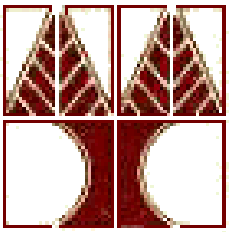
Ατομική Διπλωματική Εργασία

ΕΛΕΓΧΟΣ ΠΡΟΔΙΑΓΡΑΦΩΝ ΠΗΓΑΙΟΥ ΚΩΔΙΚΑ ΛΟΓΙΣΜΙΚΟΥ ΜΕ ΤΗΝ ΧΡΗΣΗ ΤΗΣ JAVA MODELLING LANGUAGE

Μαρίνος Αργυρού

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Τίτλος Ατομικής Διπλωματικής Εργασίας

ΕΛΕΓΧΟΣ ΠΡΟΔΙΑΓΡΑΦΩΝ ΠΗΓΑΙΟΥ ΚΩΔΙΚΑ ΛΟΓΙΣΜΙΚΟΥ ΜΕ ΤΗΝ ΧΡΗΣΗ ΤΗΣ JAVA MODELLING LANGUAGE

Μαρίνος Αργυρού

Επιβλέπων Καθηγητής

Δρ. Ανδρέας Ανδρέου

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

Μάιος 2009

Ευχαριστίες

Ολοκληρώνοντας αυτή την προπτυχιακή ατομική διπλωματική εργασία θέλω να ευχαριστήσω τον επιβλέπων καθηγητή μου κ. Ανδρέα Ανδρέου και τον καθηγητή κ. Τάσο Σοφοκλέους για όλη την απαραίτητη βοήθεια που μου έδωσαν έτσι ώστε να ολοκληρώσω τόσο την έρευνα όσο και την υλοποίηση της συγκεκριμένης εργασίας. Επίσης θέλω να τους ευχαριστήσω για το κίνητρο που μου πρόσφεραν για περεταίρω ενασχόληση με τον κλάδο της τεχνολογίας λογισμικού.

Τέλος, θέλω να ευχαριστήσω τους φίλους και την οικογένεια μου για την στήριξη που μου έδειξαν καθ' όλη την διάρκεια του πτυχίου μου και ειδικότερα στην εκπόνηση της ατομικής διπλωματικής εργασίας. Χωρίς αυτούς η επιτυχία μου στον κλάδο της πληροφορικής δεν έχει και δεν θα έχει νόημα.

Περίληψη

Ο σκοπός της ατομικής διπλωματικής εργασίας (ΑΔΕ) ήταν η μελέτη των θεμάτων γύρω από το έλεγχο του πηγαίο κώδικα ενός λογισμικού με την χρήση της συμπεριφοριστικής γλώσσας προδιαγραφών διαπροσωπίας Java Modelling Language (JML). Η συγκεκριμένη εργασία αποτελεί την συνέχεια και επέκταση της ΑΔΕ του Κωνσταντίνου Ιωάννου [5] όπου μελετήθηκαν οι βασικές αρχές της JML, η αναφορά και ταξινόμηση των προγραμματιστικών λαθών και η πρόταση προτύπων ανίχνευσης αυτών των λαθών με την χρήση της JML. Η εργασία αυτή ξεκίνησε με την μελέτη της πιο πάνω ΑΔΕ και συνεχίστηκε με την έρευνα των θεμάτων που την απασχόλησαν. Η μελέτη συνεχίστηκε με την γλώσσα JML και τα εργαλεία που προσφέρει για επεξεργασία των προδιαγραφών. Στη συνέχεια αναγνωρίστηκαν τα προβλήματα που απασχολούν τον έλεγχο λογισμικού και ειδικότερα τους χρήστες της JML, καταλήγοντας στην απόφαση να δημιουργήσω ένα plugin για ένα εργαλείο ανάπτυξης λογισμικού, το οποίο θα εισάγει αυτόματα τις προδιαγραφές του κώδικα για ανίχνευση συγκεκριμένων λαθών σε JML. Το εργαλείο αποφασίστηκε να είναι η πλατφόρμα ανάπτυξης λογισμικού Eclipse, λόγω των μεγάλων της δυνατοτήτων, ευχρηστίας, τεκμηρίωσης, φορητότητας αλλά και λόγω του ότι είναι το πιο διαδεδομένο δωρεάν εργαλείο ανάπτυξης λογισμικών στην γλώσσα Java ανοικτού κώδικα. Παρόλη την ύπαρξη πολλών εργαλείων για χρήση της JML δεν παρουσιάζεται κανένα που να δίνει την ευχέρεια στο χρήστη να εισάγει αυτόματα προδιαγραφές συμπεριφοράς κλάσεων παρά μόνο τις βασικές λειτουργίες μεταγλώττισης και ελέγχου ικανοποίησης των προδιαγραφών με την εισαγωγή σεναρίων δοκιμής.

Με την ολοκλήρωση του πιο πάνω προγράμματος η έρευνα κινήθηκε στο πεδίο ελέγχου του κώδικα με την χρήση της JML. Αφού μελετήθηκε εις βάθος το λογισμικό JET, μελετήθηκαν τα προβλήματα που παρουσιάζει η συγκεκριμένη προσέγγιση και προτάθηκε ένας νέος τρόπος παραγωγής δεδομένων δοκιμής με την χρήση γενετικών αλγορίθμων, έχοντας ως κριτήριο τις δηλώσεις JML σε συνδυασμό με White Box testing κώδικα Java.

Περιεχόμενα

Λίστα Σχημάτων και πινάκων

1 Γενική Εισαγωγή

1.1 Κίνητρο έρευνας

Ο έλεγχος λογισμικού είναι ένα από τα βασικότερα συστατικά της επιτυχημένης ανάπτυξης ενός συστήματος. Παρουσιάζεται στα περισσότερα μοντέλα ανάπτυξης συστημάτων και πολλές φορά τον συναντούμε κατά τα στάδια προ της επαλήθευσης όπως στο στάδιο του σχεδιασμού και της υλοποίησης. Δεδομένου ότι κατά την αρχική ανάπτυξη ενός συστήματος το 50% του χρόνου και κόστους κατανέμεται στην δοκιμή και επαλήθευση του ιδίου του συστήματος είναι εύκολο να κατανοήσουμε την σημαντικότητα του ελέγχου λογισμικού και τα αποτελέσματα που προκύπτουν από την παραγωγική και ορθή χρήση του [13].

Κατανοώντας την ανάγκη για καλύτερο και γρηγορότερο έλεγχο λογισμικού και λαμβάνοντας υπόψη την εμπειρία που θα προκύψει από την μελέτη ενός από τα θεμέλια της επιτυχημένης ανάπτυξης συστημάτων και της τεχνολογίας λογισμικού γενικότερα, είχα αρκετά κίνητρα για να ασχοληθώ με την συγκεκριμένη έρευνα η οποία με την βοήθεια των κατάλληλων ευκαιριών θα αποτελέσει ακρογωνιαίος λίθος στην καριέρα μου στον χώρο της ανάπτυξης συστημάτων και της τεχνολογίας λογισμικού.

1.2 Στόχοι έρευνας

Πρωταρχικός στόχος της συγκεκριμένης Ατομικής Διπλωματικής Εργασίας (ΑΔΕ) είναι η μελέτη και η κατανόηση των βασικών αρχών του ελέγχου λογισμικού και κατ' επέκταση του ελέγχου πηγαίου κώδικα Java με την χρήση μίας γλώσσας περιγραφής συμπεριφοράς και προδιαγραφών της Java Modelling Language (JML).

Στην συνέχεια η έρευνα θα επεκταθεί στη δημιουργία plugins με σκοπό την ανάπτυξη ενός plugin σε ένα περιβάλλον ανάπτυξης κώδικα Java για την αυτόματη εισαγωγή προδιαγραφών JML, που χρησιμοποιούνται για την ανίχνευση των προγραμματιστικών λογικών, μαθηματικών και κατά τον χρόνο εκτέλεσης λαθών όπως αυτά ορίστηκαν στην ΑΔΕ του Κ. Ιωάννου [5].

Τελειώνοντας θα μελετηθεί η προοπτική ανάπτυξης ενός εργαλείου το οποίο θα χρησιμεύει στον White Box έλεγχο ενός λογισμικού με γνώμονα τις JML δηλώσεις που τον διέπουν. Το εργαλείο αυτό θα πρέπει να συμπεριλαμβάνει αυτόματη παραγωγή Test

Cases για τον έλεγχο δίνοντας την δυνατότητα για ένα εύκολο, γρήγορο και αποδοτικό τρόπο ελέγχου της αναμενόμενης συμπεριφοράς των επιλεγμένων ενοτήτων Java.

1.3 Σκιαγράφηση έρευνας

Η έρευνα θα ξεκινήσει με την περιγραφή των εννοιών γύρω από τον έλεγχο λογισμικού οι οποίες είναι απαραίτητες για την κατανόηση βασικών πτυχών που θίγουν τα επόμενα κεφάλαια. Σε αυτή την ενότητα θα περιγραφούν προβλήματα τα οποία παρουσιάζονται κατά τον έλεγχο λογισμικού όπως και τα λάθη και σφάλματα τα οποία προκύπτουν. Θα επεξηγηθούν επίσης βασικοί ορισμοί όπως ο στατικός και δυναμικός έλεγχος, τα ήδη ελέγχου λογισμικού και τα χαρακτηριστικά τους.

Στην συνέχεια θα μελετηθεί η γλώσσα JML. Η έρευνα θα μελετήσει τα κυριότερα χαρακτηριστικά της, όπως οι προσέγγισης που την ορίζουν και η σύνταξη της με την παρουσίαση παραδειγμάτων. Θα μελετηθούν μερικά από τα κύρια εργαλεία που υπάρχουν και μπορούν να επεξεργαστούν δηλώσεις JML όπως και εργαλεία για δοκιμή και έλεγχο των προδιαγραφών.

Στην συνέχεια θα παρουσιαστούν τα προγραμματιστικά λάθη και τα πρότυπα τα οποία έχουν προταθεί από τον Κ. Ιωάννου. Αρχικά τα λάθη θα ταξινομηθούν και στην συνέχεια θα παρουσιαστούν ορισμένα στην JML.

Η συνέχεια προκύπτει με την δημιουργία του plugin για αυτόματη εισαγωγή των πιο πάνω προτύπων. Θα παρουσιαστεί το περιβάλλον ανάπτυξης εφαρμογών στο οποίο θα προστεθεί το Plugin. Θα επεξηγηθεί η διαδικασία ανάπτυξης του plugin μέχρι και την εγκατάσταση του.

Τέλος ο έρευνα θα ολοκληρωθεί με την μελέτη για έλεγχο των προδιαγραφών JML με την χρήση αυτοματοποιημένης δημιουργίας Test Cases και συγκεκριμένα με γενετικούς αλγορίθμους. Θα οριστούν τα χαρακτηριστικά της συγκεκριμένης προσέγγισης και οι τρόποι αξιολόγησης της σε σύγκριση με τις υπάρχουσες προσεγγίσεις για τον συγκεκριμένο τρόπο ελέγχου λογισμικού.

2 Έλεγχος Λογισμικού

2.1 Εισαγωγή

Ο έλεγχος λογισμικού είναι μία εμπειρική μελέτη που εφαρμόζεται με σκοπό την συλλογή πληροφοριών όσο αφορά την ποιότητα του προϊόντος ή της υπηρεσίας που ελέγχεται, με γνώμονα το περιβάλλον στο οποίο προορίζεται για να λειτουργήσει. Ακόμα προσφέρει μία αντικειμενική και ανεξάρτητη όψη του λογισμικού έτσι ώστε η αγορά να εκτιμήσει και να κατανοήσει τα ρίσκα που κρύβονται κατά την δημιουργία και ολοκλήρωση ενός λογισμικού. Οι τεχνικές ελέγχου λογισμικού περιλαμβάνουν μεταξύ άλλων διεργασίες εκτέλεσης προγραμμάτων ή εφαρμογών με σκοπό την ανίχνευση προγραμματιστικών λαθών. Επίσης μπορεί να οριστεί ως η διαδικασία επικύρωσης και επαλήθευσης ενός λογισμικού εάν συνάδει με τις προδιαγραφές που έχουν οριστεί κατά την ανάλυση των αναγκών, την σχεδίαση και την ανάπτυξη του, έτσι ώστε εάν λειτουργεί όπως έχει οριστεί και συμφωνηθεί να ολοκληρωθεί με τα συγκεκριμένα χαρακτηριστικά.

Ο έλεγχος λογισμικού, ανάλογα με το μοντέλο ανάπτυξης λογισμικού που έχει επιλεχθεί, μπορεί να εφαρμοστεί σε οποιοδήποτε στάδιο, αλλά ο μεγαλύτερος φόρτος ελέγχου εφαρμόζεται αφού έχουν οριστεί οι προδιαγραφές του συστήματος βάσει των αναγκών και ο η συγγραφή του κώδικα έχει ολοκληρωθεί.

2.2 Προβλήματα κατά τον έλεγχο Λογισμικού

Ο έλεγχος λογισμικού δεν έχει την δυνατότητα να αναγνωρίσει όλες τις ατέλειες μέσα σε ένα λογισμικό. Αντίθετα απλά συγκρίνει την κατάσταση και την συμπεριφορά του ίδιου του προϊόντος με διάφορους κανόνες ή μηχανισμούς με τους οποίους μπορεί κάποιος να αναγνωρίσει το πρόβλημα. Αυτοί οι κανόνες μπορούν να αφορούν τις προδιαγραφές, όμοια προϊόντα, παλαιότερες εκδόσεις του ίδιου του προϊόντος, συμπεράσματα όσο αφορά αναμενόμενα ή ηθελημένα αποτελέσματα, προσδοκίες των πελατών ή χρηστών, διάφορα standards που αφορούν το προϊόν, νόμους που εφαρμόζεται κατά την χρήση του και άλλα κριτήρια.

2.3 Λάθη και σφάλματα

Τα λάθη και τα σφάλματα [26] τα οποία εμφανίζονται στα λογισμικά δεν αφορούν μόνο λάθη στον κώδικα. Λάθη που ανιχνεύονται κατά τον έλεγχο του λογισμικού μπορεί να οφείλονται σε διάφορους παράγοντες όπως οι ανολοκλήρωτες απαιτήσεις που οδηγούν σε ελλιπή προδιαγραφές που με τη σειρά τους οδηγούν τον δημιουργό του συστήματος σε παραλήψεις όσο αφορά την υλοποίηση του κώδικα και των υπολοίπων χαρακτηριστικών της σχεδίασης του συστήματος. Όμως ακόμα και τα λάθη τα οποία βρίσκονται στον κώδικα δεν είναι πάντοτε ανιχνεύσιμα. Λάθη τα οποία βρίσκονται σε νεκρά σημεία του κώδικα, δηλ. τα αποτελέσματα της εκτέλεσης των συγκεκριμένων σημείων δεν χρησιμοποιούνται πουθενά αλλού, δεν θα παρουσιάσουν κανένα σημάδι λάθους.

Όπως προκύπτει και από τα πιο πάνω ο μόνος τρόπος να ανιχνευτούν λάθη που δεν οφείλονται σε προγραμματιστικές ατασθαλίες είναι να αποκλειστούν όλες οι περιπτώσεις λαθών λόγω του κώδικα. Ο έλεγχος όμως όλων των δυνατών συνδυασμών εισόδων ή καταστάσεων για ένα πρόγραμμα είναι πρακτικά αδύνατο ακόμα και για απλές υλοποιήσεις. Αυτό σημαίνει ότι ο αριθμός των λαθών σε ένα λογισμικό μπορεί να είναι πολύ μεγάλος και ακόμα τα λάθη που συναντώνται σπανιότερα, δυσκολότερο να εντοπιστούν κατά τον έλεγχο.

2.4 Static and Dynamic Testing

Οι προσεγγίσεις για τον έλεγχο ενός προγράμματος μπορούν να χωριστούν σε στατικό έλεγχο και δυναμικό έλεγχο. Στατικός έλεγχος μπορούν να χαρακτηριστούν τα reviews,τα walkthroughs ή τα inspections ενώ η εκτέλεση του κώδικα με βάση ένα σύνολο από σενάρια ελέγχου (Test Cases (TC)) συγκαταλέγεται στον δυναμικό έλεγχο. Παρατηρείται όμως συχνά ότι ο στατικός έλεγχος αποφεύγεται και αυτό οφείλεται στην δυνατότητα του δυναμικού ελέγχου να ανιχνεύει το μεγαλύτερο φάσμα των ανιχνεύσιμων λαθών. Ο δυναμικός έλεγχος μπορεί να ανεξαρτητοποιηθεί με την διαδικασία ολοκλήρωσης του λογισμικού εφόσον μπορούν να ελεγχθούν ξεχωριστά διάφορες ενότητες ή απλά διαδικασίες του προγράμματος.[18]

2.5 Black Box Testing

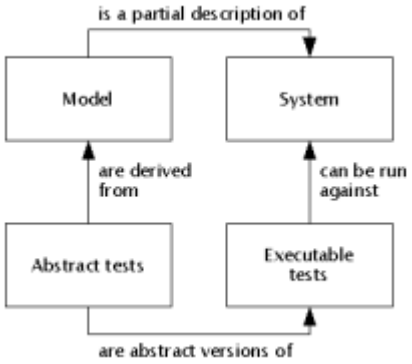
Κατά τον δυναμικό έλεγχο έχουμε την δυνατότητα να ακολουθήσουμε δύο μεθόδους παραγωγής και ελέγχου TC. Υπάρχει το Black Box κατά την οποία ο έλεγχος γίνεται χωρίς να γνώση της εσωτερικής υλοποίησης του προγράμματος. Μέθοδοι οι οποίες χρησιμοποιούν το Black Box Testing είναι η Specification-based testing, model-based testing, boundary value analysis κ.α.[16].

2.5.1 Specification – Based Testing

Ο έλεγχος βάσει των προδιαγραφών έχει ως στόχο τον έλεγχο της λειτουργίας του λογισμικού σύμφωνα με τα τις καταχωρημένες προδιαγραφές. Έτσι ο έλεγχος γίνεται στα δεδομένα εισόδου και τα αποτελέσματα που εξάγονται από το αντικείμενο ελέγχου. Κατά των έλεγχο βάσει των προδιαγραφών συνήθως απαιτείται μεγάλος αριθμός από TC τα όποια μπορούν να παράγουν αποτελέσματα με τα οποία θα γίνει η επαλήθευση και η σύγκριση με τα αναμενόμενα. Ο συγκεκριμένος έλεγχος είναι απαραίτητος αν και σε μερικές περιπτώσεις δεν παρέχει κάλυψη για συγκεκριμένους κινδύνους [14].

2.5.2 Model – Based Testing

Στην περίπτωση του ελέγχου βάσει μοντέλου τα TC εξάγονται από ολοκληρωμένο ή κομμάτι ενός μοντέλου που περιγράφει τις λειτουργικές απαιτήσεις του συστήματος που ελέγχεται. Συνήθως το μοντέλο είναι η αφαιρετική, μερική συμπεριφορά του συστήματος. Δημιουργείται ένα αφαιρετικό σύνολο από σενάρια ελέγχου βάσει του αφαιρετικού μοντέλου και στην συνέχεια δημιουργούνται τα εκτελέσιμα σενάρια με γνώμονα τα αφαιρετικά. Πιο κάτω ακολουθεί το περιγραφικό σχήμα της συγκεκριμένης μεθόδου.[20]



Σχήμα 2.1 : Model Based Testing Scheme [20]

2.5.3 Πλεονεκτήματα και Μειονεκτήματα του Black Box Testing

Ο έλεγχος Black Box δεν δεσμεύεται από τον ίδιο τον κώδικα και διατηρεί την αντίληψη του απλή και περιεκτική. Με δεδομένη την ύπαρξη λαθών στην λειτουργία του προγράμματος ο συγκεκριμένος έλεγχος ανιχνεύει λάθει τα οποία ο προγραμματιστής αδυνατεί να ελέξει. Παρόλα αυτά ο έλεγχος γίνεται χωρίς καμία γνώση του κώδικα με αποτέλεσμα ο ελεγκτής να περπατά στα τυφλά. Συχνά παρατηρείται το φαινόμενο παραγωγής πολλών αχρείαστων TC αφού το λάθος μπορεί να ανιχνευτεί με μόνο ένα. Ακόμη μερικά σημεία του κώδικα δεν ελέγχονται ποτέ εφόσον δεν έχουν καταγραφεί ρητά στις απαιτήσεις. [18]

2.6 White Box Testing

Κατά τον δυναμικό έλεγχο του λογισμικού η δεύτερη μέθοδος δημιουργίας TC είναι με το White Box Testing. Στο White Box Testing ο έλεγχος γίνεται στον κώδικα που υλοποιεί τις δομές δεδομένων και στους αλγορίθμους που έχουν χρησιμοποιούνται. Ο έλεγχος αντίθετα με το Black Box γίνεται με την γνώση της εσωτερικής αρχιτεκτονικής του λογισμικού. Διάφοροι τύποι White Box Testing είναι το Application Programming Interface (API) Testing όπου γίνεται έλεγχος του λογισμικού με την χρήση APIs, το code coverage όπου δημιουργούνται TC τα οποία καλύπτουν συγκεκριμένα κριτήρια για κάλυψη του κώδικα, π.χ. δημιουργία TC για την κάλυψη όλων των statements στο πρόγραμμα τουλάχιστο μία φορά, Fault Injection όπου εισάγονται συγκεκριμένα λανθασμένα δεδομένα για να ελεγχτεί η ορθή λειτουργία των statements και στατικός έλεγχος ο οποίος έχει την δυνατότητα να ελέγχει όλα τα είδη του στατικού ελέγχου.

