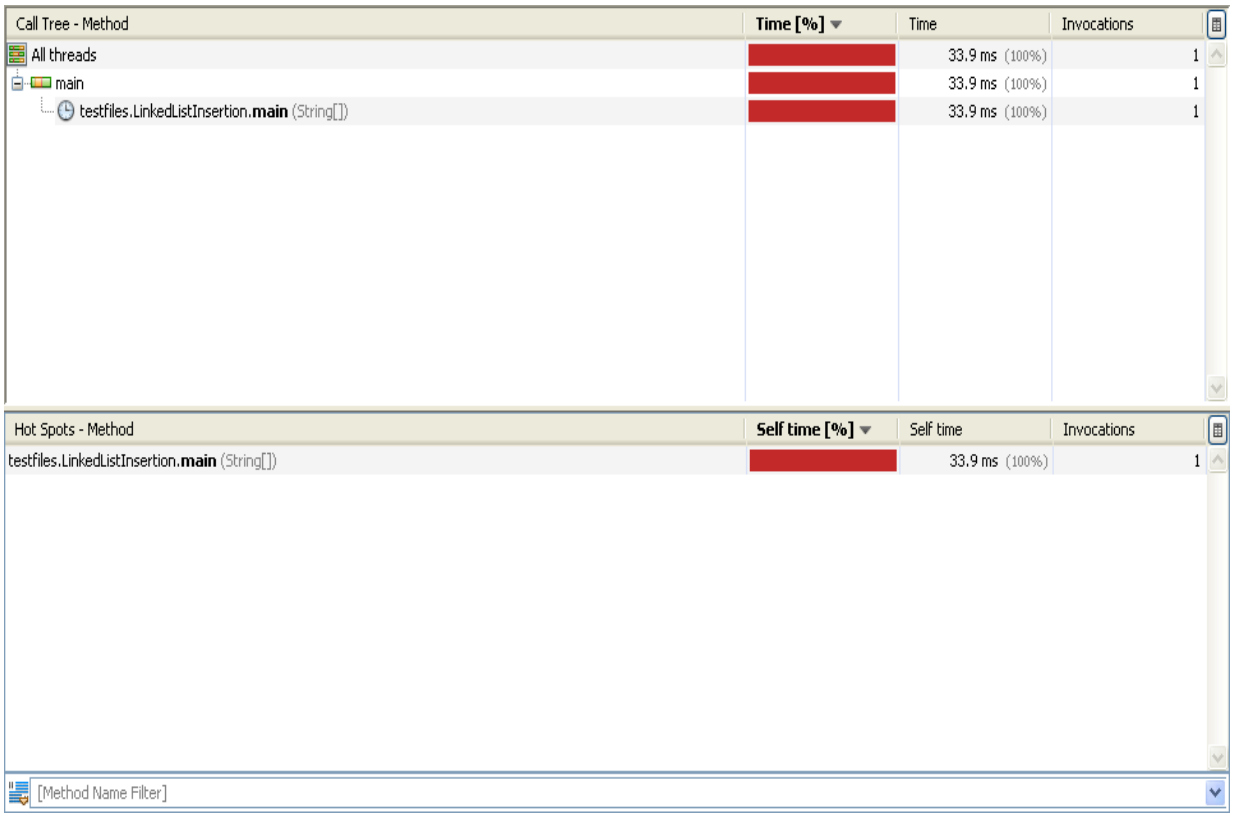
Βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει το πιο χρονοβόρο κομμάτι του αρχείου Initialization.java. Ο profiler θα μας έδινε την αρχικοποίηση της εσωτερικής κλάσης Data σαν το πιο χρονοβόρο κομμάτι (όπως παρατηρούμε πιο πάνω στην εικόνα 4.41). Αυτή η πληροφορία όμως, λόγω του ότι δεν είναι μέθοδος δεν μπορεί να χρησιμοποιηθεί από τον συντακτικό αναλυτή. Το δεύτερο πιο χρονοβόρο κομμάτι που θα μας έδινε ο profiler είναι η αρχικοποίηση της κλάσης Initialization (εικόνα 4.41). Ούτε και αυτή η πληροφορία όμως μπορεί να χρησιμοποιηθεί από τον συντακτικό αναλυτή. Το τρίτο πιο χρονοβόρο κομμάτι είναι η μέθοδος main. Επειδή είναι μέθοδος, μπορεί να χρησιμοποιηθεί από τον συντακτικό αναλυτή. Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο

που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε από τις γραμμές με τις αντίστοιχες μεταβλητές, ότι χρησιμοποιείται ένα “for” loop που δημιουργεί 250000 στιγμιότυπα της κλάσης Initialization. Δημιουργώντας αυτά τα στιγμιότυπα, αρχικοποιείται αχρείαστα κάθε φορά μαζί και ο πίνακας months με αντικείμενα τύπου Data. Επομένως, θα σκεφτόμασταν να μετατρέψουμε σε static τον πίνακα months, με αποτέλεσμα να γλιτώσουμε από τις αχρείαστες αρχικοποιήσεις και να πετύχουμε την δημιουργία ενός πιο βέλτιστου προγράμματος. Επομένως, θα παίρναμε τον κώδικα που βλέπουμε στο αρχείου InitializationOpt.java.

Στην συνέχεια, ας δούμε κάποιες διαφορές στην επίδοση μεταξύ ArrayList [7] και LinkedList [7]. Το πρόγραμμα ArrayListInsertion.java [Παράρτημα Γ-30] και το πρόγραμμα LinkedListInsertion.java [Παράρτημα Γ-38] κάνουν και τα δυο εισαγωγή νέων στοιχείων στην αρχή (στην θέση 0), μιας λίστας. Το πρώτο χρησιμοποιεί σαν λίστα ένα ArrayList και το δεύτερο, ένα LinkedList. Ακολουθούν τα snapshots από τον profiler.



Εικόνα 4.43: Profiling αρχείου ArrayListInsertion.java



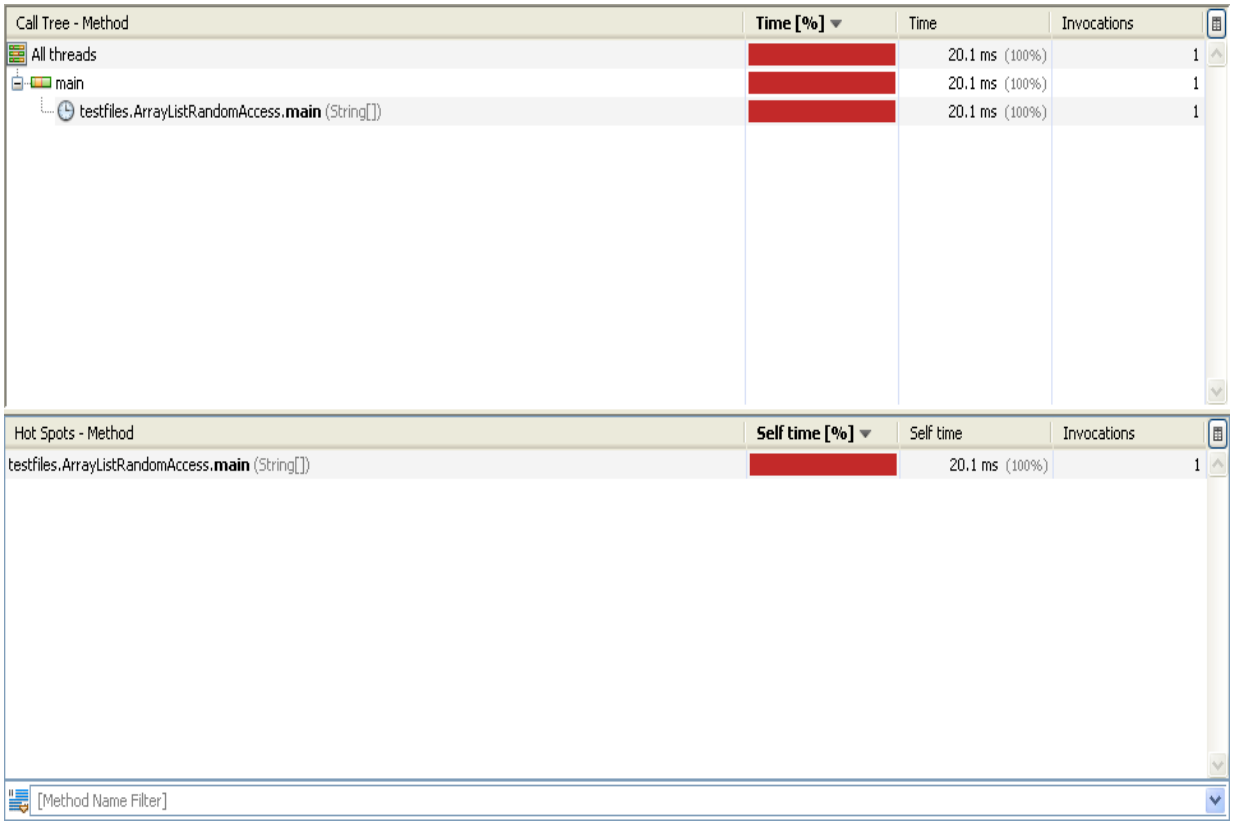
Εικόνα 4.44: Profiling αρχείου LinkedListInsertion.java

Στην εικόνα 4.43 παρατηρούμε ότι ο ολικός χρόνος εκτέλεσης του προγράμματος ArrayListInsertion.java είναι 2332 ms. Στην εικόνα 4.44, ο αντίστοιχος ολικός χρόνος εκτέλεσης του προγράμματος LinkedListInsertion.java, είναι 33.9 ms. Η εισαγωγή στοιχείων στην αρχή ενός ArrayList, απαιτεί όλα τα υπόλοιπα στοιχεία που έχουν ήδη εισαχθεί να σπρωχθούν προς τα κάτω. Αυτό είναι μια ακριβή πράξη. Από την άλλη, η εισαγωγή στοιχείων στην αρχή ενός LinkedList, είναι πιο φθηνή διαδικασία, λόγω του γεγονός ότι τα στοιχεία είναι συνδεδεμένα μεταξύ τους με συνδέσμους (links). Επομένως, είναι πιο εύκολο να δημιουργηθεί ένα καινούργιο στοιχείο και να συνδεθεί στην κορυφή (head) της λίστας.

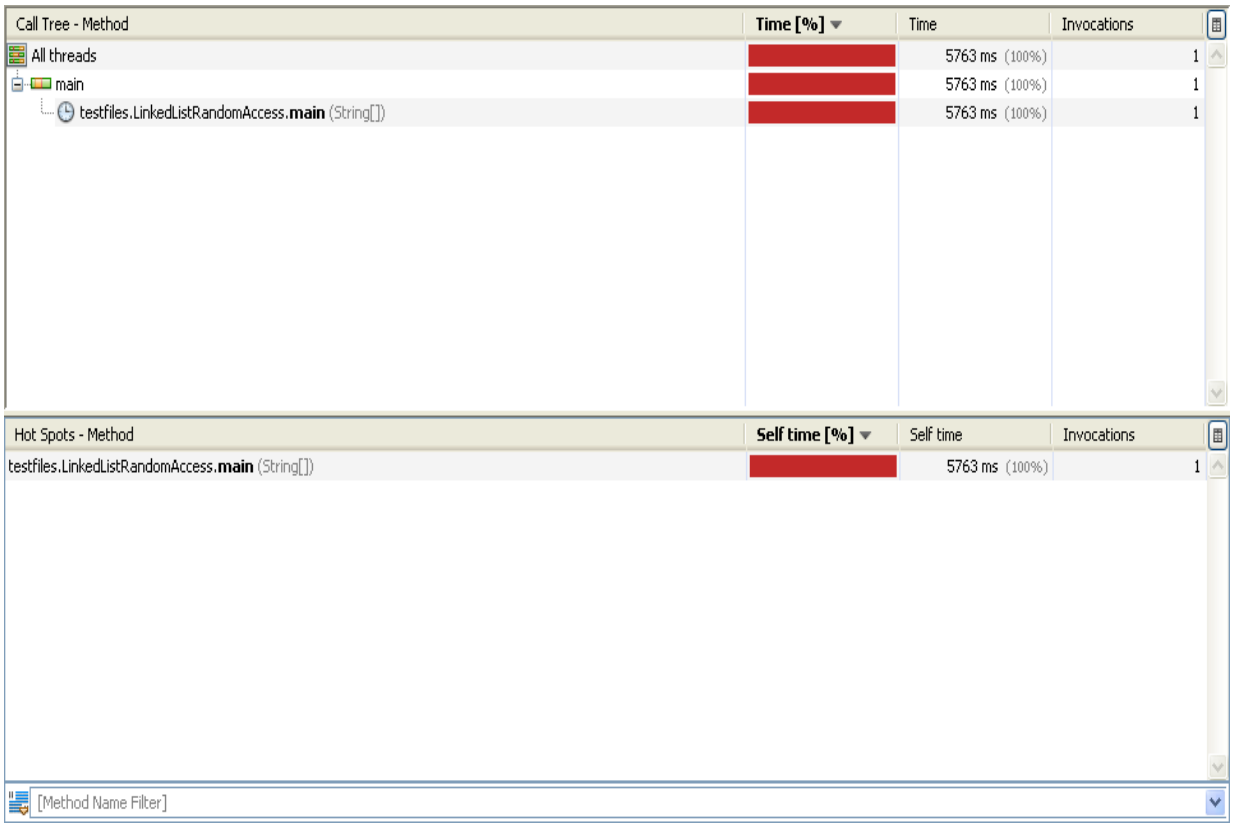
Στο πιο πάνω παράδειγμα, βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου ArrayListInsertion.java. Ο profiler θα μας έδινε την main σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.43). Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο που θα παραγόταν από τον

συντακτικό αναλυτή, θα παρατηρούσαμε από τις γραμμές με τις αντίστοιχες μεταβλητές, ότι χρησιμοποιείται ένα "for" loop για να κάνει 50000 εισαγωγές νέων στοιχείων στην αρχή μιας λίστας, χρησιμοποιώντας ένα ArrayList. Επομένως, θα σκεφτόμασταν ότι σε αυτή την περίπτωση θα ήταν πιο βέλτιστο να χρησιμοποιούσαμε ένα LinkedList. Έτσι, θα παίρναμε τον κώδικα που φαίνεται στο αρχείο LinkedListInsertion.java.

Ακολούθως, ας δούμε τι γίνεται στην περίπτωση που κάνουμε εισαγωγή στοιχείων στο τέλος μιας λίστας και έπειτα επιχειρήσουμε τυχαία (random) ανάκτηση στοιχείων από αυτή. Το πρόγραμμα ArrayListRandomAccess.java [Παράρτημα Γ-30] χρησιμοποιεί ArrayList και το πρόγραμμα LinkedListRandomAccess.java [Παράρτημα Γ-38] χρησιμοποιεί LinkedList για επίτευξη αυτής της τυχαίας ανάκτησης στοιχείων.



Εικόνα 4.45: Profiling αρχείου ArrayListRandomAccess.java



Εικόνα 4.46: Profiling αρχείου LinkedListRandomAccess.java

Στην εικόνα 4.45 παρατηρούμε ότι ο ολικός χρόνος εκτέλεσης του προγράμματος ArrayListRandomAccess.java είναι 20.1 ms. Στην εικόνα 4.46, ο αντίστοιχος ολικός χρόνος εκτέλεσης του προγράμματος LinkedListRandomAccess.java, είναι 5763 ms. Συμπεραίνουμε λοιπόν, ότι είναι πιο φθηνό (από πλευράς χρόνου εκτέλεσης) να χρησιμοποιήσουμε ένα ArrayList για τυχαία ανάκτηση στοιχείων από μια λίστα, παρά να χρησιμοποιήσουμε ένα LinkedList. Με το LinkedList αυτή η διαδικασία είναι πιο χρονοβόρα διότι πρέπει να προσπελαστούν τα στοιχεία μέχρι να βρεθεί αυτό που αναζητείται.

Επομένως, συνοψίζοντας μπορούμε να πούμε ότι το ArrayList έχει καλύτερη επίδοση σε σχέση με το LinkedList, όταν έχουμε περιπτώσεις που θέλουμε να προσθέσουμε στοιχεία στο τέλος μιας λίστας και να τα ανακτήσουμε με τυχαίο τρόπο. Από την άλλη, το LinkedList έχει καλύτερη επίδοση σε σχέση με το ArrayList, όταν έχουμε περιπτώσεις που θέλουμε να προσθέσουμε ή να αφαιρέσουμε στοιχεία από την αρχή ή το μέσο μιας λίστας.

Στο συγκεκριμένο παράδειγμα, βάσει της μεθοδολογίας μας, θα χρησιμοποιούσαμε τον profiler για να μας εντοπίσει τις πιο χρονοβόρες μεθόδους του αρχείου LinkedListRandomAccess.java. Ο profiler θα μας έδινε την main σαν την πιο χρονοβόρα μέθοδο, (όπως παρατηρούμε πιο πάνω στην εικόνα 4.46). Χρησιμοποιώντας τον συντακτικό αναλυτή θα εντοπίζαμε τις γραμμές με τις αντίστοιχες μεταβλητές μέσα στην εμβέλεια αυτής της μεθόδου. Στο highlighted αρχείο που θα παραγόταν από τον συντακτικό αναλυτή, θα παρατηρούσαμε από τις γραμμές με τις αντίστοιχες μεταβλητές, ότι χρησιμοποιείται ένα "for" loop για να κάνει 50000 εισαγωγές νέων στοιχείων στο τέλος μιας λίστας, χρησιμοποιώντας ένα LinkedList και έπειτα χρησιμοποιείται ένα άλλο "for" loop για να ανακτήσει τυχαία τα στοιχεία από αυτή την λίστα. Επομένως, θα σκεφτόμασταν ότι σε αυτή την περίπτωση θα ήταν καλύτερα να χρησιμοποιούσαμε ένα ArrayList για να κάνουμε τυχαία ανάκτηση στοιχείων, (από μια λίστα) που είναι πιο βέλτιστο. Έτσι, θα παίρναμε τον κώδικα που φαίνεται στο αρχείο ArrayListRandomAccess.java.

Κεφάλαιο 5

Συμπεράσματα

5.1 Εισαγωγή

Η εργασία αυτή σκοπό είχε την δημιουργία ενός συντακτικού αναλυτή, που με την βοήθεια ενός εργαλείου τεμαχισμού προγράμματος, να μπορεί να χρησιμοποιηθεί στην βελτιστοποίηση προγραμμάτων γραμμένα σε Java.

Σκοπός του συντακτικού αναλυτή είναι να εντοπίσει τα σημεία όπου υπάρχουν loops, αναδρομικές μεθόδοι και τελεστές modulo. Αυτά τα σημεία στα πλείστα προγράμματα συνήθως είναι η αιτία καθυστέρησης του χρόνου εκτέλεσης τους και επομένως με την βελτιστοποίηση αυτών των σημείων μπορούμε να πετύχουμε στην δημιουργία ενός βέλτιστου προγράμματος. Μέσα σε αυτά τα σημεία εντοπίζονται οι γραμμές που περιέχουν μεταβλητές. Επιπλέον, μπορούν να εντοπιστούν και οι γραμμές με τις μεταβλητές κάθε μεθόδου.

Έπειτα, αυτές οι γραμμές θα χρησιμοποιηθούν από ένα εργαλείο τεμαχισμού προγράμματος, (που δυστυχώς δεν έχει επιτευχθεί η εγκατάσταση του), με αποτέλεσμα να απομονωθούν τα σημεία του κώδικα που θεωρούνται χρονοβόρα. Παρά το γεγονός, ότι δεν έγινε τελικά χρήση τέτοιου εργαλείου, το μόνο που μας έχει στερήσει είναι την ικανότητα να εξετάσουμε από περισσότερες πλευρές κάποια κομμάτια κώδικα. Παρόλα

αυτά, ακόμα μπορούμε να κατευθυνθούμε προς τα τυχόν χρονοβόρα σημεία, ενός (πηγαίου) κώδικα, με την βοήθεια του συντακτικού αναλυτή.

Τέλος, με βάση τις εισηγήσεις που έχουν προταθεί για βελτιστοποίηση κώδικα, μπορούμε να παρατηρήσουμε αν κάτι μπορεί να βελτιστοποιηθεί. Αν υπάρχει κάτι που μπορεί να βελτιστοποιηθεί, τότε κάνουμε τις απαραίτητες αλλαγές και χρησιμοποιώντας τον profiler, επαληθεύουμε αν έχουμε πετύχει τον στόχο μας.

5.2 Συμπεράσματα

Η μελέτη αυτή επομένως είχε ως στόχο να εισηγηθεί μια μεθοδολογία που να μπορεί να προσφέρει βοήθεια στην βελτιστοποίηση προγράμματος. Για τις απαιτήσεις αυτής της διπλωματικής εργασίας, μεγαλύτερο μέρος αφιερώθηκε στην δημιουργία του συντακτικού αναλυτή κώδικα, σε Java. Ο τρόπος με τον οποίο διεξάχθηκε η υλοποίηση του είναι αρκετά αφαιρετικός και επαναχρησιμοποιήσιμος, με αποτέλεσμα στον μέλλον να μπορεί να βοηθήσει στην περαιτέρω επέκταση του.

Όσον αφορά τα αποτελέσματα των πειραματικών μετρήσεων της επίδοσης διαφόρων μικρών προγραμμάτων που έρχονται να επαληθεύσουν κάποιες από τις εισηγήσεις που προτάθηκαν για βελτιστοποίηση προγραμμάτων Java, μπορούμε να συμπεράνουμε τα εξής:

- Ο καλύτερος τρόπος με τον οποίο μπορούμε να πετύχουμε στην βελτιστοποίηση ενός προγράμματος είναι με την χρήση ενός διαφορετικού αλγορίθμου που επιλύει το ίδιο πρόβλημα αλλά με πιο βέλτιστο τρόπο. Αυτό είναι κάτι το οποίο ισχύει γενικότερα για όλες τις γλώσσες προγραμματισμού και όχι μόνο για την Java. Έχουμε δει και από τα αποτελέσματα των πειραματικών μετρήσεων με τον profiler τις διαφορές μεταξύ των δυο αλγόριθμων αναζήτησης και των δυο αλγόριθμων ταξινόμησης, όσον αφορά τους χρόνους εκτέλεσης του καθενός. Συνεπώς, η εύρεση κάποιου καλύτερου αλγόριθμου από αυτό που χρησιμοποιείται ήδη, στις περισσότερες περιπτώσεις μπορεί να βγει νικητής, στο θέμα της βελτιστοποίησης προγράμματος.

Επιπλέον, η αποφυγή της χρήσης της αναδρομικότητας στα προγράμματα μας, μπορεί και αυτή με την σειρά της να συνεισφέρει στην βελτιστοποίηση, από ότι παρατηρήσαμε στην διαφορά εύρεσης του factorial και του n-οστού αριθμού fibonacci.

- Η χρήση του τελεστή "+" μέσα σε loops, με σκοπό την συνένωση string, από τα αποτελέσματα που έχουμε πάρει αποδεικνύεται πως πραγματικά προδίδει κάποιο κόστος, επομένως είναι καλύτερα να μάθουμε να χρησιμοποιούμε τον StringBuilder.

- Έχουμε επίσης αποδείξει πως η χρήση του *buffering* στην Java, μπορεί να κάνει κάποια αισθητή διαφορά στην βελτιστοποίηση προγράμματος. Έτσι, είναι καλή ιδέα να ξεκινήσουμε την ένταξη του *buffering* στον προγραμματισμό μας. Μέχρι στιγμής, οι περισσότεροι από μας, είτε λόγω άγνοιας της ύπαρξης του *buffering*, είτε λόγω του γεγονός ότι βαριόμαστε να γράψουμε κάποιες επιπρόσθετες γραμμές κώδικα, (αφού μπορούμε να πετύχουμε το ίδιο αποτέλεσμα με πιο λίγο κώδικα), αποφεύγαμε την χρήση του.