Ακόμα με το White Box Testing μπορούμε να ελέγξουμε την κάλυψη του κώδικα με ποικίλους τρόπους έτσι ώστε ο έλεγχος να γίνεται στις ενότητες ή σε ολόκληρο τον

κώδικα ανάλογα με τις ανάγκες του ελέγχου. Εκτός από το statement coverage που συναντήσαμε πιο πάνω βλέπουμε συχνά το path coverage το οποίο ελέγχει την κάλυψη όλων των δυνατών μονοπατιών από την αρχή του κώδικα που ελέγχουμε, το function coverage που καταγράφει την κάλυψη των συναρτήσεων του κώδικα, branch coverage το οποίο ελέγχει την κάλυψη των διακλαδώσεων του κώδικα στα σημεία που διαχωρίζονται τα μονοπάτια όπως τα if statements κ.α.[15].

3 Java Modelling Language

3.1 Εισαγωγή στη JML

Η Java Modelling Language (JML) είναι μία γλώσσα προδιαγραφών συμπεριφοριστικής διαπροσωπίας η οποία χρησιμοποιείται για τον καθορισμό της συμπεριφοράς ενοτήτων του πηγαίου κώδικα Java. Συνδυάζει την προσέγγιση της σχεδίασης με συμβόλαιο (Design by Contract) της Eiffel [12] και την προσέγγιση βάση προδιαγραφών μοντέλου (model-based specification approach) που ορίζεται στην οικογένεια γλωσσών προδιαγραφών διαπροσωπίας Larch [17], χρησιμοποιώντας στοιχεία από την Refinement Calculus. Με απλά λόγια η JML παρέχει τη σημασιολογία για την ακριβή περιγραφή μίας ενότητας JAVA, αποβάλλοντας την πιθανότητα αμφιβολίας όσο αφορά τις προθέσεις με τις οποίες σχεδιάστηκε η συγκεκριμένη ενότητα. Οι προδιαγραφές των ενοτήτων μπορούν να γραφτούν σαν σχόλια μέσα στο αρχείο κώδικα JAVA είτε σε ξεχωριστό αρχείο προδιαγραφών. Λόγω του ότι στα αρχεία JAVA η σημασιολογία των προδιαγραφών βρίσκεται σε σχόλια για να μην επηρεάζεται η μεταγλώττιση από τον JAVA Compiler, είναι απαραίτητη η εφαρμογή επιπλέον εργαλείων έτσι ώστε να γίνει εφικτή η χρήση των συμπεριφοριστικών πληροφοριών που παρέχει η JML[1][7].

3.2 Design by Contract

Η μέθοδος Design by Contract (DBC) ορίζεται σαν μία μέθοδος σχεδίασης λογισμικού. Η βασική ιδέα της DBC είναι ότι μεταξύ μίας κλάσης και των πελατών αυτής της κλάσης διατηρείται ένα συμβόλαιο. Ο πελάτης πρέπει να εγγυάται συγκεκριμένες συνθήκες κατά την κλήση μίας μεθόδου που ανήκει στην κλάση και με τη σειρά της η κλάση εγγυάται συγκεκριμένες ιδιότητες κατά την επιστροφή από την κλήση της μεθόδου. Τα συμβόλαια συμβολίζονται με precondition και postconditions, ορίζονται από την γλώσσα προδιαγραφών που χρησιμοποιείται μέσα στον πηγαίο κώδικα με αποτέλεσμα η ανίχνευση τυχών παραβίασης αυτού του συμβολαίου γίνεται άμεσα κατά το την εκτέλεση του προγράμματος.

Ένα συμβόλαιο ορίζει τόσο τα δικαιώματα όσο και τις υποχρεώσεις των κλάσεων αλλά και των πελατών. Για παράδειγμα κατά την κλήση μίας μεθόδου που δέχεται ένα

ακέραιο αριθμό και στη συνέχεια επιστρέφει το τετράγωνο του, μπορούμε να ορίσουμε σαν pre-condition ότι ο αριθμός πρέπει να είναι μεγαλύτερος από το 0 και σαν post-condition ότι η τιμή που επιστρέφεται πρέπει να είναι ίση με το αποτέλεσμα του αριθμού πολλαπλασιασμένου με τον εαυτό του. Αυτό είναι το συμβόλαιο που ορίζεται από την κλάση η οποία ορίζει την μέθοδο και τον πελάτη ο οποίος καλεί την πιο πάνω μέθοδο [1].

3.3 Model-Based Specification Approach in Larch

Η Larch είναι μία προσέγγιση για αυστηρή διατύπωση των προδιαγραφών για ενότητες προγραμμάτων. Ακολουθεί και επεκτείνει την προσέγγιση των ιδεών Hoare όσο αφορά τις προδιαγραφές προγραμμάτων, με την διαφορά ότι χωρίζεται σε δύο επίπεδα . Στο πάνω επίπεδο χρησιμοποιείται η behavioural interface specification language (BISL) η οποία είναι διαμορφωμένη για ενσωμάτωση σε κάποια συγκεκριμένη γλώσσα, C++ αρχικά. Η συγκεκριμένη BISL χρησιμοποιεί preconditionsκαι postconditions για να ορίσει τις προδιαγραφές της κάθε ενότητας του προγράμματος. Στο κάτω επίπεδο βρίσκεται η γλώσσα Larch Shared Language (LSL) η οποία χρησιμοποιείται για την περιγραφή των μαθηματικών ορισμών και λεξιλογίου που συναντούμε στις precondition και postcondition προδιαγραφές. Η βασική ιδέα είναι με την LSL να ορίζεται ένα γενικό μοντέλο παρέχοντας και το απαραίτητο μαθηματικό λεξιλόγιο και η BISL για να ορίζεται τόσο η διαπροσωπία όσο και η συμπεριφορά που διέπει την κάθε προγραμματιστική ενότητα [17].

3.4 Refinement Calculus

Η Refinement Calculus είναι μία αυστηρότερα δομημένη μορφή της μεθόδου βήμα προς βήμα εκλέπτυνσης κατά την δημιουργία ενός προγράμματος. Η απαραίτητη συμπεριφορά του προγράμματος είναι καθορισμένη ως μία αφαιρετική, μη εκτελέσιμη μορφή προγράμματος που μετά από μία σειρά μετατροπών, διατηρώντας την αρχική ορθότητα και νόημα, μεταμορφώνεται σε ένα αποδοτικό εκτελέσιμο πρόγραμμα. Με αυτή την λογική τα προγράμματα γράφονται σαν μία μορφή από λογικές και μαθηματικές σημάνσεις που όταν συνδυαστούν παράγονται οι σημάνσεις προδιαγραφών με τις οποίες ένα πρόβλημα μπορεί να αναπαρασταθεί στην αφαιρετική του μορφή [25].

3.5 Σύνταξη της JML και παραδείγματα

Η σύνταξη της JML ξεκινά με την χρήση σχολίων ώστε να αγνοείται από τον Java Compiler. Στην συνέχεια με τον χαρακτήρα @ σημειώνεται ότι συγκεκριμένη γραμμή περιέχει προδιαγραφές που αφορούν την κλάση, μέθοδο ή τύπο δεδομένων και επεξηγούν την συμπεριφορά που διέπει το κάθε στοιχείο. Πιο κάτω βλέπουμε τους δύο τρόπους τους οποίους συνάσσεται μία δήλωση προδιαγραφής σε JML [11] [1][2][3][8]:

```
//@ <JML specification>

/*@ <JML specification> @*/
```

Σχήμα 3.1 : Τρόπος δήλωσης προδιαγραφών JML

3.5.1 Variable Modifiers

Τα modifiers συνιστούν δεσμευμένες λέξεις στην JML και αποτελούν την περιγραφή στοιχείων της Java όσο αφορά την συμπεριφορά τους στο επίπεδο της JML και μόνο. Τα modifiers που χρησιμοποιούνται στην Java έχουν την ίδια σημασία και στην JML. Παρακάτω φαίνεται η χρήση μερικών modifiers της JML και η σύνταξη τους.

3.5.1.1 spec_public

Το modifier *spec_public* επιτρέπει την δήλωση ενός στοιχείου ως public μόνο για σκοπούς προδιαγραφών. Το συγκεκριμένο modifier μπορεί να χρησιμοποιηθεί μόνο σε συγκεκριμένα στοιχεία τα οποία έχουν δηλωθεί με περιορισμένη ορατότητα στην Java.

```
public class Person {
        private /*@ spec_public @*/ String name;
        private /*@ spec_public @*/ int weight;
        ...
}
```

Σχήμα 3.2 : Παράδειγμα χρήσης spec_public

3.5.1.2 spec_protected

Το modifier *spec_protected* σε αντίθεση με το spec_public τον ορισμό στοιχείων του κώδικα ως protected για σκοπούς προδιαγραφών μόνο. Μπορεί να χρησιμοποιηθεί όταν ένα στοιχείο έχει ορισθεί με χαμηλότερη ορατότητα στον Java κώδικα. Μπορεί δηλαδή να χρησιμοποιηθεί για να αλλάξουμε την ορατότητα ενός πεδίου ή μεθόδου η οποία έχει οριστεί ως private ή default access.

```
public class Person {
        private /*@ spec_protected @*/ String name;
        /*@ spec_protected @*/ int weight;
        …
}
```

Σχήμα 3.3 : Παράδειγμα χρήσης spec_protected

3.5.1.3 pure

Το modifier pure αφορά μόνο μεθόδους και κατασκευαστές. Δηλώνει ότι κατά την εκτέλεση του κάθε στοιχείου δεν θα επιφέρει καμία αλλαγή στην κατάσταση του προγράμματος. Με αυτό τον τρόπο η μέθοδος μπορεί να χρησιμοποιηθεί σε άλλα JML expressions.

```
public class Person {
        private /*@ spec_public @*/ String name;
        private /*@ spec_public @*/ int weight;

        public /*@ pure @*/ int getWeight();
        …
}
```

Σχήμα 3.4 : Παράδειγμα χρήσης pure

3.5.2 Nullity Modifiers

Τα modifiers τα οποία ορίζουν το nullity τοπικών και global μεταβλητών. Κατά τον ορισμό μίας μεταβλητής τύπου reference υπονοείται ότι η συγκεκριμένη μεταβλητή δεν μπορεί να πάρει την τιμή null. Όμως μπορούν να οριστούν ρητά με τα ακόλουθα modifiers nullable και non_null για τον αυστηρό καθορισμό των προδιαγραφών τους.

```
public class Person {
        private /*@ spec_public non_null@*/ String name;
        private /*@ spec_public @*/ int weight;


        …
}
```

Σχήμα 3.5 : Παράδειγμα χρήσης non_null

```
public class Person {
        private /*@ spec_public nullable@*/ String name;
        private /*@ spec_public @*/ int weight;


        …
}
```

Σχήμα 3.6 : Παράδειγμα χρήσης nullable

3.5.3 Keywords

3.5.3.1 Requires

Δηλώνει το precondition για την μέθοδο που ακολουθεί στον κώδικα. Λειτουργεί όπως έχει οριστεί στο DBC μεταξύ μεθόδων και πελατών όπου ενημερώνει τον πελάτη ποιες είναι οι απαραίτητες προδιαγραφές για την κλήση της συγκεκριμένης μεθόδου.

```
public class Person {
        private /*@ spec_public non_null @*/ String name;
        private /*@ spec_public @*/ int weight;

        /*@
        @ requires kgs >= 0;
        @ requires weight + kgs >= 0;
        @*/
        public void addKgs(int kgs);

        ...
}
```

Σχήμα 3.7 : Παράδειγμα χρήσης requires

3.5.3.2 Ensures

Δηλώνει το postcondition για την μέθοδο που ακολουθεί στον κώδικα. Λειτουργεί όπως έχει οριστεί στο DBC μεταξύ μεθόδων και πελατών όπου ενημερώνει τον πελάτη ποιες είναι οι απαραίτητες προδιαγραφές της επιστροφής αυτής της μεθόδου.

```
public class Person {
        private /*@ spec_public non_null @*/ String name;
        private /*@ spec_public @*/ int weight;

        /*@
        @ requires kgs >= 0;
        @ requires weight + kgs >= 0;
        @ ensures weight == \old(weight + kgs);
        @*/
        public void addKgs(int kgs);

        ...
}
```

Σχήμα 3.8 : Παράδειγμα χρήσης ensures

3.5.3.3 Signals

Ορίζει την συνθήκη κατά την οποία μία μέθοδος παρουσιάζει συγκεκριμένο Exception κατά την κλήση της.

```
public class Person {
    private /*@ spec_public non_null @*/ String name;
    private /*@ spec_public @*/ int weight;


    /*@
    @ requires kgs >= 0;
    @ requires weight + kgs >= 0;
    @ ensures weight == \old(weight + kgs);
    @ signals (Exception) kgs-weight < 0;
    @*/
    public static int notCorrect2(int x) throws Exception {
          x = -1;
          throw new Exception();
     }
```

Σχήμα 3.9 : Παράδειγμα χρήσης signals

3.5.3.4 Assignable

Ορίζει τα πεδία τα οποία έχουν τη δυνατότητα να τους ανατεθεί τιμή μέσα στην μέθοδο που ακολουθεί. Η συγκεκριμένη έκφραση αναφέρεται σε μεταβλητές οι οποίες έχουν ήδη δεσμευτεί γι' αυτό και ορίζουν το pre-state τους και συμπεριλαμβάνεται στα pre-conditions της συγκεκριμένης μεθόδου.

```
public class EqualsN
{
        public int n;

        //@ assignable this.n;
        //@ ensures this.n == n;
        public EqualsN(int nVal) {
        n=nVal;
        }

        ...
}
```

Σχήμα 3.10 : Παράδειγμα χρήσης assignable

3.5.3.5 Invariant

Ορίζει ότι οι μεταβλητές, για τις οποίες περιγράφει τις προδιαγραφές τους, θα διατηρήσουν την ορθότητα των προδιαγραφών τους καθ' όλη την διάρκεια εκτέλεσης της κλάσης. Η συμπεριφορά invariant ορίζεται για τις global μεταβλητές της κλάσης και διατηρούν τις προδιαγραφές τους κατά την εκτέλεση οποιασδήποτε μεθόδου.

```
public class Account {
          private /*@ spec_public @*/ int bal;
          //@ public invariant bal >= 0;

          /*@
          @ requires amt >= 0;
          @ assignable   bal;
          @ ensures bal == amt;
          @*/
            public Account(int amt) {
                  bal = amt;
            }
          …
}
```

Σχήμα 3.11 : Παράδειγμα χρήσης invariant

3.5.3.6 Also

Δηλώνει ρητά ότι η συγκεκριμένη μέθοδος κληρονομεί τα διάφορα conditions που ορίζονται στις υπερκλάσεις της.

```
public class Account {
          private /*@ spec_public @*/ int bal;
          //@ public invariant bal >= 0;

          /*@
          @ also
          @ requires amt >= 0;
          @ assignable   bal;
          @ ensures bal == amt;
          @*/
            public Account(int amt) {
                  bal = amt;
            }
          …
}
```

Σχήμα 3.12 : Παράδειγμα χρήσης also

3.5.3.7 Assert

Μία δήλωση assert καλείται όταν θέλουμε να ελέγξουμε σε κάποιο μέρος του προγράμματος την ορθότητα ενός συγκεκριμένου κατηγορήματος. Ο έλεγχος αυτών των κατηγορημάτων γίνεται κατά την εκτέλεση του προγράμματος χρησιμοποιώντας το runtime assertion checker.

```
public class Account {
        private /*@ spec_public @*/ int bal;
        //@ public invariant bal >= 0;

        /*@
        @ also
        @ requires amt >= 0;
        @ assignable   bal;
        @ ensures bal == amt;
        @*/
          public Account(int amt) {
                bal = amt;
                //@ assert bal>15;
          }
```

Σχήμα 3.13 : Παράδειγμα χρήσης assert

3.5.4 Εκφράσεις JML

Οι εκφράσεις της JML αποτελούνται από τελεστές και identifiers τα οποία σε συνδυασμό με τα modifiers και τις δεσμευμένες λέξεις δίνουν την δυνατότητα για αλληλεπίδραση με τα αντικείμενα της Java, τις μεθόδους τους και τους τελεστές που βρίσκονται μέσα στις μεθόδους που σχολιάζονται. Με αυτό τον τρόπο παρέχεται η δυνατότητα μίας πιο αυστηρής σύνθεσης των προδιαγραφών τόσο για τις κλάσεις, όσο για τις μεθόδους και τα πεδία που τις διέπουν.

3.5.4.1 Primary Expressions

3.5.4.1.1 \result

Το Identifier το οποίο συγκρατεί το αποτέλεσμα επιστροφής της μεθόδου που ακολουθεί την δήλωση των προδιαγραφών. Έχει τον ίδιο τύπο με τον τύπο επιστροφής της μεθόδου και μπορεί να χρησιμοποιηθεί μόνο σε postcondition statements.

3.5.4.1.2 \old(name)

Το modifier το οποίο συγκρατεί την τιμή της μεταβλητής στην παρένθεση με την οποία εισήλθε στην ακόλουθη μέθοδο.

3.5.4.2 Universal and Existential Quantifiers

3.5.4.2.1 \forall

Ο ποσοτικός τελεστής που επιστρέφει true εάν για όλες τις τιμές μέσα στο πεδίο τιμών ισχύει το κατηγόρημα που ορίζεται.

3.5.4.2.2 \exists

Ο ποσοτικός τελεστής που επιστρέφει true εάν έστω για μία τιμή μέσα στο πεδίο τιμών ισχύει το κατηγόρημα που ορίζεται.

3.5.5 Λογικοί Τελεστές

Ο τελεστής “<==>” έχει την σημασία του “αν και μόνο αν” και προϋποθέτει ότι οι δύο πλευρές της συνάρτησης είναι τύπου Boolean. Παρόλο που έχει την ίδια σημασία με τον τελεστή της Java “==”, έχει χαμηλότερη προτεραιότητα με αποτέλεσμα η έκφραση “result <==> size == 0” να μεταφράζεται σε "\result == (size == 0)". Το ίδιο ισχύει και για τον τελεστή “<=!=>” σε σύγκριση με τον “!=”.

3.5.6 Πλήρες Παράδειγμα JML

Ακολουθεί ένα πλήρες παράδειγμα μίας αφαιρετικής κκλάσης η οποία αποτελείται από αυστηρά ορισμένες προδιαγραφές με τη σύνταξης της JML που μικρό μέρος της επεξηγήθηκε πιο πάνω. Πιο κάτω περιγράφεται μία κλάση με μεθόδους που αφορούν ένα σωρό από ακεραίους ορίζοντας τις προδιαγραφές των μεθόδων που την χαρακτηρίζουν.

```
package org.jmlspecs.samples.jmlrefman;                  // line 1
                                                         // line 2
public abstract class IntHeap {                          // line 3
                                                         // line 4
   //@ public model non_null int [] elements;           // line 5
                                                         // line 6
   /*@ public normal_behavior                           // line 7
     @   requires elements.length >= 1;                 // line 8
     @   assignable \nothing;                           // line 9
     @   ensures \result                                // line 10
     @       == (\max int j;                            // line 11
     @             0 <= j && j < elements.length;       // line 12
     @             elements[j]);                        // line 13
     @*/                                                // line 14
   public abstract /*@ pure @*/ int largest();           // line 15
                                                         // line 16
   //@ ensures \result == elements.length;              // line 17
   public abstract /*@ pure @*/ int size();              // line 18
}
```

Σχήμα 3.14 : Πλήρες παράδειγμα JML

3.6 Εργαλεία για JML

Έχουν παρουσιαστεί αρκετά εργαλεία [2] για την διαχείριση και επεξεργασία των προδιαγραφών JML, όπως και για την διαδικασία ελέγχου μίας εφαρμογής. Τα πιο βασικά εργαλεία για parsing και typechecking είναι ήδη αρκετά χρήσιμα, π.χ. για την διατήρηση του κώδικα ενημερωμένου κατά τις αλλαγές των ονομάτων και των παραμέτρων με ενά απλό πέρασμα του typechecker ώστε να αναγνωρίσει διάφορες JML δηλώσεις που δεν υπάρχουν πια. Εκτός από την ύπαρξη βασικών εργαλείων όπως τα πιο πάνω, η JML μπορεί να χρησιμοποιηθεί για στατική ανάλυση του κώδικα εφόσον δεν είναι απαραίτητο να εκτελεστεί ο κώδικας για γίνουν διάφορες μετρήσεις και έλεγχοι. Επίσης μπορεί να χρησιμοποιηθεί σε εργαλεία για αυστηρή επαλήθευση μίας εφαρμογής, runtime assertion checking εργαλεία, για καταγραφή δυναμικά σχηματισμένων σταθερών (JML's runtime assertion checker, jmlc), εργαλεία unit testing και για τεκμηρίωση του κώδικα (jmldoc tool).