- Επιπλέον, πολλές φορές η αχρείαστη ή απερίσκεπτη ενέργεια της αρχικοποίησης αντικειμένων στην Java, μπορεί να μας κοστίσει στον χρόνο εκτέλεσης ενός προγράμματος. Όπως, έχουμε δει με μια προσεκτική παρατήρηση, η χρήση και μόνο του static μας έδωσε καλύτερη επίδοση. Συμπεραίνουμε λοιπόν γενικότερα, πως με την προσεκτική παρατήρηση μπορούμε να εντοπίσουμε κάποιες απερίσκεπτες ενέργειες που έχουμε κάνει κατά την διάρκεια της υλοποίησης ενός προγράμματος και να της διορθώσουμε. Ο εναλλακτικός τρόπος υλοποίησης κάποιου κομματιού κώδικα μπορεί σε αρκετές περιπτώσεις να παίξει καθοριστικό ρόλο στην βελτιστοποίηση προγράμματος.

- Τέλος, η γνώση των χαρακτηριστικών των έτοιμων δομών δεδομένων που χρησιμοποιεί η Java και γενικότερα οι αντικειμενοστραφής γλώσσες προγραμματισμού, μπορεί και αυτή να προσφέρει στην βελτιστοποίηση. Έχουμε δει τις διαφορές μεταξύ του ArrayList και του LinkedList στην Java. Συνεπώς, όχι μόνο για την Java αλλά γενικότερα, πρέπει πριν ξεκινήσουμε να ασχολούμαστε με θέματα βελτιστοποίησης να έχουμε μελετήσει τι μπορεί να μας προσφέρει η κάθε έτοιμη δομή δεδομένων μιας

γλώσσας προγραμματισμού και τα πλεονεκτήματα και μειονεκτήματα από την χρήση της καθεμιάς.

Σε γενικές γραμμές, έχουμε παρατηρήσει πως αν επιθυμούμε να βελτιστοποιήσουμε ένα πρόγραμμα, υπάρχουν μερικές καλές εισηγήσεις που μπορούμε να εξετάσουμε, όπως αναφέραμε πιο πάνω και γενικότερα στο κεφάλαιο 3. Στις περισσότερες περιπτώσεις όμως αυτές οι εισηγήσεις έχουν ακολουθηθεί ήδη, με αποτέλεσμα να μην ξέρουμε για κάτι που είναι εμφανές στον κώδικα μας και να το τροποποιήσουμε. Σε τέτοιες περιπτώσεις η διαδικασία για την επίτευξη βελτιστοποίησης ενός προγράμματος, μετατρέπεται σε κάτι το χρονοβόρο. Για αυτό τον λόγο πρέπει να είμαστε απολύτως σίγουροι κατά πόσο χρειάζεται να γίνει η βελτιστοποίηση και ποια πλεονεκτήματα και μειονεκτήματα, μπορεί να μας αποφέρει.

5.3 Μελλοντική Εργασία

Σε αυτό το κομμάτι θα δούμε ορισμένα σημεία που μπορούν να γίνουν στο μέλλον σε σχέση με αυτή την ατομική διπλωματική εργασία.

- Η υλοποίηση του συντακτικού αναλυτή, έγινε με τέτοιο τρόπο που μπορεί να βοηθήσει στην επέκταση του. Πιο συγκεκριμένα, μπορούν να γίνουν αλλαγές στις κανονικές εκφράσεις που χρησιμοποιήθηκαν από τον συντακτικό αναλυτή και να γίνει υποστήριξη άλλων γλωσσών προγραμματισμού που έχουν σύνταξη παρόμοια με αυτή της Java. Για παράδειγμα, μπορούν να υποστηριχθούν οι γλώσσες προγραμματισμού C, C++ και C#.

- Μπορεί να υλοποιηθεί μια επιπρόσθετη επιλογή για τον συντακτικό αναλυτή, όπου θα συνοψίζει όλες τις λειτουργίες του σε μία. Δηλαδή, θα δίδεται μια λεπτομερή αναφορά για κάθε μέθοδο αν είναι αναδρομική η όχι, αν χρησιμοποιεί κάποιο είδος loop ή αν κάνει χρήση του τελεστή modulus. Ακολούθως, θα δίδονται όλες οι γραμμές με τις αντίστοιχες μεταβλητές όλων των ειδών των χρονοβόρων κομματιών (loops, αναδρομικές μεθόδοι κτλ).

- Ο συντακτικός αναλυτής που έχει δημιουργηθεί μπορεί να ενσωματωθεί με το εργαλείο τεμαχισμού κώδικα που έχει γίνει προσπάθεια για κατασκευή του σε παλαιότερες διπλωματικές εργασίες, χωρίς όμως να τελειοποιηθεί πλήρως η κατασκευή του. Το συγκεκριμένο εργαλείο έχει γραφτεί σε Netbeans IDE και κάνει χρήση γραφικής διαπροσωπείας (GUI). Θα μπορούσε επομένως, να μεταφερθεί και ο συντακτικός αναλυτής σαν κομμάτι αυτού του εργαλείου και να γίνει και γι' αυτόν μια γραφική διαπροσωπεία.

- Όπως έχει αναφερθεί, δεν έγινε επιτυχής η εγκατάσταση ενός εργαλείου τεμαχισμού προγράμματος (JSlice). Στον μέλλον μπορεί να υπάρξουν σίγουρα καλύτερα εργαλεία (και προπαντός συντηρήσιμα) από ότι το JSlice και να γίνει με επιτυχία η "συνένωση" τους με τον συντακτικό αναλυτή. Περισσότερο χρήσιμο θα ήταν βεβαίως αν αυτό το εργαλείο, δούλευε σαν plug-in (ενσωμάτωση) κάποιου IDE της Java. Οπότε ελπίζουμε πως στο μέλλον θα δημιουργηθεί ένα καλό εργαλείο τεμαχισμού προγράμματος, που να ολοκληρώνει την ιδέα αυτής της μεθοδολογίας.

- Βάση των εισηγήσεων που έχουν προταθεί για βελτιστοποίηση προγράμματος, μπορεί να γίνει μια προσπάθεια αυτόματης υλοποίησης τους. Με την δημιουργία ενός συστήματος βασισμένου σε κανόνες, μπορεί να γίνει έξυπνη αντικατάσταση των σημείων που εντοπίζονται ως χρονοβόρα, με κομμάτια κώδικα λιγότερο χρονοβόρα. Η επίτευξη τέτοιου στόχου σίγουρα είναι αρκετά υψηλή, όμως στο μέλλον μπορεί με την χρήση κάποιου ευφυούς συστήματος να δημιουργηθούν προγράμματα που να κάνουν βελτιστοποίηση άλλων προγραμμάτων.

Βιβλιογραφία

[1] Aho V. Alfred, Lam S. Monica, Sethi Ravi, Ullman D. Jeffrey, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2007.

[2] Baowen Xu Ju Qian, Xiaofang Zhang, Zhongqiang Wu, Li Chen , "A Brief Survey Of Program Slicing", ACM SIGSOFT Software Engineering Notes, Volume 30 Number 2, March 2005.

[3] Bodgan Korel, "Computation of Dynamic Program Slices for Unstructured Programs", IEEE Transactions on Software Engineering, Vol. 23, No. 1, January 1997.

[4] Bodgan Korel, Jurgen Rilling, "Dynamic program slicing methods", Information and Software Technology 40 647-659, 1998.

[5] D. W. Binkley, K. B. Gallagher, "Program slicing", in: M. Zelkowitz (Ed.), Advances in Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[6] Dave Binkley, Sebastian Danicic, Tibor Gyimothy, Mark Harman, Akos Kiss, Bogdan Korel, "Theoretical foundations of dynamic program slicing", February 2005.

[7] Eckel Bruce, Thinking in Java, Prentice-Hall, 2006.

[8] Frank Tip, "A Survey of Program Slicing Techniques", Journal of Programming Languages Vol. 3, No. 3, p. 121-189, September 1995.

[9] Glen I. Magee, "Code Optimization Techniques", August 10, 2000.

[10] Glen McCluskey, “Thirty Ways to Improve the Performance of your Java™ Programs”, Version 1.0, October 1999.

[11] H. Agrawal and J.R. Horgan, “Dynamic program slicing”, Proceedings of ACM SIGPLAN Conference on programming Language Design and Implementation, White Plains, New York, U.S.A., ACM Press, pp. 246-256, 1990.

[12] Weiss Allen Mark, Data structures and algorithms analysis in C, Addison-Wesley, 1997

[13] JSlice - a Java Dynamic Slicing Tool, http://jslice.sourceforge.net/, Accessed: February 2010.

[14] Lesson: Regular Expressions, http://java.sun.com/docs/books/tutorial/essential/regex/intro.html, Accessed: March 2010.

[15] Introduction to Profiling Java Applications in NetBeans IDE, http://netbeans.org/kb/docs/java/profiler-intro.html, Accessed: June 2010.

[16] Eclipse.org home, http://www.eclipse.org, Accessed: January 2008.

[17] Program optimization – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Program_optimization, Accessed: March 2010.

[18] Paul Hsieh, Programming Optimization, http://www.azillionmonkeys.com/qed/optimize.html, Accessed: March 2010.

[19] Doug Bell, Make Java fast: Optimize!, http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html, Accessed: March 2010.

[20] Sudhir Ancha, Java Programming Optimization Tips, http://www.javacommerce.com/displaypage.jsp?name=java_performance.sql&id=18264, Accessed: June 2010.

[21] Loop optimization – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Loop_optimization, Accessed: June 2010.

[22] Parallel algorithm – Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Parallel_algorithm, Accessed: June 2010.

Παράρτημα Α

Κώδικες Υλοποίησης Συντακτικού Αναλυτή

Constants.java

```
/* * * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.TextFile.*;
import java.util.regex.*;


/* Class responsible for declaring all the constants of regular expressions patterns and
strings */
public final class Constants {

	/* Regular Expressions */
	public static final String MODIFIERS =
"(((\\s*)(public|private|protected|static|final)(\\s+))*";
	public static final String VAR_TYPE = "\\s*(\\w++)";
	public static final String VAR_NAME = "\\s*(\\w+)\\s*";
	public static final String BRACKETS = "(\\s*\\[\\s*\\])*";
	public static final String GENERICS = "\\s*(<(\\s*((\\w+)|\\?|<|>)\\s*)+?)*";
	public static final String MORE_VARS = "(=.*?|(\\s*,\\s*(\\w+))*?)*?\\s*;\\s*";
	public static final String METHOD_NAME = "\\s*(\\w+)\\s*";
	public static final String RETURNED_TYPE = "\\s*(\\w++)";
	public static final String PARAMETERS = "\\((.*?)\\).*?\\{";
	public static final String PARENTHESIS = "\\s*\\((.*?)\\)";
	public static final String SEPARATOR = "\\s*,\\s*(\\w+)";
	public static final String COMMA  = ",*?";
	public static final String QUALIFED_NAME = "\\w+\\.";
	public static final String STR_QUOTES = "\"(.*?)\"";
	public static final String CHR_QUOTES = "'(.*?)'";
	public static final String FOR_PATTERN = "for\\s*\\((.*?;).*?;.*?\\)";
	public static final String FOR_EACH_PATTERN = "for\\s*\\((.*?):.*?\\)";
	public static final String COMMENTS = "(/\\*(.|\\s)*?\\*/)|(//.*)";
	public static final String FILE_PATTERN = "(\\w++)\\.java";
	public static final String CURL_BRACKETS = "\\{|\\}";
	public static final String WHILE_PATTERN = "while\\s*\\((.*?)\\)\\s*\\{";
	public static final String DO_WHILE_PATTERN = "do\\s*\\{";
	public static final String DO_WHILE_END = "\\s*while\\s*\\((.*?)\\)\\s*;\\s*";
	public static final String RESERVED_WORD =
"\\s*(public|private|protected|else)\\s*";
	public static final String CATCH_CLAUSE_PATTERN = "catch\\s*\\((.*?)\\)";
	public static final String MODULUS = "%";

	/* String names */
	public static final String PACKAGE_NAME = "package testing.files;";
	public static final String FOR_PATENT = "for (myForPatent)";
	public static final String TEST_DIR = "./testing/";
	public static final String FILES_DIR = "./testing/files/";
	public static final String HTML_DIR = "./testing/htmlview/";
	public static final String QUALIFIED_DIR = "testing.";
	public static final String JAVA_EXT = ".java";
	public static final String HTML_EXT = ".html";
	public static final String START_TAGS = "<html>" + LINE_SEPARATOR +
```

```
        "<head>" + LINE_SEPARATOR +

        "<style type=\"text/css\">" + LINE_SEPARATOR+

        ".style {" + LINE_SEPARATOR +

        "color: #FF0000;" + LINE_SEPARATOR +
                                                                                "font-
weight: bold;" + LINE_SEPARATOR +

        "background-color: #FFFF00;" + LINE_SEPARATOR +
                                                                                "}" +
LINE_SEPARATOR +

        "</style>" + LINE_SEPARATOR +

        "</head>" + LINE_SEPARATOR +

        "<body>" + LINE_SEPARATOR +

        "<pre>";
        public static final String END_TAGS = "</pre>" + LINE_SEPARATOR +
                                                                        "</body>" +
LINE_SEPARATOR +
                                                                        "</html>";
        public static final String HIGHLIGHT_TAG = "<span class=\"style\">";
        public static final String END_HIGHLIGHT_TAG = "</span>";

        /* Pattern Declarations */
        public static Pattern STRING_QUOTES = Pattern.compile(STR_QUOTES);
        public static Pattern CHAR_QUOTES = Pattern.compile(CHR_QUOTES);
        public static Pattern FOR = Pattern.compile(FOR_PATTERN);
        public static Pattern FOR_EACH = Pattern.compile(FOR_EACH_PATTERN);
        public static Pattern CURLY_BRACKETS = Pattern.compile(CURL_BRACKETS);
        public static Pattern WHILE = Pattern.compile(WHILE_PATTERN);
        public static Pattern DO_WHILE = Pattern.compile(DO_WHILE_PATTERN);
        public static Pattern METHODS = Pattern.compile(MODIFIERS + "(" + RETURNED_TYPE
        + GENERICS + BRACKETS + ")"
                                                                + METHOD_NAME +
PARAMETERS);
        public static Pattern CATCH_CLAUSE = Pattern.compile(CATCH_CLAUSE_PATTERN);
        public static Pattern MODULUS_OP = Pattern.compile(MODULUS);

}
```

DirectoryList.java

```
/* * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import java.util.*;
import java.io.*;
import java.util.regex.*;

/* Class responsible for listing the files inside the eclipse directory */
public class DirectoryList {

        /* Method that returns a list of files inside the eclipse directory */
        public ArrayList<String> getFilesName(String startDir, String javaExt) {

                ArrayList<String> files = new ArrayList<String>();
                File path = new File(startDir);
                String[] list;
```

```
                Matcher fileExtension = null;

                list = path.list();

                for (String fileName : list) {
                        // regular expression of the file pattern
                        fileExtension = Pattern.compile(javaExt).matcher(fileName);
                        if (fileExtension.matches())
                                files.add(fileExtension.group(1));
                }

                return files;
        }

}
```

Formatter.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
import java.util.*;
import java.util.regex.*;

/* Class responsible for formatting the file given in the way that each line of the file
finishes
 * with either "{", "}", ";", in order to ease parsing of code */
public class Formatter {

        /* regular expressions patterns used */
        private Pattern forPattern = Pattern.compile(FOR_PATTERN);
        private Pattern forPatent = Pattern.compile("for \\(myForPatent\\)\\d+");
        private Pattern comments = Pattern.compile(COMMENTS);

        /* list to hold temporary the char quotes, string quotes and for loops
declarations */
        private ArrayList<String> quotedString = new ArrayList<String>();
        private ArrayList<String> quotedChar = new ArrayList<String>();
        private ArrayList<String> forString = new ArrayList<String>();

        private String fileName;

        /* Constructor of the Formatter class */
        public Formatter(String fileName) {
                this.fileName = fileName;
        }

        /* Method that finds all the "for" loops and puts them inside an arraylist */
        private void getForLoops(Matcher forMatcher, Matcher commentsMatcher) {

                boolean forInsideComments = false;

                while (forMatcher.find()) {
                        // in order not to catch for loops that are inside comments
                        while (commentsMatcher.find()) {
                                if (forMatcher.start() > commentsMatcher.start()
                                                && forMatcher.end() <=
commentsMatcher.end()) {
                                        forInsideComments = true;
                                        break;
                                }
                        }
```

```
                        if (forInsideComments == false)
                                forString.add(forMatcher.group());

                        commentsMatcher.reset(); // reset the comment matcher in order to
search again
                                                                        // from the
beginning for comments
                        forInsideComments = false;

                }

        }

        /* Method that replaces all the "for" loops with a unique string that will be
later
         * be replaced by the original "for" loop */
        private String forReplacement(String testFile, Matcher forMatcher) {

                StringBuilder file = new StringBuilder();
                int counter = 0; // initialize counter for the "for" loops occurrences

                for (String line : testFile.split(LINE_SEPARATOR)) {

                        forMatcher = forPattern.matcher(line);

                        while (forMatcher.find()) {
                                line = line.replace(forMatcher.group(), FOR_PATENT +
String.valueOf(counter++));
                        }

                        file.append(line);
                        file.append(LINE_SEPARATOR);
                }

                return file.toString();

        }

        /* Method that finds all the string quotes and puts them inside an arraylist */
        private void getStringQuotes(Matcher stringMatcher, Matcher commentsMatcher) {

                boolean quotesInsideComments = false;

                while (stringMatcher.find()) {
                        // in order not to catch string quotes that are inside comments
                        while (commentsMatcher.find()) {
                                if (stringMatcher.start() > commentsMatcher.start()
                                                && stringMatcher.end() <=
commentsMatcher.end()) {
                                        quotesInsideComments = true;
                                        break;
                                }
                        }

                        if (quotesInsideComments == false)
                                quotedString.add(stringMatcher.group(1));

                        commentsMatcher.reset(); // reset the comment matcher in order to
search again
                                                                        // from the
beginning for comments
                        quotesInsideComments = false;

                }
        }

        /* Method that finds all the char quotes and puts them inside an arraylist */
        private void getCharQuotes(Matcher charMatcher, Matcher commentsMatcher) {

                boolean quotesInsideComments = false;

                while (charMatcher.find()) {
                        // in order not to catch char quotes that are inside comments
                        while (commentsMatcher.find()) {
                                if (charMatcher.start() > commentsMatcher.start()
```

```
                                        && charMatcher.end() <=
commentsMatcher.end()) {
                                quotesInsideComments = true;
                                break;
                        }
                }

                if (quotesInsideComments == false)
                        quotedChar.add(charMatcher.group(1));

                commentsMatcher.reset(); // reset the comment matcher in order to
search again
                                                                // from the
beginning for comments
                quotesInsideComments = false;

        }
    }

    /* Method that replaces all the string quotes with a unique string that will be
later
     * be replaced by the original string quotes */
    private String stringQuotesReplacement(String testFile, Matcher stringMatcher) {

        StringBuilder file = new StringBuilder();
        int counter = 0; // initialize counter for the string quotes occurrences

        for (String line : testFile.split(LINE_SEPARATOR)) {

                stringMatcher = STRING_QUOTES.matcher(line);

                while (stringMatcher.find()) {
                        line = line.replace(stringMatcher.group(), "\"myPatent"+
String.valueOf(counter++)+ "\"");
                }

                file.append(line);
                file.append(LINE_SEPARATOR);
        }

        return file.toString();

    }

    /* Method that replaces all the char quotes with a unique string that will be
later
     * be replaced by the original char quotes */
    private String charQuotesReplacement(String testFile, Matcher charMatcher) {

        StringBuilder file = new StringBuilder();
        int counter = 0; // initialize counter for the char quote occurrences

        for (String line : testFile.split(LINE_SEPARATOR)) {

                charMatcher = CHAR_QUOTES.matcher(line);

                while (charMatcher.find()) {
                        line = line.replace(charMatcher.group(), "'"+
String.valueOf(counter++)+ "'");

                }

                file.append(line);
                file.append(LINE_SEPARATOR);
        }

        return file.toString();

    }

    /* Method that resets all the replacements made from the above methods */
    private String resetReplacement(String testFile, Matcher stringMatcher, Matcher
charMatcher, Matcher forMatcher) {

        StringBuilder file = new StringBuilder();
```

```
		boolean firstLine = true;
		int i = 0; // ArrayList quotedString index
		int j = 0; // ArrayList quotedChar index
		int k = 0; // ArrayList forString index
		boolean flag;

		// replace the quotes with the original ones and also the for loops
		for (String line : testFile.split(LINE_SEPARATOR)) {

			stringMatcher = STRING_QUOTES.matcher(line); // STRING_QUOTES
pattern used here
			charMatcher = CHAR_QUOTES.matcher(line); // CHAR_QUOTES pattern
used here
			forMatcher = forPatent.matcher(line); // forPatent pattern used
here

			flag = false;

			if (line.isEmpty() || line.matches("\\s")) {
				; // do nothing -- eat up empty lines
				flag = true;
			}
			if (stringMatcher.find()) {
				line = line.replace(stringMatcher.group(1),
quotedString.get(i++));
				while (stringMatcher.find()) {
					line = line.replace(stringMatcher.group(1),
quotedString.get(i++));
				}
				file.append(line);
				file.append(LINE_SEPARATOR);
				flag = true;
			}
			if (charMatcher.find()) {
				line = line.replace(charMatcher.group(1),
quotedChar.get(j++));
				while (charMatcher.find()) {
					line = line.replace(charMatcher.group(1),
quotedChar.get(j++));
				}
				file.append(line);
				file.append(LINE_SEPARATOR);
				flag = true;
			}
			if (forMatcher.find()) {
				line = line.replace(forMatcher.group(),
forString.get(k++));
				while (forMatcher.find()) {
					line = line.replace(forMatcher.group(),
forString.get(k++));
				}
				file.append(line);
				file.append(LINE_SEPARATOR);
				flag = true;
			}
			if (flag == false) {
				if (firstLine == false) {
					file.append(line);
					file.append(LINE_SEPARATOR);
				}
				else { // (case firstLine true) if it is the first line of
the file, put the
					   // appropriate package name
					file.append(PACKAGE_NAME);
					file.append(LINE_SEPARATOR);
					firstLine = false;
				}
			}
		}

		return file.toString();

	}

	/* Method that returns a file being format by using the above methods */
```

```
	public String formatFile() {

		String testFile = readFile(fileName); // call method readFile

		Matcher stringMatcher;
		Matcher charMatcher;
		Matcher forMatcher;
		Matcher commentsMatcher;

		forMatcher = forPattern.matcher(testFile);
		commentsMatcher = comments.matcher(testFile);

		getForLoops(forMatcher, commentsMatcher); // call method getForLoops

		// "for" loop replacement(patent) must be done before finding the string
quotes
		// because the "for" loop might contain string quotes also but more
important is
		// that "for" loop contains ";" and that would split the loop in empty
lines,
		// when at the end after each ";", I place a new line
		testFile = forReplacement(testFile, forMatcher); // call method
forReplacement

		stringMatcher = STRING_QUOTES.matcher(testFile);

		getStringQuotes(stringMatcher, commentsMatcher); // call method
getStringQuotes

		charMatcher = CHAR_QUOTES.matcher(testFile);

		getCharQuotes(charMatcher, commentsMatcher); // call method getCharQuotes

		testFile = stringQuotesReplacement(testFile, stringMatcher); // call
method stringQuotesReplacement

		testFile = charQuotesReplacement(testFile, charMatcher); // call method
charQuotesReplacement