3.6.1 Βασικά εργαλεία

3.6.1.1 Jmlc

Ο JML Compiler είναι από τα πιο βασικά εργαλεία, το οποίο είναι υπεύθυνο για την μεταγλώττιση των JML statements μέσα από ένα αρχείο κώδικα Java και την μετατροπή τους σε runtime assertion checks. Λειτουργεί ως ένας Java Compiler με επιπλέον χαρακτηριστικό την πρόσθεση στο παραγόμενο αρχείο .class αυτόματων ελέγχων των assertions κατά την εκτέλεση του κώδικα.

3.6.1.2 Jmlrac

Το jmlrac είναι το εργαλείο το οποίο συνδυάζει τις απαιτούμενες βιβλιοθήκες για ανίχνευση των παραβιάσεων των προδιαγραφών, που έχουν καταγραφεί με το jmlc, και δίνει την δυνατότητα στο χρήστη να τρέξει το εκτελέσιμο αρχείο με τις προδιαγραφές για ανίχνευση των λαθών.

3.6.1.3 Jmldoc

Λειτουργεί ως ενναλλακτικό του Javadoc, το οποίο μετατρέπει εκτός από τα σχόλια και τα χαρακτηριστικά των κλάσεων, τις δηλώσεις JML σε αρχεία .html τα οποία τεκμηριώνουν τον κώδικα και τις ιδιότητες του.

3.6.1.4 Jmlunit

Το jmlunit είναι ένα εργαλείο το οποίο μέσα από την επεξεργασία ενός αρχείου πηγαίου κώδικα Java ή προδιαγραφών JML μπορεί να δημιουργήσει τον κώδικα για μία αφαιρετική Oracle κλάση που μπορεί να δημιουργεί Test Cases βασισμένα στο JUnit framework.

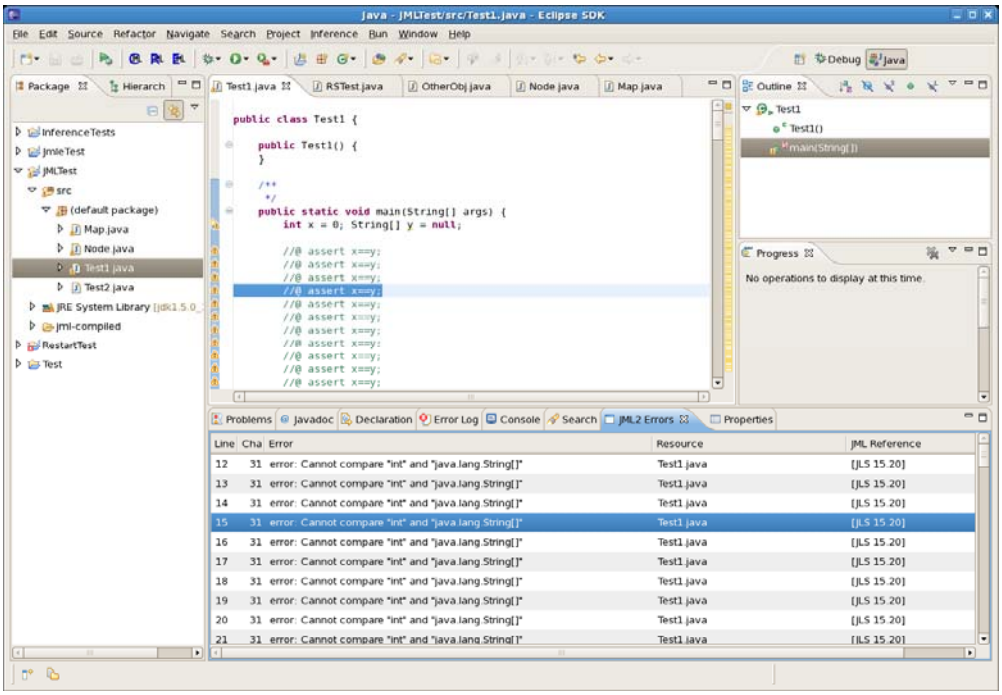
3.6.2 Επιπλέον εργαλεία

3.6.2.1 ESC/Java2

Το *Extended Static Checker for Java version 2* (ESC/Java2) είναι ένα προγραμματιστικό εργαλείο που προσπαθεί να ανιχνεύσει κοινά λάθη που προκύπτουν κατά τον χρόνο εκτέλεσης αναλύοντας στατικά τον πηγαίο κώδικα Java και τις αυστηρές προδιαγραφές JML που έχουν ορισθεί. Οι χρήστες μπορούν να ελέγξουν τα μέγεθος και τα διάφορα είδη ελέγχου που πραγματοποιεί το ESC/Java2 συμβολίζοντας τα προγράμματα με σχόλια συγκεκριμένης τροποποίησης που λέγονται *pragmas* [3][2][6].

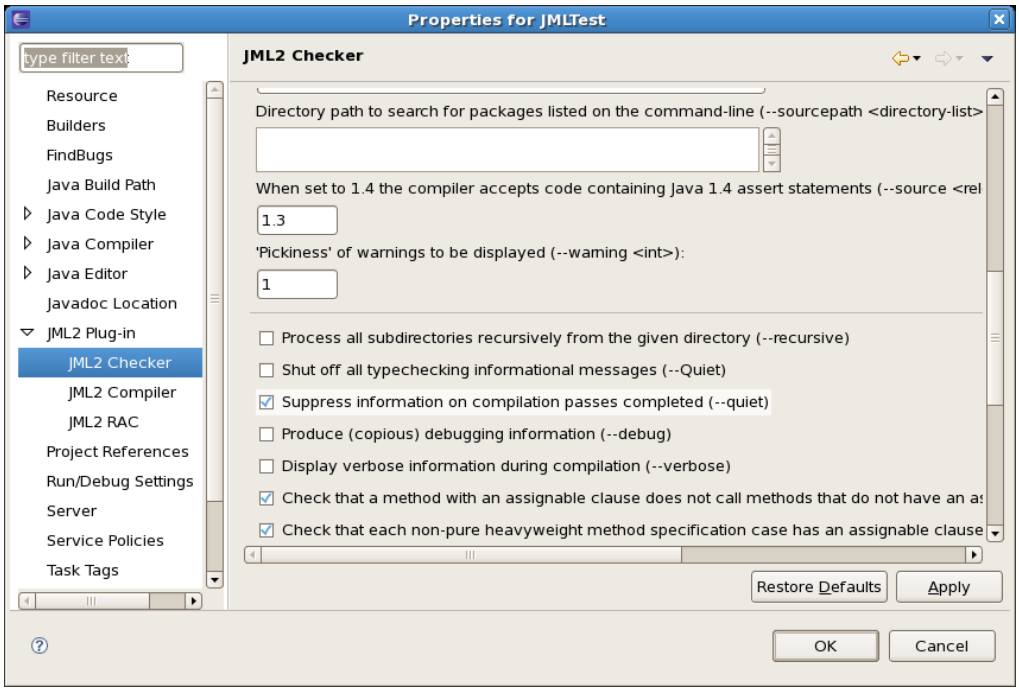
3.6.2.2 JML2 Eclipse Plug-In

Το JML2 Eclipse Plugin είναι ένα εργαλείο χρήσης των βασικών εργαλείων της JML στο περιβάλλον ανάπτυξης λογισμικού Eclipse. Με τον πιο πάνω εργαλείο μπορούν να χρησιμοποιηθούν τα βασικά εργαλεία της JML όπως το jmlrac, jmlc, jmldoc μέσα στο Eclipse IDE για καλύτερη χρήση των ιδιοτήτων της γλώσσας προδιαγραφών JML. Ακλουθούν εικόνες από την χρήση του πιο πάνω εργαλείου [35].



Σχήμα 3.15 : Παράθυρο λαθών JML2 Eclipse Plugin

Η πιο πάνω εικόνα μας δείχνει το παράθυρο λαθών που προσφέρει το συγκεκριμένο Plugin σε περίπτωση προσπάθειας ελέγχου των προδιαγραφών που έχουν οριστεί για μία ενότητα της Java.



Σχήμα 3.16 : Παράθυρο ιδιοτήτων JML2 Eclipse Plugin

Πιο πάνω βλέπουμε μία εικόνα από το παράθυρο ιδιοτήτων του JML2 plugin μπορούν να καθοριστούν τα χαρακτηριστικά που θα παρουσιάζει κατά την χρήση του.

3.6.2.3 AutoJML

Το AutoJML [31] είναι ένα εργαλείο αυτόματης παραγωγής προδιαγραφών σε JML. Παράγει αυτόματα τις προδιαγραφές βασισμένο σε μία υψηλότερου επιπέδου γλώσσα καθορισμού συμπεριφορών όπως τα Διαγράμματα καταστάσεων UML ή πρωτόκολλα προδιαγραφών ασφάλειας. Το αποτέλεσμα είναι ένας συνδυασμός από τον σκελετό του κώδικα σε Java και προδιαγραφών της κλάσης και των μεθόδων της σε JML.

3.6.2.4 Sireum/Kiasan for Java

Το Sireum/Kiasan for Java [32] είναι ένα εργαλείο αυτόματου ελέγχου προδιαγραφών JML και παραγωγής Test Cases για μονάδες προγραμμάτων Java. Το Kiasan μπορεί να επεξεργαστεί κώδικα Java τόσο με προδιαγραφές JML όσο και χωρίς. Στην περίπτωση που ο κώδικας δεν περιέχει JML τότε αυτόματα ελέγχει την πιθανότητα ύπαρξης ανεξέλεγκτου λάθους το οποίο μπορεί να προκύψει κατά την εκτέλεση του προγράμματος. Στην περίπτωση που ο κώδικας περιέχει προδιαγραφές JML τότε αναλύει τα assertions και ανιχνεύει την πιθανότητα παραβίασης τους και για preconditions, postconditions και invariants ελέγχει τα συμβόλαια μεταξύ των κλάσεων και μεθόδων.

3.7 Συμπεράσματα για την JML

Η **JML** μας προσφέρει δύο βασικά πλεονεκτήματα.

- Καθορίζει με ακρίβεια και σαφήνεια τις προδιαγραφές που χαρακτηρίζουν την αναμενόμενη συμπεριφορά μίας ενότητα Java όπως και μία πιο αυστηρή τεκμηρίωση του ίδιου του κώδικα.
- Η γλώσσα λόγω της αλληλεπίδρασης της με την Java έχει τη δυνατότητα να προσφέρει τόσο υπάρχοντα εργαλεία για υποστήριξη της ίδιας της γλώσσας όσο και την προοπτική δημιουργίας εργαλείων κομμένα και ραμμένα στις ανάγκες του κάθε προγραμματιστή.

Η JML χρησιμοποιείται συνήθως μετά το τέλος της συγγραφής του κώδικα. Η σωστή όμως χρήση της προβλέπει την ίδια τη γλώσσα να χαρακτηρίζει την τεκμηρίωση του κώδικα προτρέποντας τους προγραμματιστές αντί για τα κλασσικά σχόλια να τεκμηριώνουν τον κώδικα με κατηγορήματα και προδιαγραφές στην JML. Οι λόγοι για τους οποίους προτείνεται η ενσωμάτωση της JML κατά την διάρκεια της συγγραφής του κώδικα είναι η εξής είτε πριν είτε μετά είναι οι εξής:

- Κατά την παράδοση της τεκμηρίωσης του κώδικα στον ιδιοκτήτη μίας εφαρμογής θα ήταν ορθότερο να αποφεύγεται η πλήρης παράδοση του κώδικα για λόγους ασφάλειας των πνευματικών δικαιωμάτων αυτού. Με την παράδοση μόνο των αυστηρών, σαφών και όχι υπερβολικά συγκεκριμένων προδιαγραφών αποφεύγεται το πιο πάνω ρίσκο. Ακόμα ο πελάτης θα αποφύγει τις λεπτομέρειες της υλοποίησης π.χ. μίας βιβλιοθήκης η οποία επαναχρησιμοποιηθεί για μελλοντική χρήση σε νέες εκδόσεις της εφαρμογής.
- Ο αυστηρός καθορισμός μίας ενότητας ενός προγράμματος αναλύοντας προσεκτικά τις παραμέτρους της σχεδίασης μίας εφαρμογής είναι σίγουρο ότι θα έχει ευεργετικά αποτελέσματα στην ποιότητα του σχεδιασμού και κατ' επέκταση στην ποιότητα του ίδιου του συστήματος.
- Η χρήση των προδιαγραφών σε JML βοηθά στην προσεκτική κατανόηση της ορθότητας του προγράμματος τόσο στο στάδιο του σχεδιασμού και της υλοποίηση όσο και στην επαλήθευση του συστήματος.
- Με την ενσωμάτωση των προδιαγραφών JML σε προγραμματιστικά εργαλεία η αποσφαλμάτωση και βελτίωση του κώδικα γίνεται πιο εύκολη και πιο γρήγορη.

4 Προγραμματιστικά λάθη και προτεινόμενα πρότυπα

4.1 Εισαγωγή

Λάθη στο επίπεδο του πηγαίου κώδικα παρατηρούνται ανελλιπώς σε όλα τα μεγέθη και πολυπλοκότητες προγραμμάτων από όλα τα επίπεδα προγραμματιστών. Ένα προγραμματιστικό λάθος (software bug) είναι ο κοινός ορισμός που περιγράφει ένα σφάλμα, ψεγάδι, λάθος, αποτυχία, ή ελάττωμα σε ένα πρόγραμμα το οποίο το αποτρέπει να συμπεριφέρεται όπως αναμενόταν όταν διατυπώθηκε ο σκοπός λειτουργίας του, π.χ. να υπολογίζει λάθος ένα αποτέλεσμα ή να τερματίζει απρόσμενα. Τα περισσότερα bugs δημιουργούνται από τους ίδιους τους προγραμματιστές είτε κατά την υλοποίηση στο επίπεδο του πηγαίου κώδικα είτε στην σχεδίαση του πριν την υλοποίηση [5].

4.2 Προγραμματιστικά Λάθη

Τα προγραμματιστικά λάθη μπορούν να χωριστούν σε διάφορες κατηγορίες. Οι πιο συχνές κατηγορίες λαθών που συναντώνται είναι:

- Τα **συντακτικά λάθη**, τα οποία παρατηρούνται στο στάδιο μεταγλώττισης και ανιχνεύονται από εκάστοτε κατάλληλο μεταγλωττιστή.
- Τα **λογικά λάθη** τα οποία τα οποία λόγω της λάθος λογικής του προγραμματιστή κατά την σχεδίαση του κώδικα αποτρέπει την εκπόνηση επιθυμητών αποτελέσματα
- Τα **μαθηματικά λάθη** τα οποία παραβιάζουν τους μαθηματικούς κανόνες σε μαθηματικές πράξεις που χρησιμοποιούνται στον κώδικα.
- Τα **λάθη πόρων** ή **λάθη χρόνου εκτέλεσης** τα οποία παραβιάζουν την λογικής και ορθή χρήση των πόρων που δεσμεύει το πρόγραμμα κατά την εκτέλεση του.

Είναι σωστό να αναφέρουμε ότι υπάρχουν συντακτικά λάθη τα οποία δεν ανιχνεύονται από τους μεταγλωττιστές λόγω του ότι δεν αποτελούν λάθος της σύνταξης αλλά λάθος της χρήσης αυτής. Τα λογικά, μαθηματικά και τα λάθη χρόνου εκτέλεσης είναι αδύνατο να περιοριστούν και να ανιχνευθούν λόγω του ότι πολλές φορές δημιουργούνται δυναμικά με την εισαγωγή δεδομένων στο πρόγραμμα έτσι οι μόνες προειδοποιήσεις

που παρατηρούνται παρουσιάζονται σε ελάχιστα εργαλεία συγγραφής κώδικα υψηλού επιπέδου και αυτά με την προϋπόθεση εμφανούς ατασθαλίας.

4.3 Ταξινομία Λαθών

Μία ταξινομία , αλλιώς Taxonomy, δεν είναι απλά μια δομή για την κατηγοριοποίηση κάποιων αντιπροσωπευτικών δειγμάτων. Αναμφίβολα , ενσωματώνει την θεωρία του χώρου από τον οποίο προέρχονται αυτά τα αντιπροσωπευτικά δείγματα. Επίσης καθορίζει ποιά δεδομένα θα χρησιμοποιηθούν και με ποιό τρόπο θα χωριστούν στις ανάλογες κατηγορίες τα αντιπροσωπευτικά δείγματα.

Με την ταξινόμηση των λαθών σε κατηγορίες ομαδοποιούμε τα στοιχεία που ενεργοποιούν τα προγραμματιστικά λάθη και την συμπεριφορά που παρατηρείται μετά την συνάντηση τους. Με αυτό τον τρόπο ένας προγραμματιστής μπορεί να αντιστοιχίσει ένα λάθος σύμφωνα με την συμπεριφορά που παρουσιάζει η εκτέλεση του προγράμματος για περιορισμό του και επίλυση του.

4.3.1 Κριτήρια κατηγοριοποίησης λαθών

Για την κατηγοριοποίηση των λαθών πρέπει πρώτα να οριστούν τα κριτήρια διαχωρισμού τους βάσει εσωτερικών και εξωτερικών παραγόντων που επηρεάζουν το πρόγραμμα και φέρει τα ανεπιθύμητα αποτελέσματα κατά την εκτέλεση του.

- **Διαχείριση μνήμης :** Το συγκεκριμένο κριτήριο χαρακτηρίζει τα λάθη που προκαλούνται κατά την λανθασμένη διαχείριση της μνήμης. Η συγκεκριμένη λειτουργία αποτρέπει το πρόγραμμα να έχει πρόσβαση σε μνήμη που δεν ανήκει στον δεσμευμένο χώρο που του ανήκει ή δεν είναι προσβάσιμη.
- **Έλεγχος οριοθέτησης :** Το κριτήριο αυτό ορίζει την λειτουργία ελέγχου ορίων πινάκων όπου απαγορεύει την πρόσβαση ενός table iterator με τιμή μεγαλύτερη από την τιμή μεγέθους του στατικά δεσμευμένου πίνακα., δεδομένου του γεγονότος ότι στους πίνακες της γλώσσα που χρησιμοποιείται ορίζει αρχική διεύθυνση την θέση 0.
- **Εγκυρότητα δεδομένων :** Η εγκυρότητα των δεδομένων είναι απαραίτητη γιατί εμποδίζει την εισαγωγή δεδομένων που δεν είναι κατάλληλα για την σωστή λειτουργία του προγράμματος. Λάθη ενεργοποιούνται όταν μη έγκυρα δεδομένα

δοθούν σε αυστηρά ορισμένους περιορισμούς όπως ο τύπος των δεδομένων και το πρόσημο για μη αρνητικά μεγέθη.

- **Έλεγχος συνθήκης :** Κατά τον έλεγχο συνθήκης το Program Counter προχωρά στο σημείο που αντιστοιχεί το Boolean αποτέλεσμα μίας συνθήκης. Γι' αυτό είναι σημαντική η αποφυγή λάθος χρήσης τελεστών με το οποίο θα προκληθεί παραγωγή λανθασμένων αποτελεσμάτων, αλλαγή στη ροή εκτέλεσης του προγράμματος, ή ακόμα και ατέρμονος βρόγχος κατά την εκτέλεση του προγράμματος.
- **Υπολογισμός :** Η λειτουργία που ελέγχει την εγκυρότητα της χρήσης μαθηματικών τελεστών έτσι ώστε να συμμορφώνονται με τους κανόνες που αντιστοιχούν σε κάθε περίπτωση. Προκαλείται απρόσμενη διακοπή της εκτέλεσης του προγράμματος από την απλή παραβίαση των μαθηματικών κανόνων όπως πράξεων που έχουν αδύνατο αποτέλεσμα.

4.3.2 Κατηγοριοποίηση Λαθών

Μετά από μελέτη διαφόρων κατηγοριοποιήσεων [4] και ταξινομιών των λαθών σε αυτή την μελέτη θα χρησιμοποιηθεί ως βάση η κατηγοριοποίηση που έγινε από τον Κ. Ιωάννου [5]. Σύμφωνα και με τα πιο πάνω κριτήρια διαχωρισμού μερικά από τα πιο συχνά προγραμματιστικά λάθη κατηγοριοποιήθηκαν ως εξής.