		// eat up comments, important to be done after the string quotes
		// replacement in order not to lose quotes than have the form of comments
		testFile = testFile.replaceAll(COMMENTS, "");

		testFile = testFile.replaceAll("\t|" + LINE_SEPARATOR, ""); // eat up new
lines and tabs
		testFile = testFile.replaceAll(";", ";" + LINE_SEPARATOR); // put after
each ";" new line character
		testFile = testFile.replaceAll("\\{", "\\{" + LINE_SEPARATOR); // put
after each "{" new line character
		testFile = testFile.replaceAll("\\}", "\\}" + LINE_SEPARATOR); // put
after each "}" new line character

		testFile = resetReplacement(testFile, stringMatcher, charMatcher,
forMatcher); // call method resetReplacement

		return testFile;

	}

}
```

HighlightLines.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
```

```
import static myproject.Constants.*;
import static myproject.utilities.TextFile.*;
import java.util.*;

/* Class responsible for producing an html file */
public class HighlightLines {

		/* Method that creates an html file with highlighted lines based on
variables' lines found */
		public String createHtmlFile (String fileName, HashMap<Integer, String>
varLines) {

			int lineIndex = 0;
			boolean highlighted = false;
			StringBuilder htmlFile = new StringBuilder();
			String testFile = readFile(FILES_DIR + fileName + JAVA_EXT); //
call method readFile

			htmlFile.append(START_TAGS);
			for (String line : testFile.split(LINE_SEPARATOR)) {

				++lineIndex;

				if(varLines.containsKey(lineIndex) == true) { // case line
must be highlighted
					htmlFile.append(HIGHLIGHT_TAG);
					highlighted = true;
				}
				htmlFile.append(lineIndex);
				htmlFile.append("\t");
				htmlFile.append(line);

				if(highlighted == true) { // case line was highlighted
					htmlFile.append(END_HIGHLIGHT_TAG);
					highlighted = false;
				}

				htmlFile.append(LINE_SEPARATOR);

			}
			htmlFile.append(END_TAGS);

			return htmlFile.toString();

		}

}
```

Loops.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
import static myproject.Scope.*;
import java.util.*;
import java.util.regex.*;

/* Class responsible for finding scopes of java loops and getting their variables */
public class Loops {

	/* Method that gets the variables of a certain kind of loop inside the file given
*/
```

```
	public void getLoopsVars(String fileName, Pattern loopPattern, HashMap<Integer,
String> varLines,
						LinkedHashSet<String> variables, int
userChoice) {

		int lineIndex = 0;
		boolean loopScope = false;
		int countCurlyBrackets = 0;
		int outerLoopsCounter = 1;
		int innerLoopsCounter = -1;

		String testFile = readFile(FILES_DIR + fileName + JAVA_EXT); // call
method readFile

		testFile = ParserTool.neutralizeQuotes(testFile); // call method
neutralizeQuotes

		Matcher loopMatcher = null;

		for (String line : testFile.split(LINE_SEPARATOR)) {

			++lineIndex;
			loopMatcher = loopPattern.matcher(line);

			// in order to get the last variable of a "do-while" loop
			if( userChoice == ParserTool.DO_WHILE_CHOICE &&
line.matches(DO_WHILE_END)
					&& countCurlyBrackets == 0) {
				// call method getVariables
				ParserTool.getVariables(lineIndex, line, varLines,
variables, userChoice);
			}

			if (loopMatcher.find()) {
				loopScope = true;
				++innerLoopsCounter;
				if (userChoice == ParserTool.FOR_CHOICE) {
					if (innerLoopsCounter == 0) { // case no inner loop
found yet
						System.out.printf("\n===== For Loop %d
=====\n", outerLoopsCounter);
					} else {
						System.out.printf("\n===== For Loop %d.%d
=====\n", outerLoopsCounter,innerLoopsCounter);
					}
				} else if (userChoice == ParserTool.FOREACH_CHOICE) {
					if (innerLoopsCounter == 0) { // case no inner loop
found yet
						System.out.printf("\n===== ForEach Loop %d
=====\n", outerLoopsCounter);
					} else {
						System.out.printf("\n===== ForEach Loop
%d.%d =====\n", outerLoopsCounter,innerLoopsCounter);
					}
				} else if (userChoice == ParserTool.WHILE_CHOICE) {
					if (innerLoopsCounter == 0) { // case no inner loop
found yet
						System.out.printf("\n===== While Loop %d
=====\n", outerLoopsCounter);
					} else {
						System.out.printf("\n===== While Loop %d.%d
=====\n", outerLoopsCounter,innerLoopsCounter);
					}
				} else if (userChoice == ParserTool.DO_WHILE_CHOICE) {
					if (innerLoopsCounter == 0) { // case no inner loop
found yet
						System.out.printf("\n===== Do-While Loop %d
=====\n", outerLoopsCounter);
					} else {
						System.out.printf("\n===== Do-While Loop
%d.%d =====\n", outerLoopsCounter,innerLoopsCounter);
					}
				}

			}
```

```
                        if (loopScope == true) {
                                // call method getVariables
                                ParserTool.getVariables(lineIndex, line, varLines,
variables, userChoice);
                                countCurlyBrackets = getScope(line, countCurlyBrackets);
// call method getScope
                        }

                        if (loopScope == true && countCurlyBrackets == 0) {
                                loopScope = false;
                                innerLoopsCounter = -1;
                                ++outerLoopsCounter;
                        }

                }

        }

}
```

Methods.java

```
/* * * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.ReadInput.*;
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
import static myproject.Scope.*;
import java.util.*;
import java.util.regex.*;

/* Class responsible for getting the variables from a method */
public class Methods {

        /* Method that gets the methods' names of a file given */
        public LinkedHashSet<String> getMethodsName(String fileName,
LinkedHashSet<String> methodsNames) {

                String testFile = readFile(FILES_DIR + fileName + JAVA_EXT); // call
method readFile

                testFile = ParserTool.neutralizeQuotes(testFile); // call method
neutralizeQuotes

                Matcher methodMatcher = null;

                for (String line : testFile.split(LINE_SEPARATOR)) {

                        methodMatcher = METHODS.matcher(line);

                        if (methodMatcher.find()) {
                                // in case some reserved
word(else,public,protected,private)tricks the
                                // regular expression to be the returned type of a method,
then skip them
                                if (methodMatcher.group(5).matches(RESERVED_WORD) ==
false) {
                                        methodsNames.add(methodMatcher.group(12));
                                }

                        }
                }
```

```
                return methodsNames;

        }

        /* Method that displays the methods' menu */
        public int menu(LinkedHashSet<String> methodsNames) {

         int choice; // user's choice
         int i = 0;

         System.out.println("\n=======================================");
         System.out.println("Select the method you want to use:");
         System.out.println("=======================================\n");
         for(String name : methodsNames) {
             System.out.printf("%d. Method name : %s\n", i + 1, name);
             ++i;
         }
         System.out.printf("%d. Return to load files menu\n\n", i+1);

         choice = readInt(1, i + 1); // call method readInt

         return (choice == i + 1) ? -1 : choice - 1; // case choice is to return to
previous menu return "-1"

    }

        /* Method that gets the variables of a method */
        public void getMethodVars(String fileName, String methodName, HashMap<Integer,
String> varLines,
                                                        LinkedHashSet<String> variables) {

                int lineIndex = 0;
                boolean methodScope = false;
                int countCurlyBrackets = 0;
                HashMap<Integer, String> tmp = new HashMap<Integer, String>();

                String testFile = readFile(FILES_DIR + fileName + JAVA_EXT); // call
method readFile

                testFile = ParserTool.neutralizeQuotes(testFile); // call method
neutralizeQuotes

                Matcher methodMatcher = null;

                for (String line : testFile.split(LINE_SEPARATOR)) {

                        ++lineIndex;

                        methodMatcher = METHODS.matcher(line);

                        if (methodMatcher.find()) {

                                // in case some reserved
word(else,public,protected,private)tricks the
                                // regular expression to be the returned type of a method,
then skip them
                                if (methodMatcher.group(5).matches(RESERVED_WORD) ==
false) {

                                        if (methodName.equals(methodMatcher.group(12))) {
// case the name of a method matches

                                                methodScope = true;
                                                methodMatcher.reset(); // reset the regular
expression
                                                                                        //
matcher in order to use it again
                                        }
                                }

                        }

                        // in order to avoid to get the line of method declaration, I
check
                        // first to see if this line contains a method declaration
```

```
                        if (methodScope == true && methodMatcher.find() == false) {
                                // call method getVariables
                                ParserTool.getVariables(lineIndex, line, varLines,
variables, ParserTool.METHOD_CHOICE);
                        }

                        if (methodScope == true) {
                                countCurlyBrackets = getScope(line, countCurlyBrackets);
// call method getScope
                        }

                        if (methodScope == true && countCurlyBrackets == 0) {

                                methodScope = false;
                                System.out.println("\n===== Found Method : " + methodName
+ " =====");
                                Methods.printLines(varLines, ParserTool.METHOD_CHOICE); //
call method printLines
                                tmp.putAll(varLines);
                                varLines.clear(); // empty varLines

                        }

                }

                varLines.putAll(tmp);

        }

        /* Method that displays the lines and variables in case of recursion, modulus
operators and methods */
        @SuppressWarnings("unchecked")
        public static void printLines(HashMap<Integer, String> varLines, int userChoice)
{

                // in order to sort the HashMap by key(lines of the file), I pass it to a
TreeMap
                TreeMap<Integer, String> variableLines = new TreeMap<Integer,
String>(varLines);

                Iterator iterator = variableLines.entrySet().iterator();

                if (variableLines.isEmpty() == false) {
                        while (iterator.hasNext()) {
                                Map.Entry pairs = (Map.Entry) iterator.next();
                                System.out.print("Line: " + pairs.getKey());
                                System.out.println(" Vars: " + pairs.getValue());
                        }
                } else { // case no variables found (empty variableLines )
                        if (userChoice == ParserTool.METHOD_CHOICE ) {
                                System.out.println("No variables were found in the scope
of this method!");
                        } else if (userChoice == ParserTool.RECURSION_CHOICE ) { // case
RECURSION_CHOICE
                                System.out.println("No variables were found in the scope
of this recursive method!");
                        } else { // case MODULUS_CHOICE
                                System.out.println("No modulus operator was found in this
file, in order to\n" +
                                                                        "get the involving
variables in the equation!");
                        }
                }

        }

}
```

Modulus.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
import java.util.*;
import java.util.regex.*;

/* Class responsible for getting the variables from an equation containing modulus
operator inside */
public class Modulus {

	/* Method that gets the variables involved inside an equation that uses modulus
operator */
	public void getModulusOpVars(String fileName, HashMap<Integer, String> varLines,
					LinkedHashSet<String> variables)  {

		int lineIndex = 0;

		String testFile = readFile(FILES_DIR + fileName + JAVA_EXT); // call
method readFile

		testFile = ParserTool.neutralizeQuotes(testFile); // call method
neutralizeQuotes

		Matcher moduluMatcher = null;

		for (String line : testFile.split(LINE_SEPARATOR)) {

			++lineIndex;
			moduluMatcher = MODULUS_OP.matcher(line);

			if (moduluMatcher.find()) {

				// call method getVariables
				ParserTool.getVariables(lineIndex, line, varLines,
variables, ParserTool.MODULUS_CHOICE);

			}

		}

	}

}
```

ParserTool.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.ReadInput.*;
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
```

```
import java.util.*;
import java.util.regex.*;

/* Class responsible for running all the parser tool functions */
public class ParserTool {