- Λάθη Υπολογισμού(**Computation errors**)
  - Διαίρεση με το μηδέν σε μαθηματική πράξη
    Division by Zero

- Λάθη Ελέγχου Ορίων(**Boundary checking errors**)
  - Πρόσβαση σε θέση μεγαλύτερη του μεγέθους ενός πίνακα
    Array Index Out Of Range
  - Πρόσβαση σε θέση μεγαλύτερη από το μήκος μία συμβολοσειράς
    String Index Out Of Range
  - Πρόσβαση σε θέση μεγαλύτερη από το μέγεθος μίας στοίβας
    Stack Overflow Area

- Λάθη Διαχείρισης Δεδομένων(**Data Handling**)
    - Μη συντονισμένη χρήση αλληλοεξαρτημένων δεδομένων
      Callback
    - Μη ταξινομημένος πίνακας
      Unsorted Table

- Λάθη Εγκυρότητα Δεδομένων **(Data Validation)**
    - Μη αρχικοποίηση δημιουργίας αντικειμένου
      Dereferencing Null
    - Πρόσβαση σε δείκτη κενής μνήμης
      Null Pointer Exception

- Λάθη Ελέγχου(Control errors)
    - Εισαγωγή αρνητικού μεγέθους σαν θέση στον πίνακα
      Negative Array Size

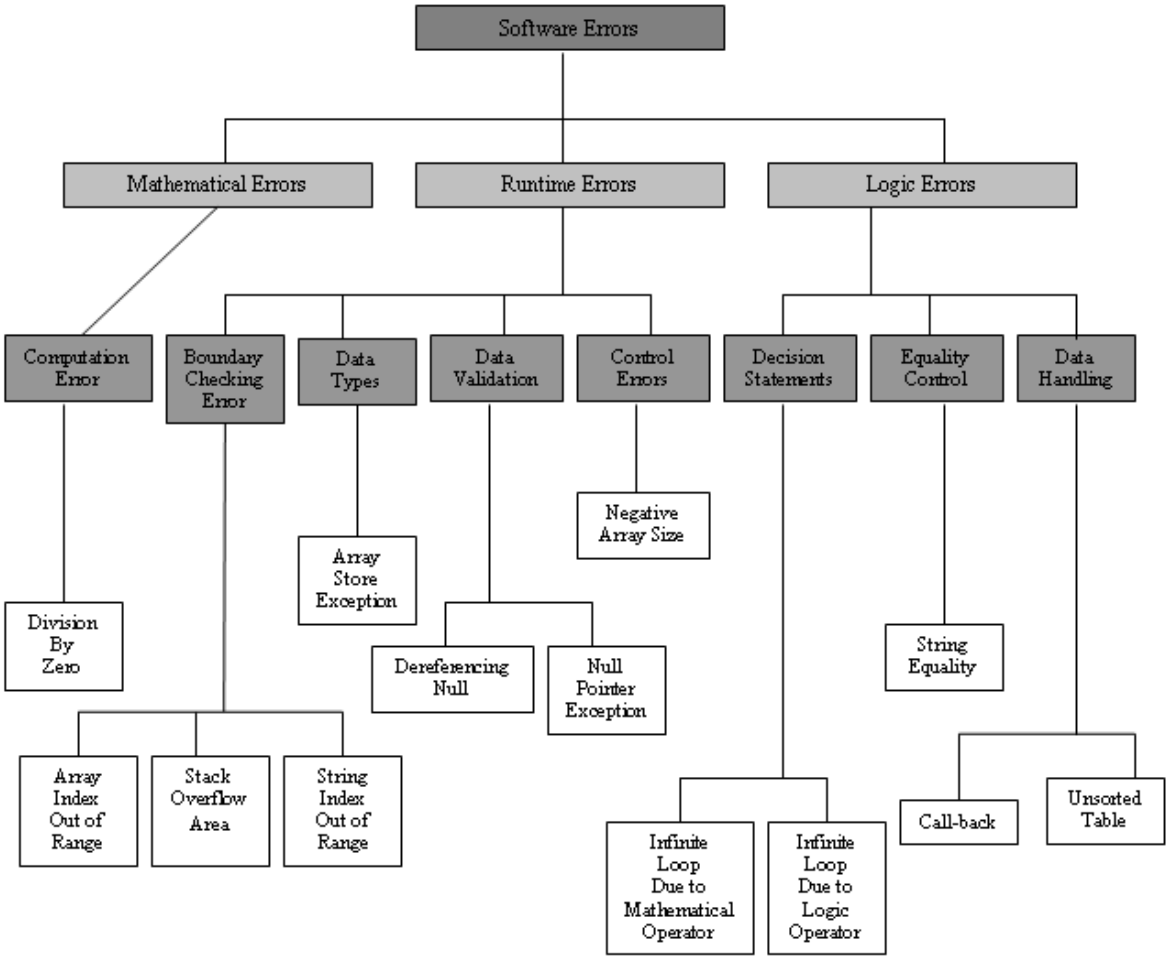
- Λάθη Αποφάσεων συνθήκης **(Decision statements)**
    - Πρόκληση ατέρμονου βρόγχου λόγω λανθασμένης χρήση λογικού τελεστή
      Infinite Loop Due To Logic Operator
    - Πρόκληση ατέρμονου βρόγχου λόγω λανθασμένης χρήση μαθηματικού τελεστή
      Infinite Loop Due To Mathematical Operator

- Λάθη Ελέγχου ισότητας **(Equality Control)**
    - Λανθασμένος έλεγχος ισότητας συμβολοσειρών
      String Equality

- Λάθη Τύπων δεδομένων **(Data Type)**
    - Καταχώρηση δεδομένων σε πίνακα μη συμβατού τύπου
      Array Store Exception

4.3.3 Δέντρο Ταξινόμησης Προγραμματιστικών Λαθών

Ο συνδυασμός των λαθών με τις κατηγορίες και υποκατηγορίες των χαρακτηριστικών τους προσφέρει την δυνατότητα σχεδιασμού ενός δέντρου για εύκολη εύρεση και κατανόηση των ταξινομημένων λαθών. Πιο κάτω παρουσιάζεται μία γενικευμένη επέκταση των διαχωρισμένων δέντρων ταξινόμησης κάτω από μία ενιαία κατηγορία, τα προγραμματιστικά λάθη.



Σχήμα 4.1 : Προτεινόμενη Ταξινόμηση Προγραμματιστικών Λαθών

4.4 Προτεινόμενα Πρότυπα Λαθών σε JML

Παρακάτω παρουσιάζονται ολοκληρωμένα τα πρότυπα προδιαγραφών των λαθών σε JML με την επεξήγηση των παραμέτρων τους όπως μελετήθηκαν στην έρευνα του Κ. Ιωάννου [5].

4.4.1 Πρότυπο για Division By Zero

Πιο κάτω παρουσιάζεται ο κώδικας σε Java και JML για το μαθηματικό λάθος που παρατηρείται από την διαίρεση ενός αριθμού με την τιμή 0.

DivisionByZero.java

```
package TemplateScrn;

public class DivisionByZero {

        //@requires a!=0;
        //@requires b!=0;
        //@ensures \result != 0;
     public/*@pure@*/ static int Divide(int a, int b){
    int apot;
      if(a>=b){
         apot = a / b;
         //assert apot!=0;
            }
     else
        {
         apot = b/a;
        //assert apot!=0;
         }
      return apot;
     }

       public static void main(String[] args) {
    /* int result=0;

      result = Divide(5,0);
      System.out.println("rer" + result);*/
      }
}
```

Σχήμα 4.2 : Πρότυπο για Division By Zero

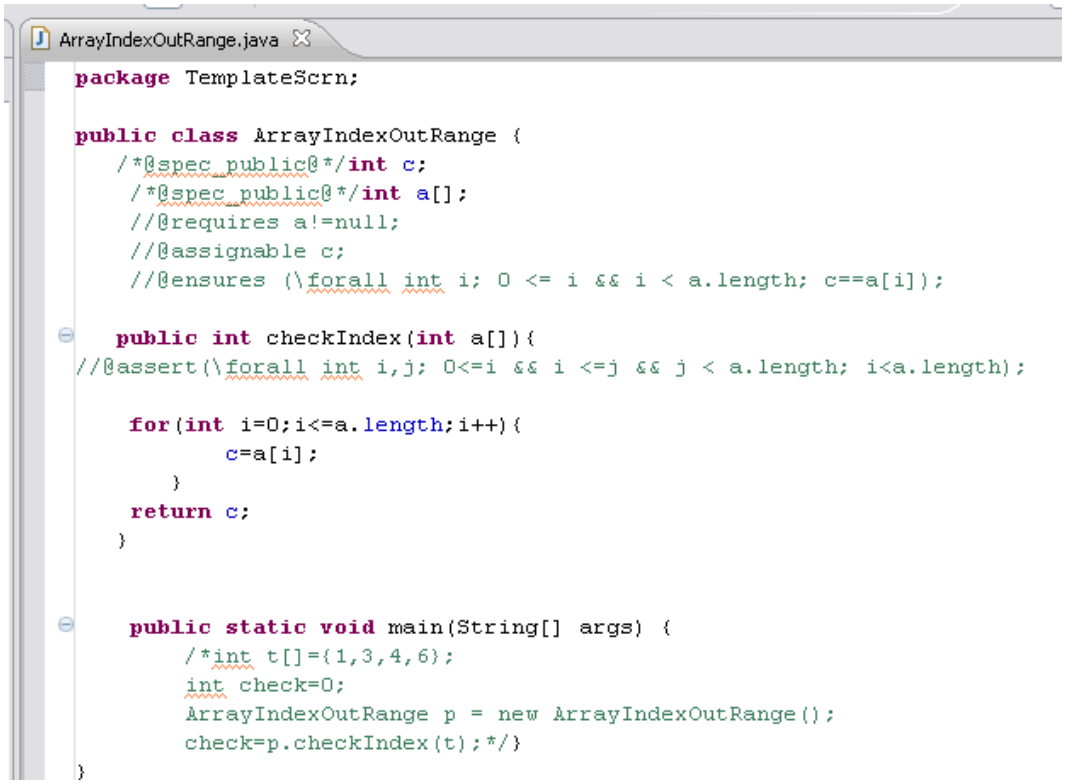
Παρατηρώντας τον πιο πάνω κώδικα παρατηρούμε ότι η συνάρτηση ανάλογα με το ποιος είναι ο μεγαλύτερος ακέραιος από τους δύο που λαμβάνει ως παραμέτρους τον τοποθετεί στον διαιρείτε και τους διαιρεί μεταξύ τους.. Λόγω της πιθανότητας διαίρεσης με το μηδέν απαιτείται να οριστεί προδιαγραφή ότι οι τιμές που δέχεται σαν παραμέτρους πρέπει να είναι άνισες με το μηδέν. Κατά τον έλεγχο της συγκεκριμένης κλάσης οι τιμές που στέλλονται στην μέθοδο ως ορίσματα είναι βέβαιο ότι θα είναι άνισες με το 0 σύμφωνα με τον κανόνα του DBC που προσφέρει η JML.

Στο συγκεκριμένο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Οι εκφράσεις **requires a!=0** και **requires b!=0** ορίζουν τα pre-conditions της μεθόδου , καθορίζοντας πώς για να εκτελεστεί η μέθοδος πρέπει να ισχύουν οι συνθήκες αυτές. Οπότε αποκλείεται να έχουμε περίπτωση διαίρεσης με το μηδέν λόγω αυτών των προδιαγραφών.
- Η έκφραση **ensures /result != 0 ,** η οποία ορίζει το post-condition της μεθόδου, και συντακτικά τοποθετείται στο σημείο μετά από τα pre-conditions. Η έκφραση αυτή εγγυάται , βάση του σχεδιασμού DBC, ότι με την εκτέλεση της μεθόδου το αποτέλεσμα θα είναι διάφορο του μηδενός.
- Οι έλεγχοι **assert** τοποθετούνται μετά από την ανάθεση τιμής στην μεταβλητή apot για επιπλέον έλεγχο και άμεση ενημέρωση τυχών λάθους στο αποτέλεσμα της .Σε περίπτωση παραβίασης της συνθήκης ελέγχου θα έχουμε διακοπή της εκτέλεσης ροής του προγράμματος.

4.4.2 Πρότυπο για Array Index Out Of Range

Πιο κάτω παρουσιάζεται ο κώδικας σε Java και JML για το λάθος χρόνου εκτέλεσης που παρατηρείται όταν ένας iterator ξεπεράσει το μέγεθος του πίνακα που ερευνά.



Σχήμα 4.3 : Πρότυπο για Array Index Out Of Range

Το πιο πάνω πρότυπο απαιτεί η τιμή του iterator για τον πίνακα που επεξεργάζεται να μην ξεπεράσει το μέγεθος του πίνακα. Όσο αφορά την περίπτωση στην οποία προκαλείται το λάθος, και συγκεκριμένα όταν η συνθήκη του βρόγχου είναι i<=a.length, αντιμετωπίζεται με την έκφραση assert της γλώσσας JML. Στην συγκεκριμένη έκφραση ελέγχου , περιορίζεται η ανάθεση τιμών στο μήκος εντός ορίων του πίνακα, αφού η συνθήκη ελέγχου περιορίζει την ανάθεση τιμών μόνο από τη θέση i=0 του πίνακα μέχρι i<a.length του πίνακα.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

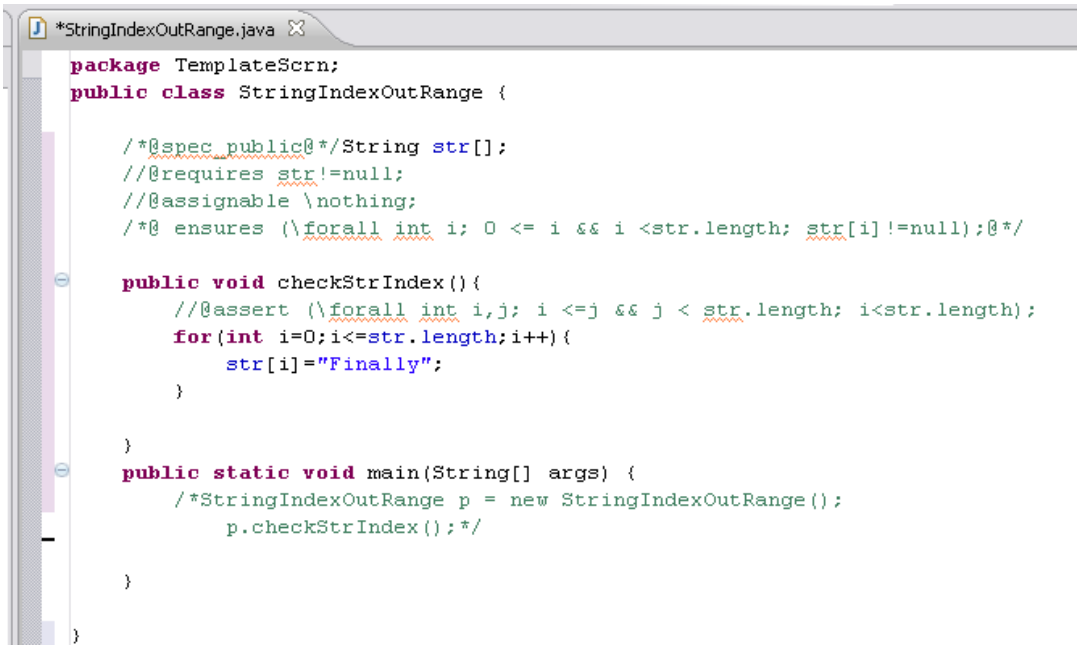
- Οι εκφράσεις **requires a!=Null**, η οποία τοποθετείται στο σημείο του κώδικα πριν να εκτελεστεί η μέθοδος με τις συγκεκριμένες παραμέτρους. Η έκφραση αυτή ορίζει το pre-condition της μεθόδου , καθορίζοντας πώς για να εκτελεστεί η μέθοδος πρέπει να

ισχύουν οι συνθήκες αυτές. Η χρησιμότητα της έκφρασης αυτής να αποφευχθεί η περίπτωση πίνακα ο οποίος να είναι κενός.

- Η έκφραση //**@ensures (\forall int i; 0 <= i && i < a.length; c!=0);** , η οποία ορίζει το post-condition της μεθόδου, και συντακτικά τοποθετείται στο σημείο όπου τοποθετούνται τα pre-conditions , δηλαδή πριν την δήλωση της μεθόδου. Η έκφραση αυτή εγγυάται , βάση του σχεδιασμού DBC, ότι με την εκτέλεση της μεθόδου το αποτέλεσμα θα είναι διάφορο του μηδενός.
- Ο έλεγχος //**@assert(\forall int i,j; i <=j && j < a.length; i<a.length);**τοποθετείται πριν από την τοποθέτηση τιμής στο μήκος του πίνακα. Στην περίπτωση αυτή περιορίζεται και ελέγχεται η ανάθεση τιμών στο πεδίο από τη θέση i=0 μέχρι i<a.length , αποκλείοντας την περίπτωση παραβίασης των ορίων του πίνακα. Στην περίπτωση αυτή , όπου i<=a.length τότε θα έχουμε παραβίαση της συνθήκης ελέγχου και θα ειδοποιηθεί ο χρήστης.
- Η εκφράσεις //**@spec_public** ορίζουν τις μεταβλητές των προδιαγραφών.

4.4.3 Πρότυπο για String Index Out Of Range

Πιο κάτω παρουσιάζεται ο κώδικας σε Java και JML για το λάθος χρόνου εκτέλεσης που παρατηρείται όταν προσπαθούμε να επεξεργαστούμε δεδομένα σε θέση σε ένα String η οποία είναι μεγαλύτερη από το μέγεθος του.



Σχήμα 4.4 : Πρότυπο για String Index Out Of Range

Το πρότυπο απαιτεί για την εκτέλεση της συγκεκριμένης μεθόδου , η οποία προκαλεί αυτού του είδους το λάθος ,η παράμετρος της μεθόδου να μην είναι Null.Όσο αφορά την περίπτωση αναίρεσης του λάθους αντιμετωπίζεται με την εντολή ελέγχου της JML , το assert, στην οποία περιορίζεται η ανάθεση τιμών από τη θέση i=0 του πίνακα μέχρι i<str.length του πίνακα.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **requires str!=null** , η οποία τοποθετείται στο σημείο του κώδικα πριν να εκτελεστεί η μέθοδος με τις συγκεκριμένες παραμέτρους. Η έκφραση αυτή ορίζει το pre-condition της μεθόδου , καθορίζοντας πώς για να εκτελεστεί η μέθοδος πρέπει να ισχύουν οι συνθήκες αυτές. Η χρησιμότητα της έκφρασης αυτής να αποφευχθεί η περίπτωση πίνακα γραμματοσειράς ο οποίος να είναι κενός.

- Η έκφραση //**@ensures (\forall int i; 0 <= i && i < str.length; str!=null);** , η οποία ορίζει το post-condition της μεθόδου, και συντακτικά τοποθετείται στο σημείο όπου τοποθετούνται τα pre-conditions , δηλαδή πριν την δήλωση της μεθόδου. Η έκφραση αυτή εγγυάται , βάση του σχεδιασμού DBC, ότι με την εκτέλεση της μεθόδου η ανάθεση έχει όντος τιμή και δεν θα είναι κενή.
- Ο έλεγχος /***@assert (\forall int i; 0 <= i && i < a.length; c==a[i]);@***/ τοποθετείται πριν από την τοποθέτηση τιμής στο μήκος του πίνακα. Στην περίπτωση αυτή περιορίζεται και ελέγχεται η ανάθεση τιμών στο πεδίο από τη θέση i=0 μέχρι i<str.length , αποκλείοντας την περίπτωση παραβίασης των ορίων του πίνακα. Στην περίπτωση αυτή , όπου i<=str.length τότε θα έχουμε παραβίαση της συνθήκης ελέγχου και θα ειδοποιηθεί ο χρήστης.
- Η έκφραση //**@assignable** ορίζει τις μεταβλητές στις οποίες μπορούν να ανατεθούν τιμές. Στην προκειμένη περίπτωση δεν υπάρχει διότι ορίζεται με το συντελεστή \nothing.

4.4.4 Πρότυπο για Array Store Exception

Στην περίπτωση αυτού του λάθους ,επιχειρείται προσπάθεια ανάθεσης τιμή σε κάποια θέση ενός πίνακας η οποία τιμή δεν έχει συμβατό τύπο με τον τύπο ορισμού του ίδιου του πίνακα.

*Arraystorexception.java

```
package TemplateScrn;

public class Arraystorexception {

    /*@spec_public non_null@*/ Object[] elems;
    /*@spec_public@*/ int top;
    //@invariant 0 <= top && top <= elems.length && \typeof(elems) == \type(Object[]
    //@ requires top < elems.length;
    //@requires o!=null;
    /*@requires \typeof(o) <: \elemtype(\typeof(elems));@*/

    public void add(Integer o) {
        elems[top++] = o;

    }
    public static void main(String[] args) {
        /*int t=0;
        Arraystorexception a = new Arraystorexception();
        a.add(t);
         */
    }
}
```

Σχήμα 4.5 : Πρότυπο για Array Store Exception

Για παράδειγμα , στην πιο πάνω μέθοδο δημιουργείται ένας πίνακας elems, τον τύπο του οποίου ορίζουμε να είναι string και στην συνέχεια προσπαθούμε να αναθέσουμε στις θέσεις του πίνακα αυτού ακέραιους αριθμούς. Φυσικά αυτή η διαδικασία δεν είναι έγκυρη και θα αποφέρει το λάθος. Για την αποφυγή του λάθους αυτού , ορίζεται στις προδιαγραφές της μεθόδου ένα Invariant, το οποίο καθορίζει ότι ο πίνακας elems ,όταν δημιουργηθεί θα έχει τύπο δεδομένων ίδιο με αυτό που ορίζεται όταν δημιουργείται , για παράδειγμα String[] elems ,θα είναι τύπου string.Η έκφραση που μας ενδιαφέρει είναι το pre-condition το οποίο προδιαθέτει ότι για σωστή εκτέλεση της μεθόδου είναι αναγκαίο ο τύπος δεδομένου του αντικειμένου που θα ανατεθεί στον πίνακα, να είναι του ιδίου τύπου με τον τύπο δεδομένων του πίνακα. Με τον τρόπο αυτό αποφεύγονται μη συμβατές αναθέσεις τύπων δεδομένων.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **//@invariant 0 <= top && top <= elems.length && \typeof(elems) == \type(Object[]);**, η οποία τοποθετείται στο σημείο του κώδικα πριν να εκτελεστεί η

μέθοδος με τις συγκεκριμένες παραμέτρους. Η έκφραση αυτή ορίζει συνθήκη η οποία αφορά τις μεταβλητές των προδιαγραφών και απαιτείται να έχει τιμή true , δηλαδή να ικανοποιείται κατά την εκτέλεση της μεθόδου την οποία προδιαγράφει και συγκεκριμένα με την δημιουργία του κατασκευαστή της μεθόδου. Η έκφραση //**@ requires top < elems.length;**, η οποία ορίζει ένα από τα pre-conditions της μεθόδου , απαιτεί ότι η μεταβλητή top του πίνακα να μην ξεπερνά το μήκος του πίνακα.