	/* constants of user's menu choice */
	public static final int FOR_CHOICE = 1;
	public static final int FOREACH_CHOICE = 2;
	public static final int WHILE_CHOICE = 3;
	public static final int DO_WHILE_CHOICE = 4;
	public static final int RECURSION_CHOICE = 5;
	public static final int MODULUS_CHOICE = 6;
	public static final int METHOD_CHOICE = 7;

	/* lists used to keep variables and files through out the program implementation
	*/
	private static ArrayList<String> files = new ArrayList<String>();
	private static LinkedHashSet<String> variables = new LinkedHashSet<String>();
	private static LinkedHashSet<String> methodsNames = new LinkedHashSet<String>();
	private static HashMap<Integer, String> varLines = new HashMap<Integer,
	String>();

	/* Method that checks if no variables where found in case of loops & recursion
	methods */
	void checkIfVarsNotFound(HashMap<Integer, String> varLines, int choice) {

		if (varLines.isEmpty() == true ) {
			if (choice == FOR_CHOICE || choice == FOREACH_CHOICE) {
				System.out.println("\nNo variables were found in \"for\"
	loops' scope of this file!");
			} else if (choice == WHILE_CHOICE || choice == DO_WHILE_CHOICE) {
				System.out.println("\nNo variables were found in \"while\"
	loops' scope of this file!");
			} else if (choice == RECURSION_CHOICE) {
				System.out.println("\nNo recursion methods' variables were
	found in the scope of this file!");
			}
		}
	}

	/* Method that gets the variables from a line given and puts them in the given
	hash map */
	public static void getVariables(int lineNumber, String line, HashMap<Integer,
	String> varLines,
					LinkedHashSet<String> variables, int
	choice) {

		boolean foundVars = false;
		// VAR_NAME regular expression is used here
		Matcher var = Pattern.compile(VAR_NAME).matcher(line);
		StringBuilder varString = new StringBuilder();

		while (var.find()) {
			if (variables.contains(var.group(1))) {
				varString.append(var.group(1) + ",");
				foundVars = true;
			}
		}

		if (foundVars == true) {
			varString.deleteCharAt(varString.length() - 1); // in order to
	delete extra comma
			varLines.put(lineNumber, varString.toString());
			// case it is a loop, print available info
			if (choice != MODULUS_CHOICE && choice != RECURSION_CHOICE &&
	choice != METHOD_CHOICE) {
				System.out.print("Line: " + lineNumber);
				System.out.println(" Vars: " + varString.toString());
			}

		}

	}
```

```
    /* Method that neutralizes all quotes inside the test file in order not to
    mislead the regular
     * expressions  */
    public static String neutralizeQuotes (String testFile) {

           // replace all string and char in order not to interfere with the
           // regular expression matching of specific functionalities
           testFile = STRING_QUOTES.matcher(testFile).replaceAll("\"myPatent\"");
           testFile = CHAR_QUOTES.matcher(testFile).replaceAll("'!'");

           return testFile;
    }

    /* Method that displays the files' menu */
    public int loadFileMenu(int numOfChoices) {

     int choice; // user's choice
     System.out.println("\n======================================");
     System.out.println("Select the test file you want to use:");
     System.out.println("======================================\n");
     for (int i = 0; i < numOfChoices; i++) {
         System.out.printf("%d. Load file %s.java\n", i + 1, files.get(i) );
     }

     System.out.printf("%d. Exit\n\n", numOfChoices + 1);

     choice = readInt(1, numOfChoices + 1); // call method readInt

     if (choice == numOfChoices + 1) {
         System.out.println("\nExiting...");
         System.exit(0); // exit the system
     }

     return choice - 1;

}

    /* Method that displays the "time consuming code parts" menu */
    public int menu() {

                   int choice; // user's choice

    System.out.println("\n============================================================
    ===");
            System.out.println("Select what time consuming code parts you want to
    check out:");

    System.out.println("============================================================
    =\n");
            System.out.println("1. For loops ");
            System.out.println("2. While loops");
            System.out.println("3. Recursions");
            System.out.println("4. Modulus Operator");
            System.out.println("5. Choose a method");
            System.out.println("6. Return to load files menu\n");

            choice = readInt(1, 6); // call method readInt

            return choice ;
    }

    /* M A I N - M E T H O D */
    public static void main(String[] args) {

           ParserTool parser = new ParserTool();

           // must be specified the working directory of eclipse first as the scr
           DirectoryList list = new DirectoryList();
           int choice;
           int menuChoice;

           files = list.getFilesName(TEST_DIR, FILE_PATTERN); // call method
    getFilesName

           if ( files.size() == 0 ) {
```

```
                System.out.println("\nYou need to import some test files in the "
+
                                "testing package inside Eclipse package explorer
first!");
                System.out.println("\nExiting...");
                System.exit(0); // exit the system
        }

        // declare object used in do-while
        Formatter formatter;
        String testFile;
        Variables var;
        Loops forLoop;
        Loops whileLoop;
        Recursions recursion;
        Modulus mod;
        Methods methods;
        HighlightLines highlight;
        Object[] methodName;
        int methodChoice;

        do {

                choice = parser.loadFileMenu(files.size()); // call method
loadFileMenu

                formatter = new Formatter(TEST_DIR + files.get(choice)+ JAVA_EXT);
                testFile = formatter.formatFile(); // call method formatFile

                // file must be written before making use of Variable class
                writeFile(FILES_DIR + files.get(choice)+ JAVA_EXT , testFile); //
call method writeFile

                var = new Variables();
                variables = var.getVariablesList(files.get(choice)); // call
method getVariablesList
                //System.out.println("VARIABLES: " + variables);

                menuChoice = parser.menu(); // call method menu


        System.out.println("\n=================================================
====================");

                switch (menuChoice) {

                        case 1: // for loop
                                forLoop = new Loops();
                                // call method getLoopsVars
                                forLoop.getLoopsVars(files.get(choice), FOR,
varLines, variables,FOR_CHOICE);
                                forLoop.getLoopsVars(files.get(choice), FOR_EACH,
varLines, variables,FOREACH_CHOICE);
                                parser.checkIfVarsNotFound(varLines,FOR_CHOICE); //
call method checkIfVarsNotFound
                                break;
                        case 2: // while loops
                                whileLoop = new Loops();
                                // call method getLoopsVars
                                whileLoop.getLoopsVars(files.get(choice), WHILE,
varLines, variables,WHILE_CHOICE);
                                whileLoop.getLoopsVars(files.get(choice), DO_WHILE,
varLines, variables,DO_WHILE_CHOICE );
                                parser.checkIfVarsNotFound(varLines,WHILE_CHOICE);
// call method checkIfVarsNotFound
                                break;
                        case 3: // recursion methods
                                recursion = new Recursions();
                                // call method getRecursionMethodVars

        recursion.getRecursionMethodVars(files.get(choice),varLines,variables);

        parser.checkIfVarsNotFound(varLines,RECURSION_CHOICE); // call method
checkIfVarsNotFound
                                break;
```

```
                        case 4: // modulus operator
                                mod = new Modulus();
                                // call method getModulusOpVars
                                mod.getModulusOpVars(files.get(choice),
        varLines, variables);
                                System.out.println();
                                Methods.printLines(varLines,
        MODULUS_CHOICE); // call method printLines
                                break;
                        case 5: // methods
                                methods = new Methods();
                                // call method getMethodsName
                                methodsNames =
        methods.getMethodsName(files.get(choice), methodsNames);
                                methodName = methodsNames.toArray();
                                if (methodsNames.size() != 0 ) {
                                        methodChoice =
        methods.menu(methodsNames); // call method menu
                                        if ( methodChoice != -1 ) { // case
        a method was chosen

                System.out.println("\n=============================================
        ====================");
                                                        // call method
        getMethodsName

                methods.getMethodVars(files.get(choice),methodName[methodChoice].toString
        (),

                varLines,variables);

                                        }
                                } else {
                                        System.out.println("\nNo methods
        where found in this file!");
                                }
                                break;
                        default:
                                break;

                }


                System.out.println("\n=============================================
        ====================");

                highlight = new HighlightLines();
                testFile = highlight.createHtmlFile(files.get(choice), varLines);
        // call method createHtmlFile
                writeFile(HTML_DIR + files.get(choice)+ HTML_EXT, testFile); //
        call method writeFile

                variables.clear(); // clear variables list, in order to take new
        elements inside
                methodsNames.clear(); // clear methods list, in order to take new
        elements inside
                varLines.clear(); // clear varLines variables list, in order to
        take new elements inside


        } while(true);

        }

}
```

Recursions.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
import static myproject.Scope.*;
import java.util.*;
import java.util.regex.*;

/* Class responsible for getting the variables from a recursion method */
public class Recursions {

	/* Method that gets the variables of a recursion method */
	public void getRecursionMethodVars(String fileName, HashMap<Integer, String>
varLines,

	LinkedHashSet<String> variables) {

		int lineIndex = 0;
		boolean methodScope = false;
		boolean recursionFound = false;
		int countCurlyBrackets = 0;
		HashMap<Integer, String> tmp = new HashMap<Integer, String>();

		String testFile = readFile(FILES_DIR + fileName + JAVA_EXT); // call
method readFile

		testFile = ParserTool.neutralizeQuotes(testFile); // call method
neutralizeQuotes

		Matcher methodMatcher = null;
		Matcher methodCallMatcher = null;
		String methodName = "";

		for (String line : testFile.split(LINE_SEPARATOR)) {

			++lineIndex;

			methodMatcher = METHODS.matcher(line);

			if (methodMatcher.find()) {

				// in case some reserved
word(else,public,protected,private)tricks the
				// regular expression to be the returned type of a method,
then skip them
				if (methodMatcher.group(5).matches(RESERVED_WORD) ==
false) {
					methodName = methodMatcher.group(12);
					methodScope = true;
					methodMatcher.reset(); // reset the regular
expression matcher in order to
												// use it
again in the if statement below
				}

			}

			// in order to avoid to get the line of method declaration, I
check
			// first to see if this line contains a method declaration
			if (methodScope == true && methodMatcher.find() == false) {
```

```
                                methodCallMatcher = Pattern.compile(methodName +
PARENTHESIS).matcher(line);

                                if (methodCallMatcher.find()) {
                                        recursionFound = true;
                                }

                                // call method getVariables
                                ParserTool.getVariables(lineIndex, line, varLines,
variables, ParserTool.RECURSION_CHOICE);
                        }

                        if (methodScope == true) {
                                countCurlyBrackets = getScope(line, countCurlyBrackets);
// call method getScope
                        }

                        // case where the end of a method's scope has been reached
                        if (methodScope == true && countCurlyBrackets == 0) {

                                methodScope = false;

                                if (recursionFound == true) {

                                        recursionFound = false; // case recursionFound is
true, turn it to false

                                        System.out.println("\n===== Found Recursive Method
: " + methodName + " =====");
                                        Methods.printLines(varLines,
ParserTool.RECURSION_CHOICE); // call method printLines
                                        tmp.putAll(varLines);

                                }

                                varLines.clear(); // empty varLines
                        }

                }

                varLines.putAll(tmp);

        }

}
```

Scope.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.Constants.*;
import java.util.regex.*;

/* Class responsible for keeping track of a particular scope given */
public class Scope {

        /* Method that calculates whether the end scope has been reached by incrementing
the left
         * curly bracket occurrences and decrementing the right curly bracket occurrences
and returning
         * that counter */
        public static int getScope(String line, int countCurlyBrackets) {

                Matcher curlyBracket = CURLY_BRACKETS.matcher(line);
```

```
                if (curlyBracket.find()) {
                        if (curlyBracket.group().equals("{")) {
                                ++countCurlyBrackets;
                        }
                        if (curlyBracket.group().equals("}")) {
                                --countCurlyBrackets;
                        }
                }
                return countCurlyBrackets;
        }

}
```

Variables.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael        *
 * Student Id: 886077                         *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject;

/* Import Packages */
import static myproject.utilities.TextFile.*;
import static myproject.Constants.*;
import java.lang.reflect.*;
import java.util.*;
import java.util.regex.*;

/* Class responsible for parsing a file, in order to get all each variables
 * (it also gets constants and object handles) */
public class Variables {

        /* regular expressions patterns used */
        private Pattern qualifedName = Pattern.compile(QUALIFED_NAME);
        private Pattern varDeclaration = Pattern.compile(MODIFIERS + "(" + VAR_TYPE
        + GENERICS + BRACKETS + ")"

                + VAR_NAME + MORE_VARS);
        private Pattern varAfterComma = Pattern.compile(SEPARATOR);
        private Pattern varParameters = Pattern.compile(MODIFIERS + "(" + VAR_TYPE +
GENERICS + BRACKETS + ")"

                + VAR_NAME + COMMA);
        private Pattern forEachVar = Pattern.compile(MODIFIERS + "(" + VAR_TYPE    +
GENERICS + BRACKETS + ")"

        + VAR_NAME);

        private Pattern exceptionVar = Pattern.compile(MODIFIERS + VAR_TYPE + VAR_NAME);

        /* lists to hold the variables and the constructor name */
        private LinkedHashSet<String> variables = new LinkedHashSet<String>();
        private LinkedHashSet<String> constructorName = new LinkedHashSet<String>();

        /* Method for finding all the constructors inside the .java file given */
        private void findConstructors(Class<?> c) {

                Constructor<?>[] constructors = c.getDeclaredConstructors();
                String name;

                for (Constructor<?> constructor : constructors) {
                        // in order to eat up the qualified name of the constructor's name
                        name =
qualifedName.matcher(constructor.toString()).replaceAll("");
                        constructorName.add(name.replaceAll(PARENTHESIS, ""));

                }

        }
```

```
	/* Method for getting all the parameters variables */
	private void getParameterVars(String testFile, Matcher methodMatcher, Matcher
params) {

		// get variables from method parameters
		for (String line : testFile.split(LINE_SEPARATOR)) {

			methodMatcher = METHODS.matcher(line);

			if (methodMatcher.find()) {
				// case reserved word "else" is consider as returned type
of a method
				if ( !methodMatcher.group(5).matches("\\s*else\\s*") ) {
					params =
varParameters.matcher(methodMatcher.group(13));
					if (!methodMatcher.group(13).isEmpty()) { // method
without parameters
						while (params.find()) {
							variables.add(params.group(12));
						}
					}
				}

			}
		}
	}

	/* Method for getting all the constructors' variables */
	private void getConstructorVars(String testFile,Matcher constructorMatcher,
Matcher params) {

		for (String name : constructorName) {

			constructorMatcher = Pattern.compile(name +
PARENTHESIS).matcher(testFile);

			// I need while because constructor's name of a class has the same
name
			while (constructorMatcher.find()) {
				params =
varParameters.matcher(constructorMatcher.group(1));
				if (!constructorMatcher.group(1).isEmpty()) { // method
without parameters
					while (params.find()) {
						variables.add(params.group(12));
					}
				}
			}
		}
	}

	/* Method for getting all the "for" loops variables */
	private void getForLoopVars(String testFile, Matcher forMatcher, Matcher
varDeclared, Matcher varName) {

		// get variables from "for" loop
		while (forMatcher.find()) {

			varDeclared = varDeclaration.matcher(forMatcher.group(1));
			varName = varAfterComma.matcher(forMatcher.group(1));
			if (varDeclared.matches()) {

				variables.add(varDeclared.group(12));

				while (varName.find()) {
					variables.add(varName.group(1));
				}
			}
		}

	}

	/* Method for getting all the "foreach" loops variables */
	private void getForEachVars(Matcher forEachMatcher, Matcher varName) {
```

```
		// get variables from "foreach" loop
		while (forEachMatcher.find()) {

			varName = forEachVar.matcher(forEachMatcher.group(1));

			if (varName.matches()) {
				variables.add(varName.group(12));
			}

		}
	}

	/* Method for getting all the exception catch clause variables */
	private void getCatchClauseVars(Matcher catchClauseMatcher, Matcher varName) {

		// get variables from exception catch clause
		while (catchClauseMatcher.find()) {

			varName = exceptionVar.matcher(catchClauseMatcher.group(1));

			if (varName.matches()) {
				variables.add(varName.group(6));
			}

		}
	}

	/* Method for getting all the fields and method's local variables */
	private void getFieldsAndLocalVars(String testFile, Matcher varDeclared, Matcher
varName) {

		// get variables from fields and method's local variables
		for (String line : testFile.split(LINE_SEPARATOR)) {

			varDeclared = varDeclaration.matcher(line);
			varName = varAfterComma.matcher(line);

			if (varDeclared.matches()) {
				// in case of simple package declaration eg. "package
testing;"
				// and not "package testing.files;", will be consider by
the
				// regular expression as variable, also in case of eg.
"return 1;"
				// that would also be consider as variable, the "1"
				// So, if the variable type is "package" or the variable
name contains only
				// numbers, then skip it
				if (!varDeclared.group(5).matches("\\s*package\\s*")
						&& !varDeclared.group(12).matches("\\d+"))
{
					variables.add(varDeclared.group(12));
				}

				while (varName.find()) {
					variables.add(varName.group(1));
				}
			}
		}
	}

	/* Method that returns a list containing all the variables of the file given */
	public LinkedHashSet<String> getVariablesList(String fileName) {

		String testFile = readFile(FILES_DIR + fileName + JAVA_EXT);
		Matcher varDeclared = null;
		Matcher varName = null;
		Matcher methodMatcher = null;
		Matcher constructorMatcher = null;
		Matcher params = null;
		Matcher forMatcher = null;
		Matcher forEachMatcher = null;
		Matcher catchClauseMatcher = null;
```

```
                Class<?> className = null;
                try {
                        className = Class.forName(QUALIFIED_DIR + fileName);
                } catch (ClassNotFoundException e) {
                        System.out.println("No such class: " + e);
                }

                testFile = ParserTool.neutralizeQuotes(testFile); // call method
neutralizeQuotes

                findConstructors(className); // call method findConstructors

                getConstructorVars(testFile, constructorMatcher, params);  // call method
getConstructorVars

                catchClauseMatcher = CATCH_CLAUSE.matcher(testFile);

                getCatchClauseVars(catchClauseMatcher, varName); // call method
getCatchClauseVars

                forMatcher = FOR.matcher(testFile);

                getForLoopVars(testFile, forMatcher, varDeclared, varName); // call
method getForLoopVars

                forEachMatcher = FOR_EACH.matcher(testFile);

                getForEachVars(forEachMatcher, varName); // call method getForEachVars

                getParameterVars(testFile, methodMatcher, params); // call method
getParameterVars

                getFieldsAndLocalVars(testFile, varDeclared, varName); // call method
getFieldsAndLocalVars

                return variables;
        }

}
```

ReadInput.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject.utilities;

/* Import Packages */
import java.io.*;

/* Class responsible for reading input */
public class ReadInput {

    /* Method for reading an integer number within a given range
     and ensure that nothing else exceeding the given boundaries will be given */
    public static int readInt(int from, int to) {
        //  open up standard input
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String input = null;
        boolean isInt;
        int num = 0;

        try {
            do {
                isInt = true;

                System.out.printf("Enter your choice (Must be from %d to %d) : ", from,
to);
```

```
                input = br.readLine();
                try {
                    num = Integer.parseInt(input);
                } catch (NumberFormatException e) {
                    System.out.println("Invalid input!");
                    isInt = false;
                }
            } while (isInt == false || num < from || num > to);
        } catch (IOException e) {
            System.out.println("IO error while trying to read your input!");
            System.exit(1);
        }

        return num;

    }
}
```

TextFile.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                 *
\* * * * * * * * * * * * * * * * * * * * * * */

/* Package Name */
package myproject.utilities;

/* Import Packages */
import java.io.*;

/* This is a class that offers methods for reading,writing and counting the lines of a
file */
/* This is just a utility class that could be part of any program
*/
public class TextFile {

    // get the line separator of your operating system as a string constant
    public static final String LINE_SEPARATOR = System.getProperty("line.separator");

    /* Method that reads data from a file and returns them as a string format */
    public static String readFile(String fileName) {
        StringBuilder sb = new StringBuilder();

        try {
            BufferedReader input = new BufferedReader(new FileReader(fileName) );
            try {
                String line = null;

                while ((line = input.readLine()) != null) {
                    sb.append(line);
                    sb.append(LINE_SEPARATOR);
                }
            } catch (Exception e) {
                throw new RuntimeException();
            } finally {
                input.close();
            }
        } catch (Exception e) {
            System.err.println("Wrong File Name Or File Doesn't Exist!");
            System.exit(0);
        }

        return sb.toString();

    }

    /* Method that writes the desired text to a file */
    public static void writeFile(String fileName, String text) {
        try {
            BufferedWriter output = new BufferedWriter(new FileWriter(fileName) );
```

```
            try {
                output.write(text);
            } catch (Exception e) {
                throw new RuntimeException();
            } finally {
                output.close();
            }
        } catch (Exception e) {
            System.err.println("Wrong File Name Or File Doesn't Exist!");
        }
    }

}
```

Παράρτημα Β

Μερικά από τα αρχεία ελέγχου (testing files) του συντακτικού αναλυτή

ForLoops.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

package testing; /* package declaration */

/* import packages */
import java.util.Random;

// for loop examples
public class ForLoops {

	public static void main(String[] args) {

		Random rand = new Random(47);
		float[] f = new float[10];
		int k = 0;

		for (int i = 1; i <= 5; i++) {
			for (int j = 1; j <= i; j++) {
				System.out.print(i);
			}
			System.out.println();
		}

		for (int i = 0; i < 10; i++) {
			for (int j = 0; j < 10; j++) {
				for (int m = 0; m < 10; m++) {
					for (int n = 0; n < 10; n++) {
						System.out.println(n*m*j*i);
					}
				}
			}
		}

		for (int i = 0; i < 10; i++) {
			f[i] = rand.nextFloat();

		}

		for (float x : f) {
			System.out.println(x);
			for (float g2 : f) {
				System.out.println(g2);
			}
		}

		for (;;) {
			System.out.println(k++);
		}
	}

}
```

ModulusOperatorExample.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

package testing;

/*
Modulus Operator Example
This example shows how to use Java modulus operator (%).
 */

public class ModulusOperatorExample {

	public static void main(String[] args) throws Exception {

		String str = "Java Modulus Operator(%%) example";
		System.out.println(str);

		/*
		 * Modulus operator returns reminder of the devision of
		 * floating point or integer types.
		 */

		int i = 50;

		double d = 32;

		System.out.println("i mod 10 = " + i % 10);

		System.out.println("d mod 10 = " + d % 10);

	}

}

/*
 * Output would be
 * Java Modulus Operator example
 * i mod 10 = 0
 * d mod 10 = 2.0
 */
```

RecursionSample.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

package testing;

public class RecursionSample {

        // recursive factorial method
    public int factorial (int n)
    {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial (n - 1);
        }
    }

    // non-recursive fibonacci method
    public int fibonacci (int n)
    {
        int previous = -1;
        int result = 1;
        for (int i = 0; i <= n; )
        {
            int sum = result + previous;
            previous = result;
            result = sum;
            i = i + 1;
        }
        return result;
    }

    // recursive fibonacci method
    public static long fib(int n) {
        if (n <= 1) {
               return n;
        }
        else {
               return fib(n-1) + fib(n-2);
        }
    }

    public static void main(String[] args) {

        int num = Integer.parseInt(args[0]);

        for (int i = 1; i <= num; i++) {
               System.out.println(i + ": " + fib(i));
        }
    }

}
```

WhileLoops.java

```
/* * * * * * * * * * * * * * * * * * * * * *\
 * Author       : Christodoulos Michael         *
 * Student Id: 886077                          *
 * Program   : Senior Project                  *
\* * * * * * * * * * * * * * * * * * * * * * */

package testing;

public class WhileLoops {

	static boolean condition() {

		boolean result = Math.random() < 0.99;
		System.out.print(result + ", ");
		return result;