- Η έκφραση **/*@requires \typeof(o) <: \elemtype(\typeof(elems));@*/** η οποία ορίζει το άλλο από τα pre-conditions της μεθόδου , απαιτεί ότι ο τύπος δεδομένων του αντικειμένου O , στην περίπτωση μας, πρέπει να είναι του ίδιου τύπου με τον τύπο δεδομένων του πίνακα , στον οποίο θα γίνει η ανάθεση , έτσι ώστε να μην υπάρχει πρόβλημα ανάθεσης λόγω μη συμβατών τύπων δεδομένων.
- Η εκφράσεις **/*@spec_public non_null@*/** ορίζουν τις μεταβλητές των προδιαγραφών και ταυτόχρονα δηλώνει πώς δεν πρέπει να είναι κενές(Null), δηλαδή πρέπει να δημιουργηθεί το εν λόγω αντικείμενο.

4.4.5 Πρότυπο για Dereferencing Null

Στην περίπτωση αυτή , ελέγχεται πιθανή δημιουργία Null δείκτη , όπως ορισμός αντικειμένου και προσπάθεια χρήσης του , χωρίς να γίνει δημιουργία του με τον ορθό τρόπο, για παράδειγμα string example[] = new string[2];

*DereferencingNull.java

```
package TemplateScrn;
public class DereferencingNull {
    /*@spec_public@*/  int size;
    /*@non_null*/ int[] elements;
    //@invariant 0<=size && size <= elements.length;
    //@requires input!=null;
    public void adding(int[] input) {
        size = input.length;
        elements = new int[size];
        System.arraycopy(input,0,elements,0,size);
        for(int i =0;i<input.length;i++){
            System.out.print(" " + input[i]);}
    }
    public int extractMin() {
        int min =Integer.MAX_VALUE;
        int minIndex = 0;
        for (int i=0; i <size; i++) {
            if (elements[i] < min) {
                min = elements[i];
                //@assert min>0;
                minIndex = i;
            }
        }
        size--;
        elements[minIndex]=elements[size];
        return min;
    }
    public static void main(String[] args) {
        /*System.out.println("Entering");
        int table[]=null;
        DereferencingNull t = new DereferencingNull();
        t.adding(table);*/
    }
}
```

Σχήμα 4.6 : Πρότυπο για Dereferencing Null

Για την περίπτωση αυτού του λάθους , η λύση είναι πολύ απλή. Ορίζοντας όπως προδιαγραφές όπως ότι το συγκεκριμένο αντικείμενο πρέπει να είναι άνισο του null ,αποφεύγουμε αυτό το λάθος. Το ίδιο αποτέλεσμα θα έχουμε εάν χρησιμοποιήσουμε την ειδική έκφραση όπως JML, η οποία είναι το non_null , για το συγκεκριμένο αντικείμενο το οποίο θέλουμε να μην είναι Null.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση //**@invariant 0<=size && size <= elements.length;**, η οποία τοποθετείται στο σημείο του κώδικα πριν να εκτελεστεί η μέθοδος με όπως συγκεκριμένες παραμέτρους. Η έκφραση αυτή ορίζει συνθήκη η οποία αφορά όπως μεταβλητές των προδιαγραφών και απαιτείται να έχει τιμή true , δηλαδή να ικανοποιείται κατά την εκτέλεση όπως μεθόδου την οποία προδιαγράφει και συγκεκριμένα με την δημιουργία του κατασκευαστή όπως μεθόδου. Στη συγκεκριμένη JML έκφραση ορίζεται ότι η μεταβλητή size, η οποία χρησιμοποιείται για την θέση του πίνακα δεν θα ξεπεράσει τα όρια του πίνακα και όπως ο τύπος δεδομένων του πίνακα θα είναι ο τύπος με τον οποίο θα δηλωθεί.
- Η έκφραση //**@requires input!=null;**, η οποία ορίζει το pre-condition όπως μεθόδου , απαιτεί ότι η μεταβλητή input, η οποία δηλώνει την δημιουργία πίνακα που θα χρησιμοποιηθεί για σκοπούς λειτουργίας την μέθοδο, να μην είναι κενός, δηλαδή να δημιουργηθεί ο πίνακας και να αρχικοποιηθεί πριν την χρήση του.
- Η εκφράσεις /***@non_null***/ορίζουν τις μεταβλητές των προδιαγραφών και ταυτόχρονα δηλώνει πώς δεν πρέπει να είναι κενές(Null), δηλαδή πρέπει να δημιουργηθεί το εν λόγω αντικείμενο. Βασικά λειτουργά όπως το προαναφερόμενο pre-condition απλώς είναι διαφορετική η σύνταξη όπως έκφρασης.

4.4.6 Πρότυπο για Stack Overflow Area

Στην περίπτωση αυτή , ελέγχεται πιθανή υπερχείλιση όπως στοίβας , η οποία χρησιμοποιείται για την αποθήκευση μεταβλητών και παραμέτρων , όπως αναφέραμε στην περιγραφή του εν λόγω προβλήματος με το χαρακτηριστικό παράδειγμα του παραγοντικού.

*StackOverflowArea.java

```
package TemplateScrn;
public class StackOverflowArea {
    /*@requires 1 <= n ;
      @ensures \result== (\product int i ; 1<= i && i<=n ;i) && \result < 6227020800;
      @signals (ArithmeticException) n > 12 ;
     */

    public static int factorial(int n){
        if ( n > 12 ) throw new ArithmeticException();
        else {
            int f = 1;
            int i = 1;
            /*@loop_invariant i<=n && f ==(\product int j ; 1 <= j && j <= i ;j);@*/
            while ( i < n ) {
                i = i + 1 ;
                f = f * i ;
            }
            return f ;

        }
    }
    public static void main(String[] args){
        int n =11;
        System.out.println("Factorial of" + n + "=" + factorial(n));
    }
}
```

Σχήμα 4.7 : Πρότυπο για Stack Overflow Area

Για την επίλυση του προβλήματος αυτού έχουμε υπόψη μας πως δεδομένου ότι η γλώσσα Java δεν έχει την συγκεκριμένη εξαίρεση Arithmetic Overflow , η οποία προκαλείται όταν έχουμε μεγάλη τιμή ακεραίου αριθμού και δεν χωρεί στην στοίβα, ο υπολογισμός οποιασδήποτε αξίας ακέραιου (int) που υπερβαίνει 2^{31} - 1 = 2147483647 θα δώσει ένα ανακριβές αποτέλεσμα. Αυτό συμβαίνει διότι η μεγαλύτερη τιμή που μπορεί να πάρει ένας ακέραιος αριθμός(int) n , για το οποίο n! < 2^{31} – 1 , είναι το 12.Όπως διαφαίνεται στον πιο πάνω γράφο , περιορίζουμε αυτή την περίπτωση πρόκλησης λάθους , με την εισαγωγή της JML έκφρασης **/*@requires n<13;** , η οποία σε περίπτωση εισαγωγής αριθμού μεγαλύτερου από το επιτρεπόμενο όριο θα ανιχνεύσει το λάθος. Επιπλέον γίνεται έλεγχος στο post-condition της μεθόδου όπου ελέγχεται ότι το παραγόμενο αποτέλεσμα δεν ξεπερνά

την τιμή του 6227020800 , η οποία είναι η τιμή υπολογισμού για το n! όπου n=13 , η οποία θα δώσει λανθασμένα αποτελέσματα.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **/*@requires 1 <= n; ,**η οποία ορίζει το pre-condition της μεθόδου , απαιτεί ότι η μεταβλητή n η οποία θα δοθεί σαν παράμετρος στην μέθοδο για να υπολογιστεί το παραγοντικό της, πρέπει να είναι οπωσδήποτε θετικός αριθμός και μικρότερος του 12 διότι όπως προαναφέραμε μεγαλύτερη τιμή που μπορεί να πάρει η παράμετρος είναι το 11 για τους λόγους που εξηγήσαμε πιο πάνω.
- Η έκφραση **@ensures \result== (\product int i ; 1<= i && i<=n ;i); &&\result<6227020800**ορίζει το post-condition της μεθόδου και διαβεβαιώνει ότι το τελικό αποτέλεσμα θα είναι το γινόμενο των αριθμών από i=1 μέχρι i<=n , βασικά της μορφής

$$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}.$$

- Η έκφραση **/*@loop_invariant i<=n && f ==(\product int j ; 1 <= j && j <= i ;j);@*/**, η οποία τοποθετείται στο σημείο του κώδικα πριν να εκτελεστεί η συνθήκη επανάληψη στου βρόγχου , ελέγχει και πιστοποιεί ότι σε κάθε επανάληψη του βρόγχου θα τηρούνται οι συνθήκες υπολογισμού του παραγοντικού του εν λόγω αριθμού έτσι ώστε να εκτελεστεί ορθά η διαδικασία για τον υπολογισμό του αποτελέσματος.

4.4.7 Πρότυπο για Unsorted Table

Στην συγκεκριμένη περίπτωση αντιμετωπίζουμε πιθανό πέρασμα παραμέτρων στην μέθοδο η οποία εκτελεί το Binary Search , και ο πίνακας ο οποίος αποτελεί την παράμετρο της μεθόδου να μην είναι ταξινομημένος.

UnsortedTable.java

```
package TemplateScrn;
public class UnsortedTable {
    /** Creates a new instance of UnsortedTable */
    public UnsortedTable() {
    }
    public int i;
    //@requires A!=null;
    //@requires A.length > 0;
    //@ requires(\forall int i, j;0 <= i && i < j && j < A.length; A[i] <= A[j]);
    //@ensures (\forall int i ;0 <= i &&  i < A.length ==> \old(A[i]== N)) ? (\result==i)
    public  int binarySearch(int[] A, int N) {

        int lowestPossibleLoc = 0;
        int highestPossibleLoc = A.length - 1;
        int middle = 0;

        while (highestPossibleLoc >= lowestPossibleLoc) {
            middle = (lowestPossibleLoc + highestPossibleLoc) / 2;
            if (A[middle] == N) {
                // N has been found at this index!
                return middle;
            } else if (A[middle] > N) {
                // eliminate locations >= middle
                highestPossibleLoc = middle - 1;

            } else {
                // eliminate locations <= middle
                lowestPossibleLoc = middle + 1;

            }
        }
        return -1;
    }
```

Σχήμα 4.8 : Πρότυπο για Unsorted Table

Αυτό το σχεδιαστικό πρότυπο αναφέρεται συγκεκριμένα για την μέθοδο εκτέλεσης του αλγόριθμου Binary Search, ο οποίος βρίσκει το στοιχείο N μέσα στον πίνακα με τον ιδιόρρυθμο τρόπο ανίχνευσης. Προϋπόθεση εκτέλεσης είναι ο πίνακας να είναι ταξινομημένος σε αύξουσα σειρά αλλιώς δεν θα εκτελεστεί με ορθό τρόπο η μέθοδος και θα έχουμε λάθος αποτελέσματα. Αυτή η προϋπόθεση είναι εγγυημένη από το precondition , το οποίο ορίζει ο πίνακας να είναι ταξινομημένος σε αύξουσα σειρά. Επίσης με το τέλος της εκτέλεσης εγγυάται ότι εάν βρεθεί το στοιχείο N θα επιστραφεί η θέση του αλλιώς θα επιστραφεί η τιμή -1, ένδειξη ότι δεν βρέθηκε το στοιχείο αυτό μέσα στον πίνακα.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **//@requires A!=null;** και **//@requires A.length > 0;**, οι οποίες ορίζουν τα pre-conditions της μεθόδου , απαιτούν όπως ο πίνακας που θα περασθεί σαν παράμετρος στην μέθοδο να μην είναι κενός.
- Η έκφραση **//@ requires(\forall int i, j;0 <= i && i < j && j < A.length; A[i] <= A[j]);** ορίζει ότι προϋπόθεση για εκτέλεση της μεθόδου είναι η ταξινόμηση του πίνακα, δηλαδή ο πίνακας πρέπει να είναι ταξινομημένος σε αύξουσα σειρά για την ορθή λειτουργία της μεθόδου.
- Η έκφραση **//@ensures (\forall int i ;0 <= i && i < A.length ==> \old(A[i]== N)) ? (\result==i) : (\result== -1);** η οποία αποτελεί το post-condition της μεθόδου τοποθετείται στο αρχικό μπλοκ εντολών , πριν την δήλωση και κατόπιν εκτέλεση της μεθόδου. Η έκφραση αυτή επιβεβαιώνει πώς με την εκτέλεση της μεθόδου , το τελικό αποτέλεσμα θα είναι η θέση του στοιχείου που αναζητείται μέσα στον πίνακα, αλλιώς αρνητική τιμή ως ένδειξη πώς δεν βρέθηκε το στοιχείο αυτό στον πίνακα.

4.4.8 Πρότυπο για Infinite Loop due to mathematical operator

Στην προκειμένη περίπτωση αντιμετωπίζουμε τη δημιουργία άπειρου βρόγχου λόγω χρήσης λανθασμένου μαθηματικού τελεστή.

InfiniteLoop.java

```
package TemplateScrn;
public class InfiniteLoop{

    //@requires number > 0;
    //@requires num > 0;
    //@assignable number;

    public int Compute(int number, int num){

        /*@assert(\forall int i; 0<=i &&
          *@ i<(\old(number)-\old(num));
          *@ number==\old(number)-1);
          *@*/
        while(number > num){
            number++;

            System.out.println("Done");
        }
        return number;
    }
    public static void main(String[] args) {
        //InfiniteLoop p =new InfiniteLoop();
        //int k =p.Compute(7,5);

    }

}
```

Σχήμα 4.9 : Πρότυπο για Infinite Loop due to mathematical operator

Στην προκειμένη περίπτωση παρατηρείται άπειρος βρόγχος διότι ενώ έπρεπε να γίνεται μείωση του αριθμού number κατά μια μονάδα για να μπορέσει σε κάποια φάση να γίνει μικρότερος από τον num και να τερματίσει η συνθήκη , έγινε λανθασμένη χρήση του τελεστή. Αυτό αποφεύγεται από την έκφραση της JML την οποία προσθέσαμε και συγκεκριμένα το assert, η οποία καθορίζει πώς πρέπει να γίνεται μείωση του αριθμού κατά μία μονάδα για την ομαλή και ορθή εκτέλεση της μεθόδου.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **//@requires number > 0;** και **//@requires num > 0;**, οι οποίες ορίζουν τα pre-conditions της μεθόδου , απαιτούν όπως και οι δύο αριθμοί να είναι θετικοί.

- Η έκφραση **//@assignable number;** ορίζει ότι στην μεταβλητή προδιαγραφών Number μπορεί να ανατεθεί τιμή.
- Η έκφραση **/*@assert(\forall int i; 0<=i && i<(\old(number)-\old(num)); number==\old(number)-1);@*/** η οποία αποτελεί το post-condition της μεθόδου τοποθετείται στο αρχικό μπλοκ εντολών , πριν την δήλωση και κατόπιν εκτέλεση της μεθόδου και μετά από τις δηλώσεις οποιονδήποτε άλλων εντολών των προδιαγραφών. Η έκφραση αυτή επιβεβαιώνει πώς με την εκτέλεση του βρόγχου επανάληψης , όπου οι επαναλήψεις θα είναι όσες και το αποτέλεσμα της διαφοράς number – num , η μεταβλητή number θα μειώνεται κατά μια μονάδα. Σε αντίθετη περίπτωση θα έχουμε παραβίαση της συνθήκης ελέγχου και θα έχουμε αποτυχία εκτέλεσης.

4.4.9 Πρότυπο για Infinite Loop due to Logical operator

Στην προκειμένη περίπτωση ,παρατηρούμε εξαγωγή λανθασμένων και απρόσμενων αποτελεσμάτων , λόγω χρησιμοποίησης του τελεστή < αντί για το >, γεγονός που οφείλεται σε απροσεξία εκ μέρους του προγραμματιστή.

InfiniteLoopException.java

```
package TemplateScrn;
public class InfiniteLoopException {

    public int count;
    private/*@spec_public@*/ int i;
    /*@ requires ia != null;
    @assignable ia[*];
    @assignable count;
    @ensures(\forall int i; 0<=i &&  i <ia.length ==> (\old(ia[i]) >0) && (\forall int j;

    public int negatefirst(int ia[]) {

        for( int i = 0; i < ia.length; i++) {
            if (ia[i] < 5) {
                count++;

            }
            else{
                ia[i] = -ia[i]; }
        }
        return count;
    }


    public static void main(String[] args) {
        /*  int res=0;
        int pin[]={1,2,3,6,7};
        InfiniteLoopException inf = new InfiniteLoopException();
        res=inf.negatefirst(pin);
        System.out.println("This is" +" "+res);
         */
    }
}
```

Σχήμα 4.10 : Πρότυπο για InfiniteLoop due to Logical operator

Στην προκειμένη περίπτωση ο προγραμματιστής σκόπευε να υπολογίσει το σύνολο των αριθμών οι οποίο θα είναι μικρότεροι του 5 , και να μηδενίσει αυτούς οι οποίοι δεν είναι μικρότεροι του. Τα αποτελέσματα του θα είναι λάθος λόγω του τελεστή που χρησιμοποίησε την συνθήκη ελέγχου. Η προσθήκη του post-condition , αναιρεί αυτό το λάθος , διότι εγγυάται ότι όταν ο αριθμός είναι μεγαλύτερος από το 5 ,τότε θα αυξάνεται ο μετρητής count.Ενώ στο λανθασμένο παράδειγμα δεν γίνεται αυτό λόγω της χρήσης λανθασμένου τελεστή.

Να αναφέρουμε ότι στην θέση του 5 , μπορεί να χρησιμοποιηθεί οποιοσδήποτε αριθμός , απλά ο αριθμός αυτός επιλέχθηκε τυχαία για την εκτέλεση του παραδείγματος.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **/*@ requires ia != null;**, η οποία ορίζει το pre-condition της μεθόδου , απαιτούν όπως ο πίνακας που θα περασθεί σαν παράμετρος στην μέθοδο να μην είναι κενός.
- Οι εκφράσεις **@assignable ia[*];** και **@assignable count;** ορίζουν ότι στις δύο συγκεκριμένες μεταβλητές προδιαγραφών μπορούν να ανατεθούν τιμές. Οι δηλώσεις αυτές τοποθετούνται στο αρχικό μπλοκ των εντολών προδιαγραφών μετά τις δηλώσεις των pre-conditions.
- Η έκφραση **@ensures(\forall int i; 0<=i && i <ia.length ==> (\old(ia[i]) >0) && (\forall int j; 0<=j && j<i ==> (\old(ia[j])>5))) ? (count==count+1) : (ia[i]== \old(ia[i])); @*/** η οποία αποτελεί το postcondition της μεθόδου τοποθετείται στο αρχικό μπλοκ εντολών , πριν την δήλωση και κατόπιν εκτέλεση της μεθόδου και μετά από τις δηλώσεις οποιονδήποτε άλλων εντολών των προδιαγραφών. Η έκφραση αυτή επιβεβαιώνει πώς με την εκτέλεση της μεθόδου , η ροή εκτέλεσης του προγράμματος θα εκτελεστεί ορθά , ενώ στην περίπτωση παραβίασης θα έχουμε αποτυχία.

4.4.10 Πρότυπο για Callback

Στην προκειμένη περίπτωση αντιμετωπίζουμε τη μη συντονισμένη χρήση αλληλοεξαρτημένων δεδομένων με συνέπεια την παραγωγή λανθασμένων αποτελεσμάτων.