	}

	public static void main(String[] args) {

		/*
		 * Do while loop executes statement until certain condition become
		 * false. Syntax of do while loop is do <loop body> while(<condition>);
		 *
		 * where <condition> is a boolean expression.
		 *
		 * Please note that the condition is evaluated after executing the loop
		 * body. So loop will be executed at least once even if the condition is
		 * false.
		 */

		int i = 0; // initialization

		do {
			System.out.println("i is : " + i);
			i++;

		} while (i < 5);

		int n = 0;
		int m = 0;
		i = 0;

		while (n < 5) {
			System.out.println(++n);
			while (m < 5) {
				System.out.println(++m);

			}
		}
		System.out.println("Exited 'while'");

		while(true) {
			int j = 0;
			System.out.println("Infinite While");
			System.out.println(j);
		}
	}

}
```

Παράρτημα Γ

Αρχεία Benchmark

ArrayListInsertion.java

```
package testfiles;

import java.util.*;

public class ArrayListInsertion {

    public static void main(String args[]) {

        final int N = 50000;

        ArrayList al = new ArrayList();

        for (int i = 1; i <= N; i++) {
            al.add(0, new Integer(i));
        }

    }
}
```

ArrayListRandomAccess.java

```
package testfiles;

import java.util.*;

public class ArrayListRandomAccess {

    public static void main(String args[]) {

        final int N = 50000;
        Object o;

        ArrayList al = new ArrayList();
        for (int i = 0; i < N; i++) {
            al.add(new Integer(i));
        }

        for (int i = 0; i < N; i++) {
            o = al.get(i);
        }

    }
}
```

BinarySearch.java

```
package testfiles;
/** Binary search of sorted array.  Negative value on search failure.
 *  The upperbound index is not included in the search.
 *  This is to be consistent with the way Java in general expresses ranges.
 *  The performance is O(log N).
 *  @param sorted Array of sorted values to be searched.
 *  @param first Index of beginning of range.  First element is a[first].
 *  @param upto  Last element that is searched is sorted[upto-1].
 *  @param key   Value that is being looked for.
 *  @return      Returns index of the first match, or or -insertion_position
 *               -1 if key is not in the array. This value can easily be
 *               transformed into the position to insert it.
 */

public class BinarySearch {

    public int binarySearch(int[] sorted, int first, int upto, int key) {

        int mid;
        while (first < upto) {
            mid = (first + upto) / 2;  // Compute mid point.
            if (key < sorted[mid]) {
                upto = mid;     // repeat search in bottom half.
            } else if (key > sorted[mid]) {
                first = mid + 1;  // Repeat search in top half.
            } else {
                return mid;     // Found it. return position
            }
        }
        return -(first + 1);    // Failed to find key
    }

    public static void main(String[] args) {

        int[] array = new int[10000000];
        BinarySearch search = new BinarySearch();

        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }

        System.out.println(search.binarySearch(array, 0,  array.length, 5000000));
    }

}
```

BubbleSort.java

```
package testfiles;

public class BubbleSort {

    public static void bubbleSrt(int a[], int n) {
        int i, j, t = 0;
        for (i = 0; i < n; i++) {
            for (j = 1; j < (n - i); j++) {
                if (a[j - 1] > a[j]) {
                    t = a[j - 1];
                    a[j - 1] = a[j];
                    a[j] = t;
                }
            }
        }
    }

    public static void main(String a[]) {
        int i;
        int array[] = new int[65536];

        for (int j = 0 ; j < array.length; j++) {
            array[j] = array.length - j;
        }

        System.out.println("Values Before the sort:");
        for (i = 0; i < array.length; i++) {
            System.out.print(array[i] + "  ");
        }
        System.out.println();
        bubbleSrt(array, array.length);
        System.out.print("Values after the sort:\n");
        for (i = 0; i < array.length; i++) {
            System.out.print(array[i] + "  ");
        }
        System.out.println();

    }
}
```

Buffered.java

```
package testfiles;

import java.io.*;

public class Buffered {

    public static void main(String args[]) {
        try {

            FileInputStream fis = new
FileInputStream("./src/testfiles/letterRecognition.txt");
            int cnt = 0;
            int n;

            // buffered
            byte buf[] = new byte[1024];

            while ((n = fis.read(buf)) > 0) {
                for (int i = 0; i < n; i++) {
                    if (buf[i] == 'x') {
                        cnt++;
                    }
                }
            }

            fis.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

Factorial.java

```
package testfiles;

public class Factorial {

    public long factorial(int n) {

        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }

    }

    public static void main(String[] args) {

        Factorial fact = new Factorial();
        System.out.println(fact.factorial(20));

    }
}
```

FactorialOpt.java

```
package testfiles;

public class FactorialOpt {

    public long factorial(int n) {

        long fact = 1;
        for (int i = 1; i <= n; i++) {
            fact = fact * i;
        }

        return fact;

    }

    public static void main(String[] args) {

        FactorialOpt fact = new FactorialOpt();
        System.out.println(fact.factorial(20));

    }
}
```

Fibonacci.java

```
package testfiles;

public class Fibonacci {

    public long fibonacci(int n) {
        if (n <= 1) {
               return n;
        }
        else {
               return fibonacci(n-1) + fibonacci(n-2);
        }
    }

    public static void main(String[] args) {
        Fibonacci fib = new Fibonacci();
        System.out.println(fib.fibonacci(20));
    }
}
```

FibonacciOpt.java

```
package testfiles;

public class FibonacciOpt {

    public long fibonacci(int n) {
        long previous = -1;
        long result = 1;
        long sum;
        for (int i = 0; i <= n; ++i) {
            sum = result + previous;
            previous = result;
            result = sum;
        }
        return result;
    }

    public static void main(String[] args) {
        FibonacciOpt fib = new FibonacciOpt();
        System.out.println(fib.fibonacci(20));
    }
}
```

Initialization.java

```
package testfiles;

public class Initialization {

    static class Data {

        private int month;
        private String name;

        Data(int i, String s) {
            month = i;
            name = s;
        }
    }

    Data[] months = {
        new Data(1, "January"),
        new Data(2, "February"),
        new Data(3, "March"),
        new Data(4, "April"),
        new Data(5, "May"),
        new Data(6, "June")
    };

    public static void main(String args[]) {
        final int N = 250000;
        Initialization x;

        for (int i = 1; i <= N; i++) {
            x = new Initialization();
        }

    }
}
```

InitializationOpt.java

```
package testfiles;

public class InitializationOpt {

    static class Data {

        private int month;
        private String name;

        Data(int i, String s) {
            month = i;
            name = s;
        }
    }

    static Data[] months = {
        new Data(1, "January"),
        new Data(2, "February"),
        new Data(3, "March"),
        new Data(4, "April"),
        new Data(5, "May"),
        new Data(6, "June")
    };

    public static void main(String args[]) {
        final int N = 250000;
        InitializationOpt x;

        for (int i = 1; i <= N; i++) {
            x = new InitializationOpt();
        }

    }
}
```

LinearSearch.java

```
package testfiles;

/** Linear search of array for key.  Returns index or -1 if not found.
 *  The upperbound index is not included in the search.
 *  This is to be consistent with the way Java in general expresses ranges.
 *  This searches from low to high index values.
 *  The performance is O(N).
 *  @param a Array of values to be searched.
 *  @param first Index of beginning of range.  First element is a[first].
 *  @param upto  Last element that is searched is a[upto-1].
 *  @param key   Value that is being looked for.
 *  @return      Returns index of the first match, or -1 if no match.
 */

public class LinearSearch {

    public int linearSearch(int[] a, int first, int upto, int key) {

        for (int i = first; i < upto; i++) {
            if (key == a[i]) {
                return i;  // Found key, return index.
            }
        }
        return -1;        // Failed to find key
    }

    public static void main(String[] args) {

        int[] array = new int[10000000];
        LinearSearch search = new LinearSearch();

        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }

        System.out.println(search.linearSearch(array, 0,  array.length, 5000000));

    }

}
```

LinkedListInsertion.java

```
package testfiles;

import java.util.*;

public class LinkedListInsertion {

    public static void main(String args[]) {

        final int N = 50000;

        LinkedList ll = new LinkedList();

        for (int i = 1; i <= N; i++) {
            ll.add(0, new Integer(i));
        }

    }
}
```

LinkedListRandomAccess.java

```
package testfiles;

import java.util.*;

public class LinkedListRandomAccess {

    public static void main(String args[]) {

        final int N = 50000;
        Object o;

        LinkedList ll = new LinkedList();
        for (int i = 0; i < N; i++) {
            ll.add(new Integer(i));
        }

        for (int i = 0; i < N; i++) {
            o = ll.get(i);
        }

    }
}
```

MergeSort.java

```
package testfiles;

public class MergeSort {

    public static void mergeSort(int array[], int lo, int n) {
        int low = lo;
        int high = n;
        if (low >= high) {
            return;
        }

        int middle = (low + high) / 2;
        mergeSort(array, low, middle);
        mergeSort(array, middle + 1, high);
        int end_low = middle;
        int start_high = middle + 1;
        int temp;
        while ((lo <= end_low) && (start_high <= high)) {
            if (array[low] < array[start_high]) {
                low++;
            } else {
                temp = array[start_high];
                for (int k = start_high - 1; k >= low; k--) {
                    array[k + 1] = array[k];
                }
                array[low] = temp;
                low++;
                end_low++;
                start_high++;
            }
        }
    }

    public static void main(String a[]) {
        int i;
        int array[] = new int[65536];

        for (int j = 0 ; j < array.length; j++) {
            array[j] = array.length - j;
        }

        System.out.println("Values Before the sort:");
        for (i = 0; i < array.length; i++) {
            System.out.print(array[i] + "  ");
        }
        System.out.println();
        mergeSort(array, 0, array.length - 1);
        System.out.print("Values after the sort:\n");
        for (i = 0; i < array.length; i++) {
            System.out.print(array[i] + "  ");
        }
        System.out.println();

    }
}
```

StringExample.java

```
package testfiles;

public class StringExample {

    public String implicit(String[] fields) {
        String result = "";
        for (int i = 0; i < fields.length; i++) {
            result += fields[i];
        }
        return result;
    }

    public static void main(String[] args) {

        StringExample strExample = new StringExample();

        String[] str = new String[3000];

        for (int i = 0; i < str.length; i++) {
            str[i] = "A string sample!\n";
        }

        System.out.println(strExample.implicit(str));

    }
}
```

StringExampleOpt.java

```
package testfiles;

public class StringExampleOpt {

    public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < fields.length; i++) {
            result.append(fields[i]);
        }
        return result.toString();
    }

    public static void main(String[] args) {

        StringExampleOpt strExample = new StringExampleOpt();
        String[] str = new String[3000];

        for (int i = 0; i < str.length; i++) {
            str[i] = "A string sample!\n";
        }

        System.out.println(strExample.explicit(str));

    }
}
```

UnBuffered.java

```
package testfiles;

import java.io.*;

public class UnBuffered {

    public static void main(String args[]) {
        try {

            // one read() per character
            FileInputStream fis =  new
FileInputStream("./src/testfiles/letterRecognition.txt");
            int cnt = 0;
            int c;

            while ((c = fis.read()) != -1) {
                if (c == 'x') {
                    cnt++;
                }
            }

            fis.close();

        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

WithBuffering.java

```
package testfiles;

import java.io.*;

public class WithBuffering {

    public static void main(String args[]) {
        try {

            // with buffering
            Reader r = new FileReader("./src/testfiles/letterRecognition.txt");
            r = new BufferedReader(r);

            int c;
            while ((c = r.read()) != -1) {
                ;
            }

            r.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

WithoutBuffering.java

```
package testfiles;

import java.io.*;

public class WithoutBuffering {

    public static void main(String args[]) {
        try {

            // without buffering
            Reader r = new FileReader("./src/testfiles/letterRecognition.txt");

            int c;
            while ((c = r.read()) != -1) {
                ;
            }
            r.close();

        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```