*Callback.java

```
package TemplateScrn;
public class Callback {
    private/*@spec_public@*/int k;
    private/*@spec_public@*/ int m;
    /*@ invariant
        @ k + m == 0;
        @*/
    /*@ normal_behavior
        @ requires true;
        @ assignable k,m;
        @ ensures k == \old(k) - 1 &&
        @ m == \old(m) + 1;
        @*/
    public void decrementk() { k--; m++;}
    B b;
    /*@ normal_behavior
        @ requires b != null;
        @ assignable k,m;
        @ ensures true;
        @*/
    public void incrementk() { k++; b.go(this); m--;}
    class B {
        /*@ normal_behavior
            @ requires b != null;
            @ assignable k,m;
            @ ensures true;
            @*/
        void go(Callback arg) {

            arg.decrementk(); }
    }
```

Σχήμα 4.11 : Πρότυπο για Callback

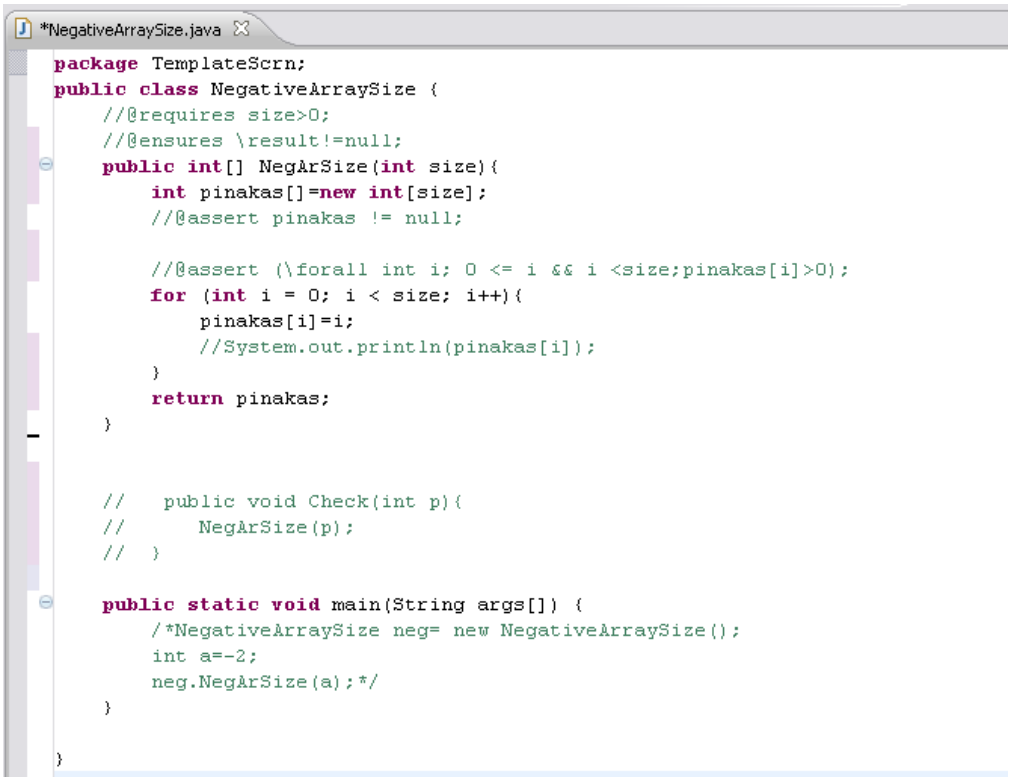
Στην πιο πάνω περίπτωση αντιμετωπίζεται λανθασμένη σειρά χρήσης μεταβλητών. Στο αρχικό πρόγραμμα-εδώ φαίνεται μια μέθοδος μόνο- υπάρχουν δύο μέθοδοι, μία για αύξηση της μεταβλητής m και μια για μείωση της μεταβλητής k. Υπάρχει περίπτωση κλήσης των δύο μεθόδων με τέτοιο τρόπο έτσι ώστε να μην ελέγχεται η αρχική συνθήκη, η οποία προβλέπει ότι η πρόσθεση των δύο μεταβλητών θα δίνει μηδέν. Αυτό το λάθος αποφεύγεται με την διαπίστωση από το postcondition ότι μετά το τέλος εκτέλεσης της μεθόδου ,όντος θα ισχύει ότι το άθροισμα των δύο μεταβλητών θα δίνει μηδέν.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **/*@ invariant k+m==0;**, η οποία ορίζει πώς κατά την διάρκεια εκτέλεσης της μεθόδου πρέπει η συνθήκη αυτή να ισχύει. Τοποθετείται σαν πρώτη εντολή στο αρχικό μπλοκ των εντολών προδιαγραφών.
- Η έκφραση **//@ensures k==\old(k)-1 && m==\old(m)+1** η οποία αποτελεί το postcondition της μεθόδου τοποθετείται στο αρχικό μπλοκ εντολών , πριν την δήλωση και κατόπιν εκτέλεση της μεθόδου και μετά από τις δηλώσεις οποιονδήποτε άλλων εντολών των προδιαγραφών. Η έκφραση αυτή επιβεβαιώνει πώς με την εκτέλεση της μεθόδου , η ροή εκτέλεσης του προγράμματος θα εκτελεστεί ορθά , ενώ στην περίπτωση παραβίασης θα έχουμε αποτυχία. Πιο συγκεκριμένα για την περίπτωση μας η μεταβλητή κ θα μειώνεται κατά μια μονάδα ενώ η μεταβλητή m θα αυξάνεται κατά μια μονάδα.

4.4.11 Πρότυπο για Negative Array Size

Στην προκειμένη περίπτωση αντιμετωπίζουμε τη δημιουργία πίνακα με αρνητικό μέγεθος.



Σχήμα 4.12 : Πρότυπο για Negative Array Size

Όπως γνωρίζουμε δεν είναι εφικτή η δημιουργία πίνακα με αρνητικό μέγεθος αλλά υπάρχουν περιπτώσεις που κάποιοι άπειροι προγραμματιστές τείνουν να το επιχειρούν. Για να το αποτρέψουμε αυτό , ορίζουμε σαν συνθήκη εκτέλεσης της μεθόδου αυτής στην οποία δημιουργείται πίνακας ακεραίων , ότι το μέγεθος του πίνακα πρέπει να είναι θετικό , αλλιώς δεν εκτελείται η μέθοδος. Επιπρόσθετα γίνεται έλεγχος και μετά την δημιουργία του πίνακα ότι όντος είναι θετικό το μέγεθος του, έτσι ώστε να αποφευχθεί το συγκεκριμένο λάθος.

Στο επικείμενο προτεινόμενο πρότυπο ανίχνευσης του συγκεκριμένου λάθους χρησιμοποιούνται οι εξής εκφράσεις JML:

- Η έκφραση **/*@ requires size > 0;**, η οποία ορίζει το pre-condition της μεθόδου , απαιτεί όπως το μέγεθος το οποίο ορίζεται σαν το μέγεθος του πίνακα που δημιουργείται να μην είναι αρνητικό. Όπως όλα τα pre-condition ορίζεται στην αρχή του μπλοκ , πριν από τις άλλες εντολές.

- Οι εκφράσεις //**@ensures \result!=null;** η οποία αποτελεί το postcondition της μεθόδου τοποθετείται στο αρχικό μπλοκ εντολών , πριν την δήλωση και κατόπιν εκτέλεση της μεθόδου και μετά από τις δηλώσεις οποιονδήποτε άλλων εντολών των προδιαγραφών. Δηλώνει πώς το τελικό αποτέλεσμα θα έχει τιμή.
- Η έκφραση //**@assert (\forall int i; 0 <= i && i <size;pinakas[i]>0);** αυτή ελέγχει την διαδικασία ανάθεσης τιμών στον πίνακα. Νοουμένου ότι το μέγεθος του πίνακα δεν είναι αρνητικό , γεγονός που ελέγχεται από το pre-condition , θα γίνει έλεγχος για τα στοιχεία ανάθεσης να μην είναι αρνητικά. Τοποθετείται πριν από την εκτέλεση του βρόγχου επανάληψης για να μπορεί να ελέγχει τα στοιχεία του πίνακα κάθε φορά που γίνεται η ανάθεση.

4.5 Συμπεράσματα

Μελετώντας τα λάθη και τα πρότυπα που προτείνονται είναι εύκολο να ανιχνεύσουμε και άλλα λάθη τα οποία μπορούν μπουν σε πρότυπα JML. Η δημιουργία προτύπων είναι ένα σημαντικό μέρος της τεχνολογίας λογισμικού εφόσον προσφέρει αυτοματοποίηση του κώδικα το οποίο σημαίνει πιο εύκολη και παραγωγική συγγραφή. Επιπλέον με την χρήση προτύπων στην JML είναι πιο εύκολο για κάποιο αρχάριο στις προδιαγραφές λογισμικού να εισάγει τον απαραίτητο κώδικα για αυστηρή τεκμηρίωση της συμπεριφοράς που αναμένει. Ακόμα σε περίπτωση αναζήτησης των λαθών στον κώδικα είναι πιο εύκολο για ένα προγραμματιστή να εισάγει αυτά τα πρότυπα μαζί έτσι ώστε να μπορεί να ελέγχει ταυτόχρονα όλες τις κύριες αιτίες λαθών στον κώδικα του παρά να ψάχνει στα τυφλά σε μεγάλα κομμάτια κώδικα για το που είναι το λάθος.
Με την χρήση των πιο πάνω προτύπων μπορούμε να δημιουργήσουμε ένα λογισμικό το οποίο θα έχει ως σκοπό τον έλεγχο λογισμικού με την χρήση της JML για εύκολη και γρήγορη ανίχνευση των προγραμματιστικών λαθών που παρατηρούνται. Όπως έχουμε δει η JML μπορεί να βοηθήσει στην γρήγορη αποσφαλμάτωση του κώδικα σε οποιοδήποτε στάδιο της ανάπτυξης μίας εφαρμογής. Η ύπαρξη διαφόρων εργαλείων για την υποστήριξη της JML μας δίνει την δυνατότητα να επιλέγουμε ορισμένες ελλείπεις λειτουργίες για να τις προσθέσουμε. Μία λειτουργία που δεν υφίσταται είναι η αυτοματοποίηση της ανίχνευσης των προγραμματιστικών λαθών που είναι σημαντική για την ανίχνευση και την διόρθωση των σφαλμάτων στον κώδικα, και μπορεί να προσφέρει εύκολη και αποδοτική χρήση της JML τόσο από πεπειραμένους όσο και από αρχάριους χρήστες της συγκεκριμένης γλώσσας προδιαγραφών.

5 JML Templates Generator Plugin

5.1 Εισαγωγή

Παρ' όλη την ύπαρξη πολλών εργαλείων για την υποστήριξη της JML και των πολλών δυνατοτήτων που προσφέρει δεν έχουν παρουσιαστεί αρκετά εργαλεία τα οποία να είναι προσιτά στο ευρύ προγραμματιστικό δυναμικό της Java. Εργαλεία ενσωματωμένα στην προγραμματιστική πλατφόρμα του Eclipse μόνο ένα το οποίο προσφέρει μόνο τις βασικές λειτουργίες και την επεξεργασία κώδικα με προδιαγραφές JML. Λόγω του περιορισμού αυτού πολλοί χρήστες δεν χρησιμοποιούν την JML λόγω των δυσκολιών για την κατανόηση της γλώσσας και της έλλειψης όχι απαραίτητα τεκμηρίωσης αλλά γενικής βοήθειας ακόμα και μέσω του διαδικτύου.

Στην συγκεκριμένη έρευνα αυτό που θέλω να προσφέρω είναι η προοπτική δημιουργίας τόσο plugins όσο και περισσοτέρων εφαρμογών που να προσφέρουν δυνατότητες εύκολης συγγραφής κώδικα προδιαγραφών σε JML. Η προηγούμενη ενότητα που αφορούσε τα κοινά προγραμματιστικά λάθη που γίνονται στην γλώσσα Java έρχεται να αποδείξει ότι συγκεκριμένες προδιαγραφές μπορούν να μπουν σε πρότυπα και να αυτοματοποιηθούν σε μεγάλο βαθμό έτσι ώστε ένας χρήστης της γλώσσας Java να μπορεί να τα ενσωματώνει στον κώδικα του εύκολα και γρήγορα με μόνη αλλαγή τα ονόματα των μεταβλητών και ίσως κάποιες επιπλέον υποκειμενικές προδιαγραφές. Με αυτό τον τρόπο το έργο του όσο αφορά τον αυστηρό καθορισμό των προδιαγραφών για περεταίρω έλεγχο του λογισμικού μπορεί να γίνει πιο εύκολο όπως επίσης και η τεκμηρίωση του κώδικα για αποδοτικότερη και καλύτερη κατανόηση και συντήρηση.

Πιο κάτω παρουσιάζεται η ανάπτυξη ενός plugin στην πλατφόρμα ανάπτυξης λογισμικού Eclipse απ' όπου ένα προγραμματιστής θα μπορεί κατά την διαδικασία συγγραφής κώδικα Java με δεξί click στον Editor να του παρέχεται η δυνατότητα αυτόματης εισαγωγής κάποιου συγκεκριμένου προτύπου σε JML στο σημείο στο οποίο έχει επιλέξει. Επιλέχθηκε το δεξί κλικ λόγω του ότι είναι γρήγορο και εύκολο με την δυνατότητα αναγραφής ολόκληρου του ονόματος του προτύπου σε αντίθεση με την επιλογή του επιπλέον toolbar ή view που προκαλούν ασάφειες και καταναλώνουν μεγάλο οπτικό χώρο στο παράθυρο της πλατφόρμας Eclipse.

5.2 Εισαγωγή στο Eclipse και στα Plugins

5.2.1 Eclipse Software Development Platform

Το Eclipse είναι μία πλατφόρμα ανάπτυξης λογισμικών η οποία προσφέρει την δυνατότητα χρήσης πολλών γλωσσών, και το πιο κύριο χαρακτηριστικό του είναι το ότι συμπεριλαμβάνει συστήματα **Integrated Development Environment (IDE)** και **Plugin** για επέκταση της υφιστάμενης πλατφόρμας από τους ίδιους τους χρήστες. Το Eclipse είναι γραμμένο σε Java χρησιμοποιείται για την ανάπτυξη εφαρμογών τόσο σε Java όσο και σε άλλες γλώσσες, με την βοήθεια των plugins, όπως C, C++, Cobol, Python, Perl, PHP και άλλες. Έχει εκδοθεί σύμφωνα με τις παραμέτρους του Eclipse Public Licence που έχει ορίσει το Eclipse Foundation, δημιουργήθηκε από το Free Software Community, είναι δωρεάν και προσφέρεται ως εφαρμογή ανοικτού κώδικα [28].

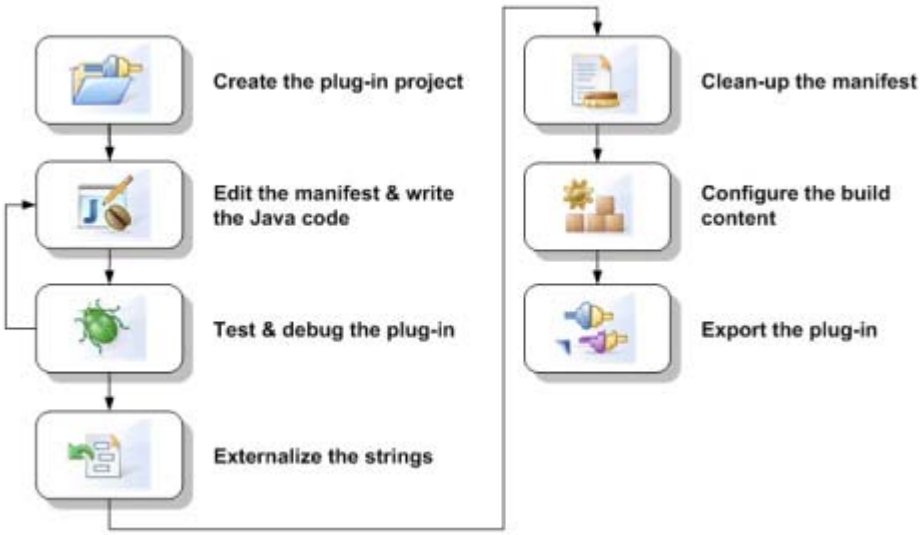
5.2.2 Plugins

Τα Plugins [27] αποτελούν προγράμματα τα οποία αλληλεπιδρούν με ένα κεντρικό λογισμικό για να προσφέρουν συνήθως συγκεκριμένες επιπλέον λειτουργίες κατά παραγγελία. Συνήθως τα λογισμικά υποστηρίζουν αλληλεπίδραση με plugins για διάφορους λόγους όπως, για να επιτρέπουν σε τρίτους προγραμματιστές να προσθέτουν περισσότερες λειτουργίες σαν επέκταση στο ίδιο το λογισμικό, τα επεκτείνουν υπάρχουσες λειτουργίες, για να μειώσουν το αρχικό μέγεθος της εφαρμογής και αποσυνδέσουν τυχών κώδικα του λογισμικού που προκαλεί ασυμβατότητα με διάφορα άλλα κομμάτια του.

Στο Eclipse τα πράγματα είναι λίγο διαφορετικά. Ένα plugin στο Eclipse συνδέεται με ένα μεγάλο αριθμό από άλλα plugins και έτσι δημιουργείται η εφαρμογή που τρέχει. Η καλύτερη αναλογία για τον ορισμό ενός plugin είναι το αντικείμενο του αντικειμενοστραφούς προγραμματισμού. Ένα plugin, όπως ένα αντικείμενο, ενθυλακώνει την συμπεριφορά ή τα δεδομένα, και τα χρησιμοποιεί για την αλληλεπίδραση του με τα υπόλοιπα plugins.

5.3 Δημιουργία Plugin στο Eclipse

Στο πιο κάτω σχήμα φαίνεται η ροή ανάπτυξης ενός plugin στο Eclipse χρησιμοποιώντας το εργαλείο ανάπτυξης Plugins που προσφέρει η πλατφόρμα [10].



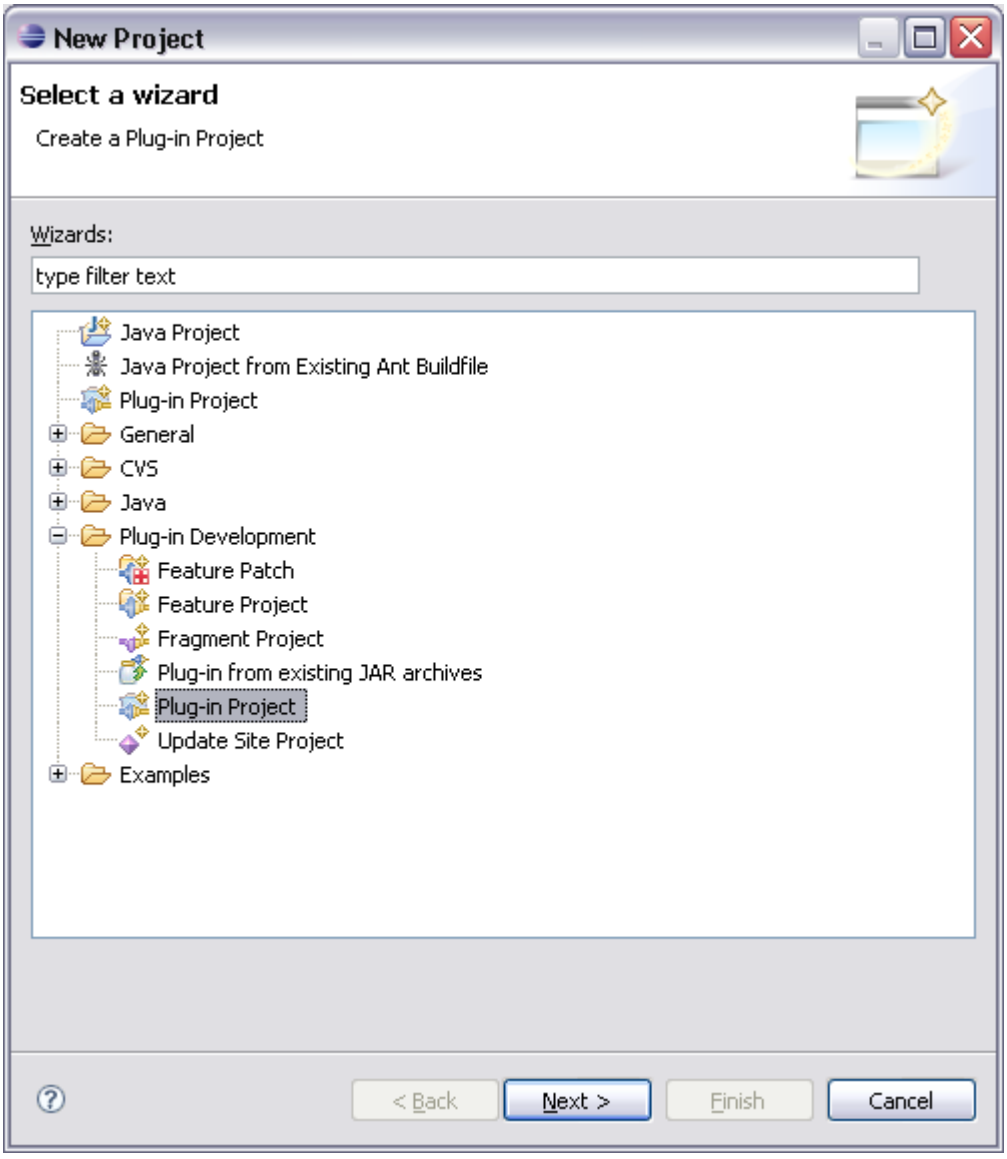
Σχήμα 5.1 : Ροή ανάπτυξης Eclipse Plugin [10]

Αφού κατεβάσουμε και εγκαταστήσουμε το Eclipse με έκδοση την συγκεκριμένο που προσφέρει το Plug-in Development Environment (PDE) από την οικοσελίδα του www.eclipse.org, πρέπει να δημιουργήσουμε ένα νέο project της μορφής Plug-in Project. Στη συνέχεια θα πρέπει να επεξεργαστούμε το μανιφέστο που περιγράφει της συμπεριφορές και τις ιδιότητες του συγκεκριμένου plugin και να τον εκτελέσιμο κώδικα που ορίζει την συμπεριφορά. Για testing και debugging τρέχουμε το πρόγραμμα σαν ένα Eclipse Application στο Debug Perspective και ανάλογα με τα αποτελέσματα επιστρέφουμε στην διόρθωση και συγγραφή του κώδικα και του μανιφέστου. Με την ολοκλήρωση της ανάπτυξης του plugin είναι σημαντικό να εξάγουμε τα στοιχεία που περιγράφουν το συγκεκριμένο plugin, να οργανώσουμε για τελευταία φορά το μανιφέστο, να ορίσουμε ποια από όλα τα δεδομένα θα συμπεριληφθούν στο τελικό για εγκατάσταση plugin και να τα το διαθέσουμε με την μορφή που επιθυμούμε έτοιμο για εισαγωγή στην κύρια εφαρμογή της πλατφόρμας του Eclipse.

5.3.1 Δημιουργία του Plugin Project

Από το μενού επιλέγουμε File → New → Project

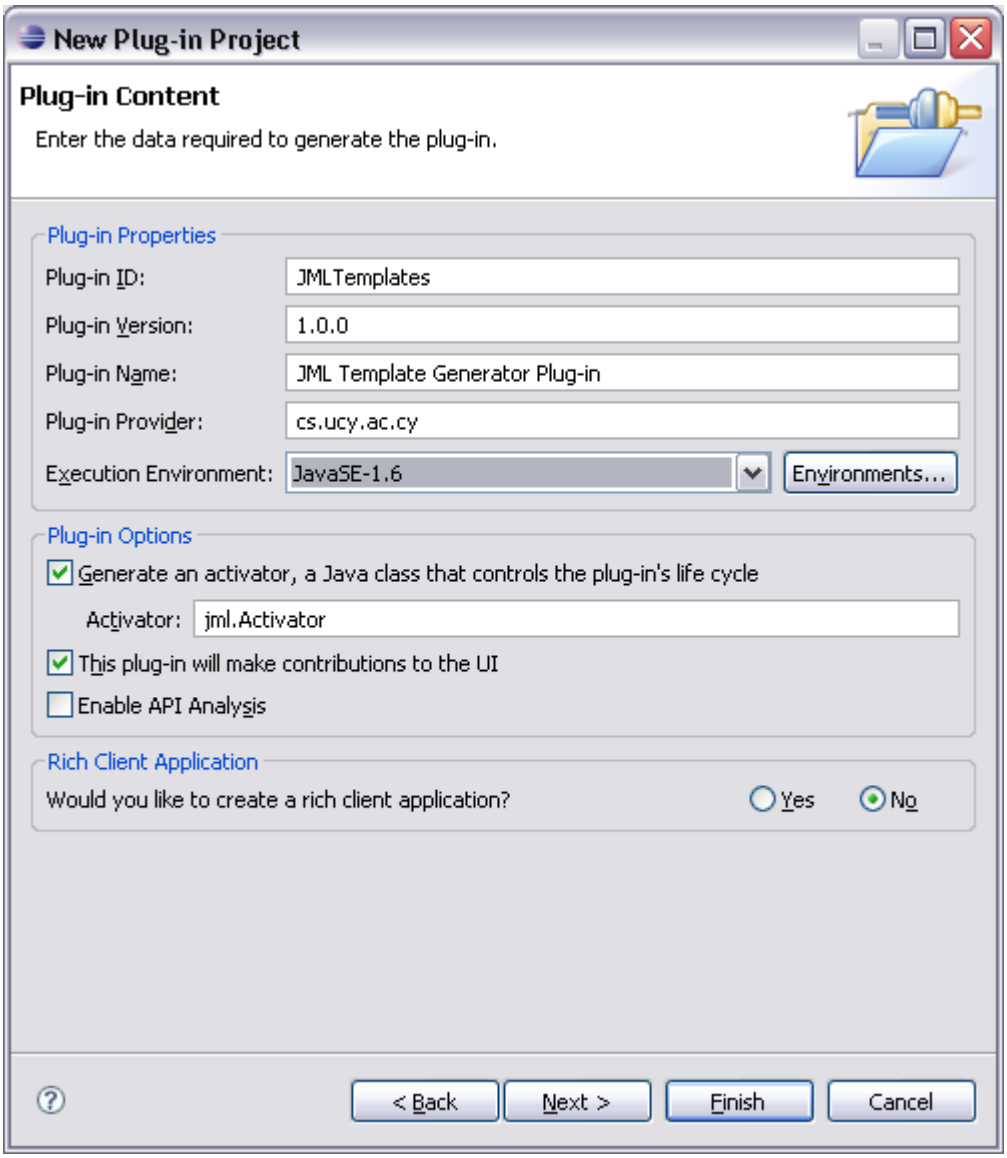
Στην συνέχεια επιλέγουμε το Plug-in Project όπως φαίνεται πιο κάτω από τον σύνδεσμο Plug-in Development.



Σχήμα 5.2 : Δημιουργία νέου Eclipse Plugin Project

Στην συνέχεια όπως όλα τα Project στο Eclipse πρέπει να οριστεί όνομα του project, τα ονόματα των φακέλων αποθήκευσης κώδικα και εξόδων του προγράμματος και ο αριθμός της έκδοσης για τον οποίο προορίζεται η χρήση του συγκεκριμένου προγράμματος.

Μετά ορίζονται οι γενικές πληροφορίες του plugin όπως φαίνεται πιο κάτω και η μέθοδος ενεργοποίησης του plugin. Για execution environment ορίζεται η έκδοση του Java Virtual Machine που θα υποστηρίζει την λειτουργία του συγκεκριμένου plugin.

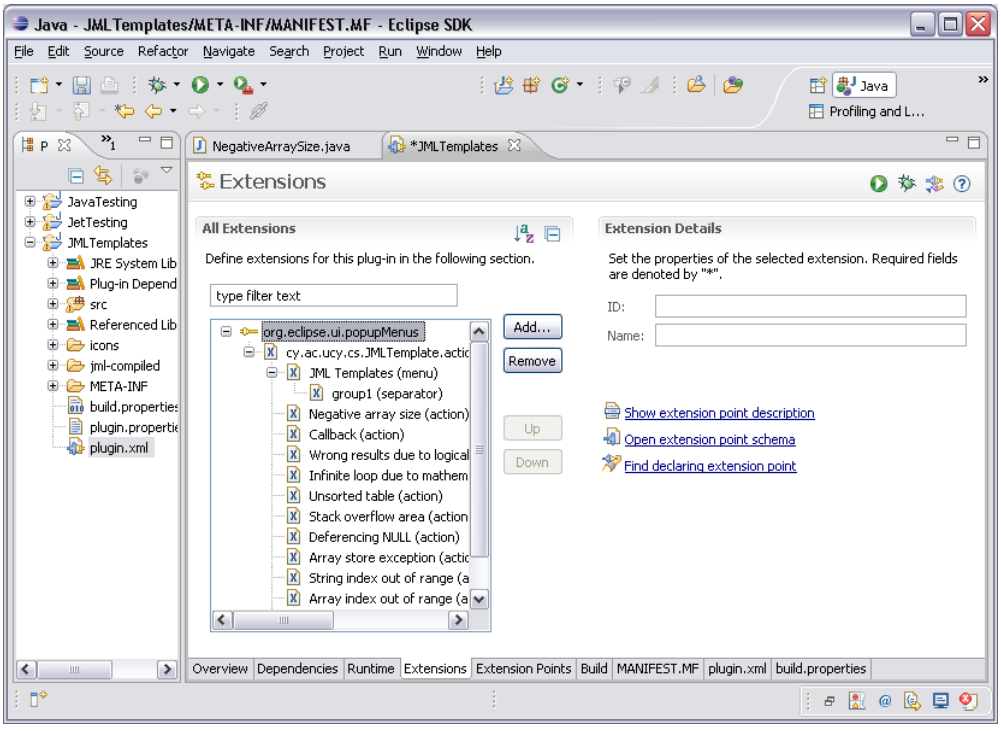


Σχήμα 5.3 : Χαρακτηριστικά νέου Eclipse Plugin Project

5.3.2 Επεξεργασία Μανιφέστου του Plugin

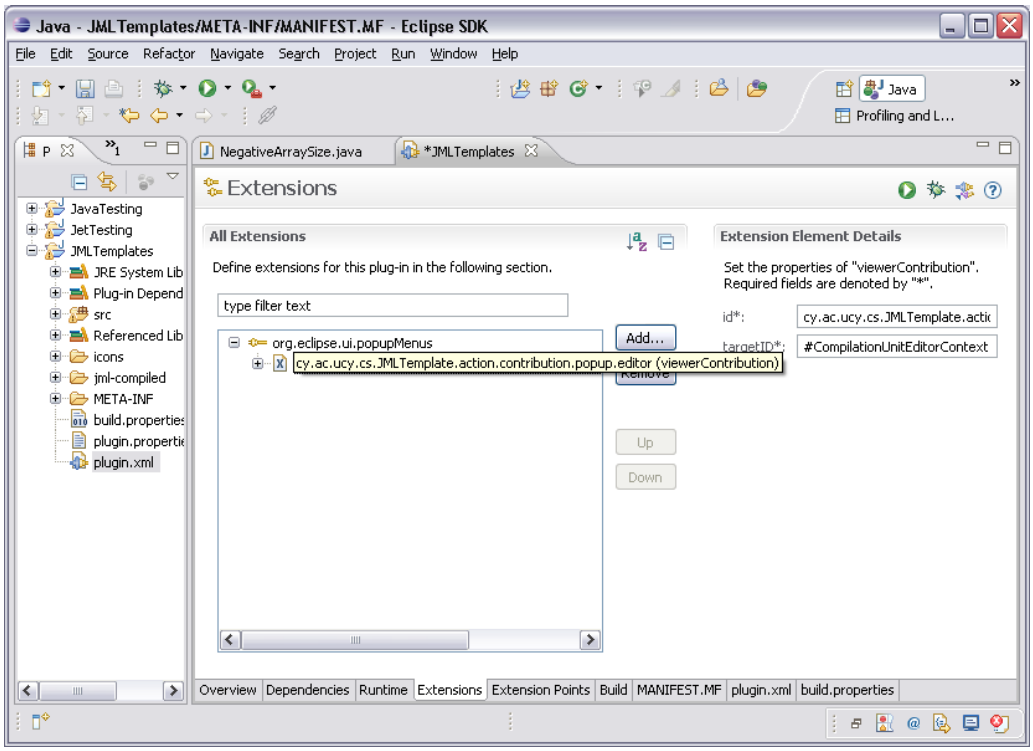
Με την ολοκλήρωση της αρχικής δημιουργίας του plugin project έχουμε την δυνατότητα να το επεξεργαστούμε όπως όλα τα projects στο Eclipse με την διαφορά το συγκεκριμένο για να λειτουργεί ως ένα plugin θα πρέπει να επεξεργαστούμε το μανιφέστο μεταπληροφοριών (MANIFEST.MF) και όλα τα άλλα αρχεία που περιγράφουν το Plugin.

Το αρχείο plugin.xml περιγράφει σε κώδικα xml τις επεκτάσεις του plugin ως προς το υπάρχον λογισμικό. Το PDE του Eclipse μας προσφέρει την επιλογή επεξεργασίας του plugin.xml μέσω ενός wizard από το tab *extensions* αλλά μπορούμε να το γράψουμε και hardcoded μέσα στο αρχείο plugin.xml. Και τα δύο αλληλεπιδρούν δυναμικά ενημερώνοντας τα δεδομένα του από όπου και δουλεύει ο προγραμματιστής. Λόγω του ότι η εισαγωγή των δεδομένων γίνεται μέσω της ίδιας της πλατφόρμας πρέπει να γίνει αρχικά μία επέκταση του plugin που διαχειρίζεται το right click menu του Java Editor. Το συγκεκριμένο plugin ονομάζεται όπως φαίνεται και πιο κάτω *org.eclipse.ui.popupMenus.* Αυτό το plugin είναι υπεύθυνο για τις λειτουργίες που αναγράφονται και εκτελούνται κατά τα right clicks menus σε όλο το Eclipse, και μπορούμε να το προσθέσουμε στη λίστα των επεκτάσεων από το κουμπί Add.



Σχήμα 5.4 : Πρόσθεση επεκτάσεων στο Plugin

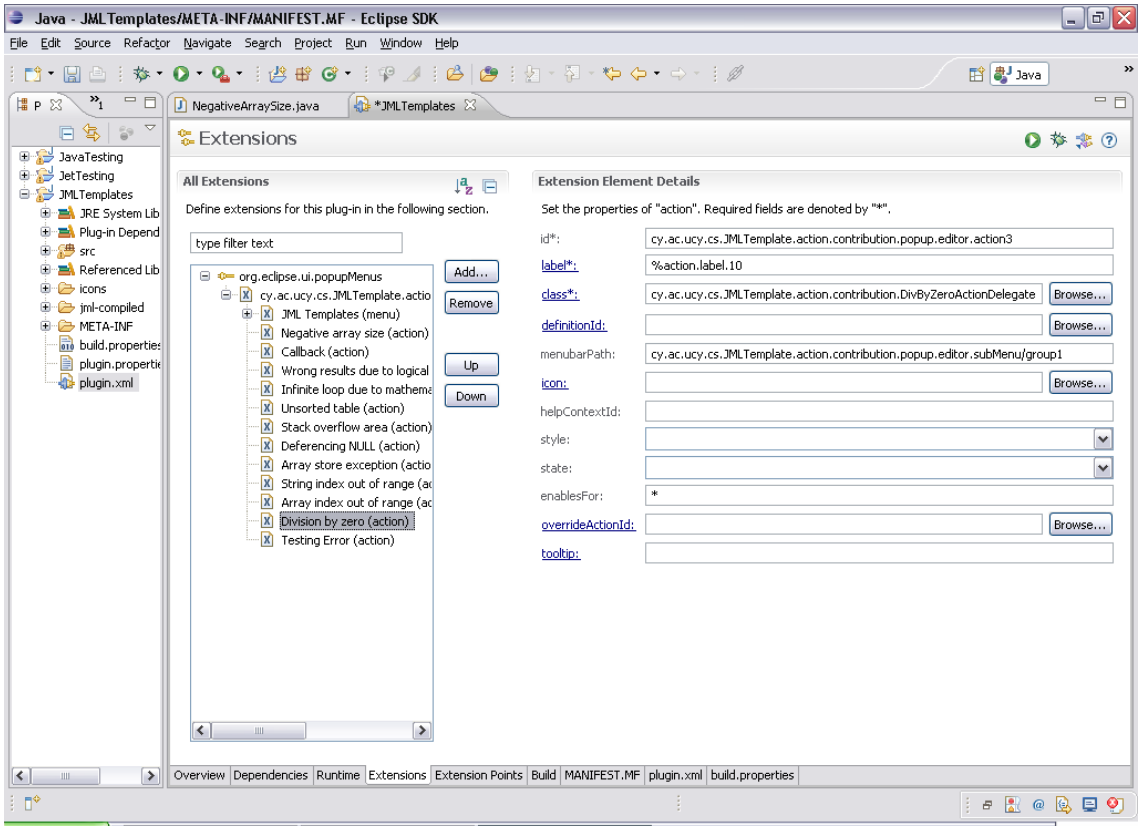
Αφού επιλέξουμε το plugin για επέκταση θα πρέπει να ορίσουμε για πού προορίζεται το συγκεκριμένο extension. Εάν προοριζόταν για όλα τα right click menus οπουδήποτε τότε θα πρέπει να προσθέταμε ένα object contribution το οποίο θα περιέγραφε την όψη του extension. Αντί αυτού προσθέτουμε ένα viewer contribution και ορίζουμε ότι προορίζεται για το *#CompilationUnitEditorContext* το οποίο είναι το right click menu του Java Editor σύμφωνα με το εγχειρίδιο χρήσης του Eclipse 3.0 [29], όπως φαίνεται πιο κάτω.



Σχήμα 5.5 : Καθορισμός χαρακτηριστικών επέκτασης

Αφού ορίσαμε το contribution του plugin πρέπει να ορίσουμε την όψη που θα έχει το υπομενού στο right click menu στον editor. Αφού προσθέσουμε ένα μενού με το όνομα JML Templates στον χώρο additions του default menu της κλάσης Sub, για κάθε λάθος που αναφέραμε σε προηγούμενη ενότητα πρέπει να ορίσουμε ένα Action το οποίο θα καθορίζει το πώς και το που θα εμφανίζεται στο μενού αλλά και ποια κλάση στον πηγαίο κώδικα του plugin θα είναι ο εκπρόσωπος της λειτουργίας την οποία εκτελεί κατά την επιλογή του κάθε action από το μενού. Ακόμα στις επιλογές υπάρχουν και άλλες επιλογές για τα χαρακτηριστικά του κάθε action όπως τι tooltip θα εμφανίζεται, την αρχική κατάσταση, το στυλ που ακολουθεί κ.α.. Όπως φαίνεται και στην πιο κάτω

εικόνα έχουν οριστεί τα λάθη που προαναφέραμε και το extension του μενού έχει ολοκληρωθεί.

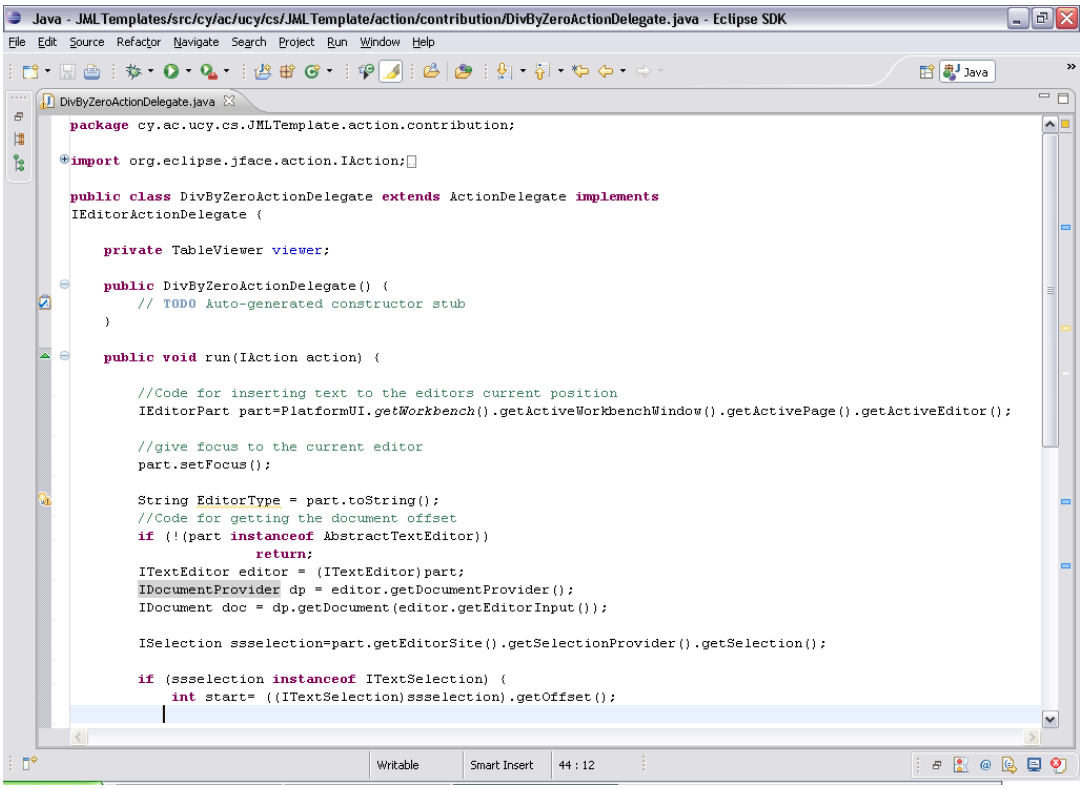


Σχήμα 5.6 : Καθορισμός ενεργειών επέκτασης

5.3.3 Επεξεργασία πηγαίου κώδικα Plugin

Με την ολοκλήρωση της επέκτασης του μενού και με τον ορισμό των κλάσεων αντιπροσώπων του κάθε Action έχει δημιουργηθεί αυτόματα στον φάκελο του πηγαίου κώδικα ένα πακέτο που περιέχει τα αρχεία όλων των κλάσεων που έχουμε δηλώσει. Σε αυτές τις κλάσεις υπάρχουν όλες οι μέθοδοι που καλούνται κατά την εκτέλεση του action που αντιπροσωπεύει η συγκεκριμένη κλάση. Η πιο σημαντική μέθοδος είναι η run η οποία περιέχει και τον κώδικα εκτέλεσης στην περίπτωση που επιλεγεί ένα action του JML Templates από το μενού.

Όπως φαίνεται και πιο κάτω είναι απαραίτητο να συμπεριλάβουμε διάφορες άλλες κλάσεις και διαπροσωπίες οι οποίες περιέχουν μεθόδους και αντικείμενα ελέγχου του ίδιου του περιβάλλοντος εργασίας κατά το χρόνο εκτέλεσης του Eclipse. Για να μπορέσουμε να έχουμε πρόσβαση στο σημείο όπου γίνεται το right click θα πρέπει να ανιχνεύσουμε τον τρέχον ενεργό editor. Στη συνέχεια δίνοντας το focus στο συγκεκριμένο editor ο δρομέας επιστρέφει στην θέση όπου έχει γίνει το click. Χρησιμοποιώντας της κατάλληλες μεθόδους των κλάσεων διαχείρισης αρχείων και των events στον editor μπορούμε να ανιχνεύσουμε το offset στο οποίο βρίσκεται ο δρομέας

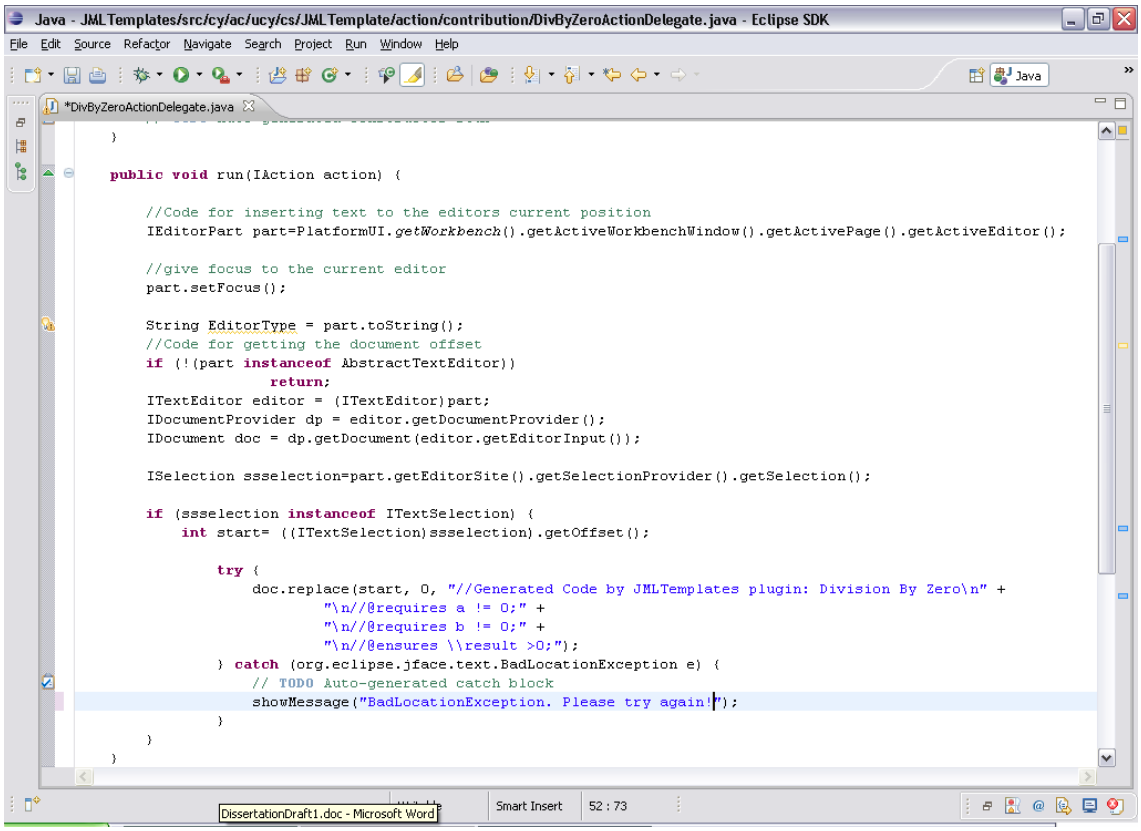


Σχήμα 5.7 : Παράδειγμα κώδικα εισαγωγής προτύπου (1)

την δεδομένη χρονική στιγμή και πιο είναι το τρέχον αρχείο εργασίας στο οποίο δουλεύουμε μέσω του Java Editor.

Όπως βλέπουμε και στην εικόνα του πηγαίου κώδικα πιο κάτω η συνέχεια είναι αρκετά απλή εφόσον γνωρίζουμε τόσο το αρχείο όσο και τη θέση μέσα στο αρχείο στην οποία θα πρέπει να τοποθετήσουμε τον κώδικα του προτύπου σε JML. έτσι το μόνο που έχουμε να κάνουμε είναι να προσθέσουμε στο σημείο αυτό τον κώδικα που θέλουμε με την χρήση της κατάλληλης μεθόδου

Λόγω του ότι τα πρότυπα πολλές φορές περιέχουν assertions και άλλες εκφράσεις εντός της μεθόδου κατά την αυτόματη εισαγωγή αυτών των προτύπων δεν θα γίνεται εισαγωγή αυτών των εκφράσεων παρά μόνο τα precondition και postconditions.



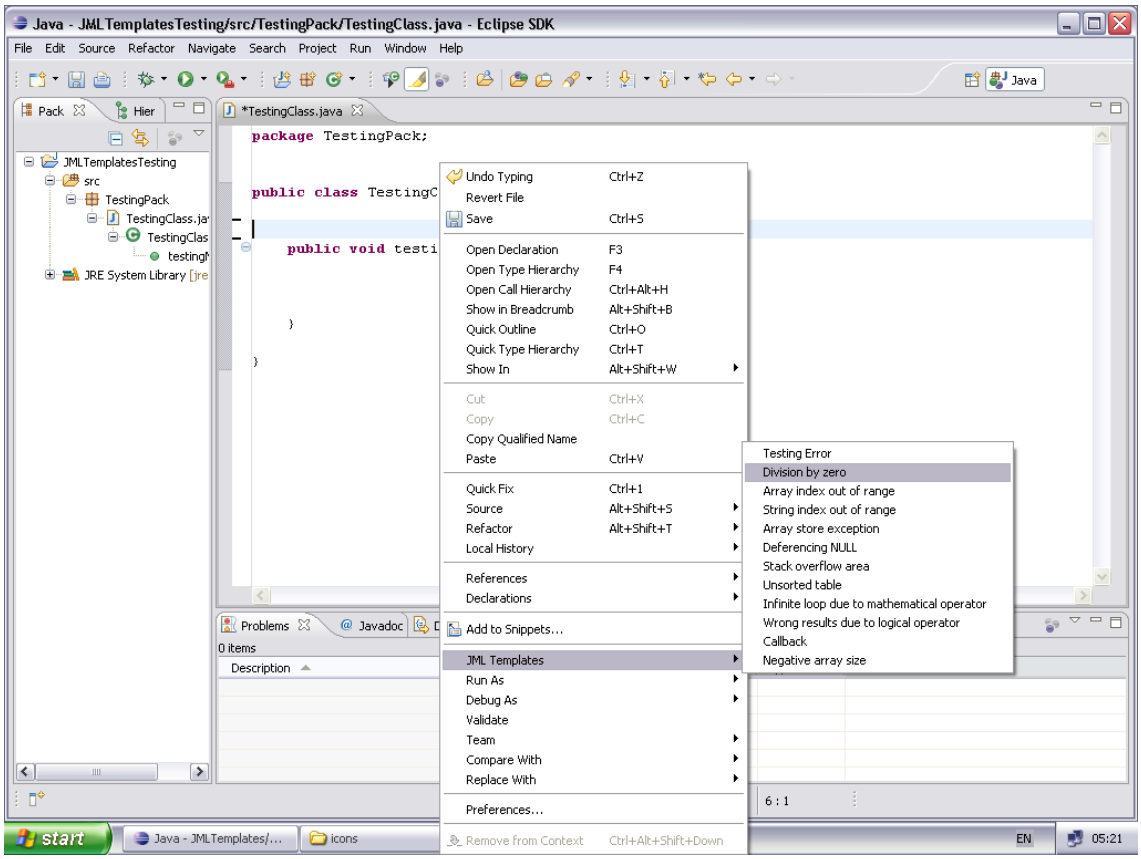
Σχήμα 5.8 : Παράδειγμα κώδικα εισαγωγής προτύπου (2)

Η ίδια διαδικασία γίνεται για όλες της μεθόδους *run* όλων των κλάσεων αντιπροσώπων που έχουν οριστεί για όλα τα διαθέσιμα πρότυπα προγραμματιστικών λαθών.

5.3.4 Δοκιμή και αποσφαλμάτωση του plugin

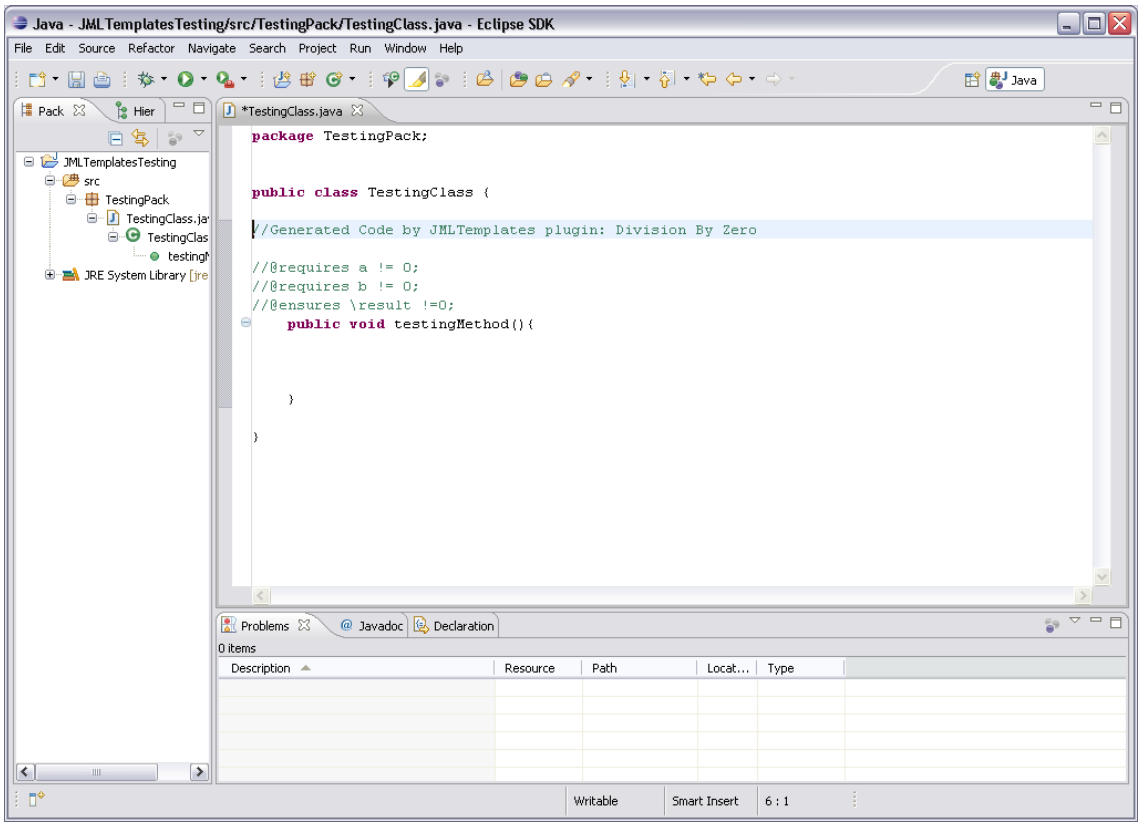
Η δοκιμή και αποσφαλμάτωση του plugin δεν γίνεται με τον ίδιο τρόπο όπως όλα τα προγράμματα java διότι απαιτεί την εκτέλεση ολόκληρη ξανά της εφαρμογής με συμπερίληψη του plugin στα plugins με οποία γίνεται initialized. Το eclipse μας δίνει πλέον την δυνατότητα να τρέχουμε πολλαπλές διεργασίες της ίδιας της πλατφόρμας με αποτέλεσμα κάνοντας το run→as Eclipse Application ολόκληρη η εφαρμογή να τρέχει σαν δεύτερο instance με ενσωματωμένο το ελεγχόμενο plugin. Ακόμα υπάρχει και η δυνατότητα να κάνουμε και ταυτόχρονο debugging με την επιλογή debug → as Eclipse Application.

Όπως βλέπουμε και από την πιο κάτω εικόνα ένα νέο υπομενού έχει εμφανιστεί το οποίο με την σειρά του περιέχει τις ονομασίες των λαθών για τις οποίες έχουν οριστεί τα πρότυπα. Το right click έγινε στην γραμμή πάνω από την μέθοδο στην οποία θέλουμε να προσθέσουμε τις προδιαγραφές για την ανίχνευση του μαθηματικού λάθους διαίρεσης με το 0. Στη συνέχεια με την πλοήγηση μέσα από το μενού που έχουμε δημιουργήσει μέσω του plugin φτάνουμε στο επιθυμητό template που θέλουμε να εισάγουμε.



Σχήμα 5.9 : Έλεγχος λειτουργίας plugin

Με την επιλογή του προτύπου διαίρεσης με το 0 τοποθετείται ο κώδικας ο οποίος έχει προταθεί πιο σε προηγούμενη ενότητα της έρευνας με την κατάλληλη σήμανση JML έτσι ώστε να είναι έτοιμη η ανάγνωση των προδιαγραφών από το JML parser χωρίς καμία άλλη αλλαγή .



Σχήμα 5.10 : Έλεγχος ορθής εισαγωγής προτύπου από το plugin

Το πρότυπο έχει εισαχθεί με την σωστή μορφή και η επέκταση λειτουργεί όπως αναμενόταν έτσι οδηγούμαστε στο συμπέρασμα ότι το συγκεκριμένο plugin είναι έτοιμο για εξαγωγή και χρήση στο κύριο λογισμικό του Eclipse.

5.3.5 Εξαγωγή του plugin

Για την εξαγωγή του Plugin θα πρέπει να ακολουθήσουμε τα βήματα για την σωστή εξαγωγή του έτσι ώστε να μην υπάρχουν ασάφειες όσον αφορά την προέλευση του, το περιεχόμενο και τον τρόπο λειτουργίας του.

5.3.5.1 Εξαγωγή στοιχείων και παραμέτρων

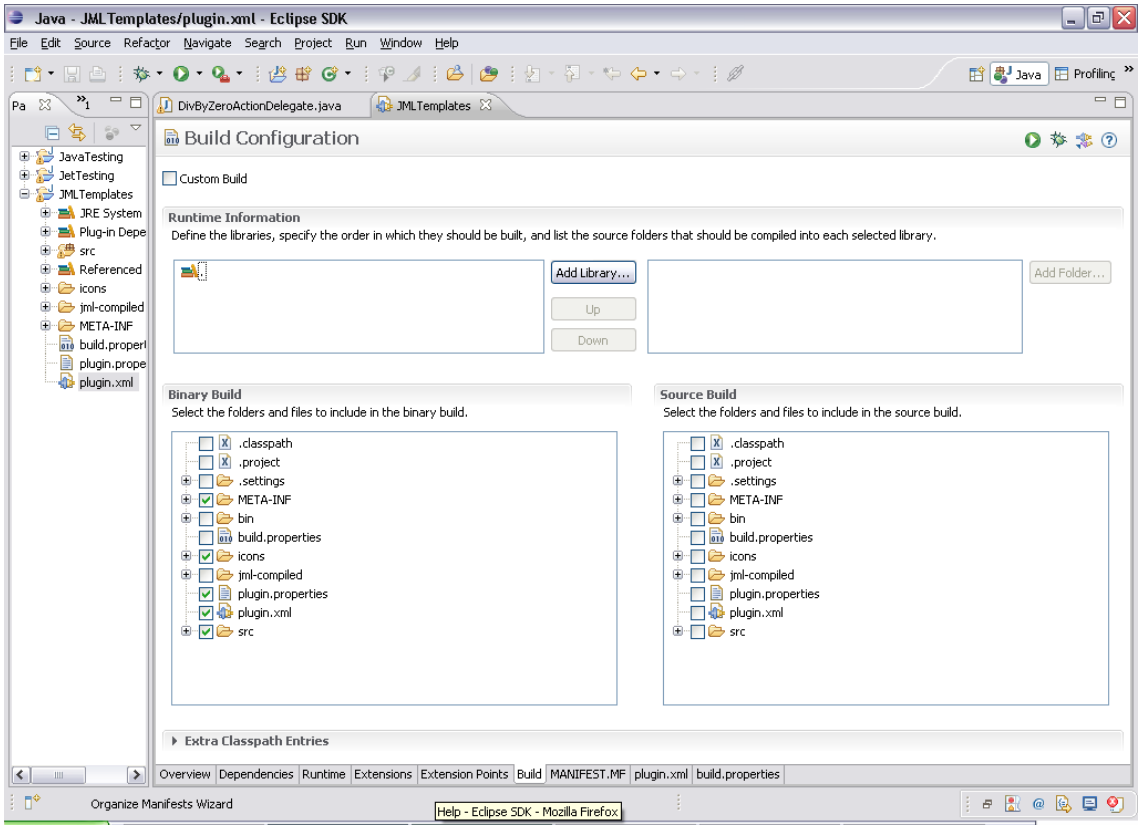
Η λειτουργία εξαγωγής των στοιχείων και των παραμέτρων **(Externalize the Strings)** γίνεται για την εξαγωγή όλων των πληροφοριών του plugin έτσι για να δίνεται η δυνατότητα μετάφρασης αυτών των πληροφοριών σε οποιαδήποτε υποστηριζόμενη γλώσσα του Eclipse χωρίς την ανάγκη επεξεργασίας τους μέσω του Eclipse και δημιουργίας ξανά του plugin. Το Eclipse προσφέρει διάφορες λειτουργίας για επεξεργασία αυτών των πληροφοριών όπως και οδηγό αυτόματης εξαγωγής τους.

5.3.5.2 Οργάνωση του Μανιφέστου

Η οργάνωση του μανιφέστου (**Organizing Manifest Files)** είναι απαραίτητη πριν από την τελική εξαγωγή του plugin διότι με την τελευταία του επεξεργασία, επιβεβαιώνουμε ότι όλες οι πληροφορίες που είναι καταχωρημένες στο μανιφέστο είναι ενημερωμένες. Και σε αυτή την περίπτωση το Eclipse μας παρέχει ένα οδηγό οργάνωσης και καθαρισμού του μανιφέστου είτε από περιττές πληροφορίες είτε ελλιπείς εξαρτήσεις των δεδομένων και των πακέτων που υλοποιούν το plugin.

5.3.5.3 Διαμόρφωση της δομής του plugin

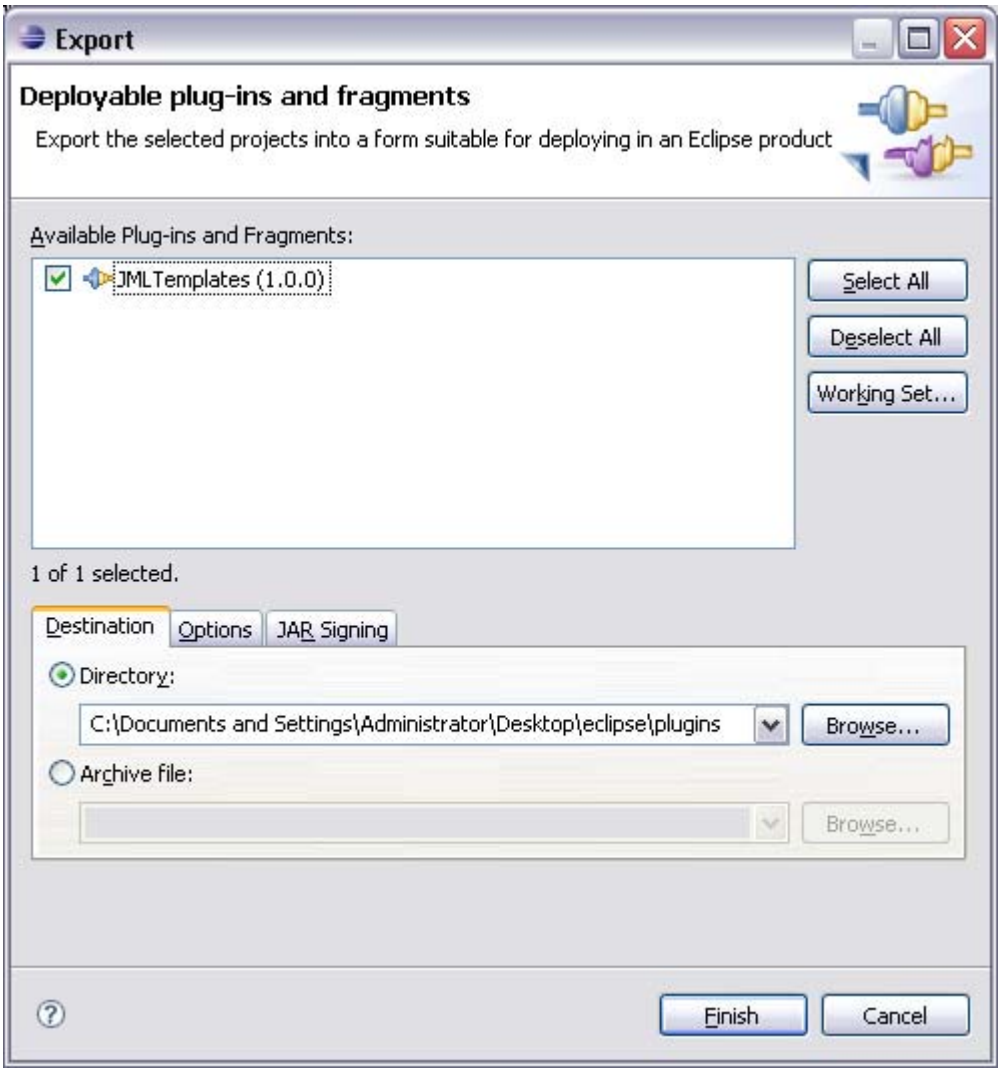
Πριν από το τελευταίο βήμα εξαγωγής του plugin πρέπει να καθορίσουμε την δομή του plugin σύμφωνα με τις ιδιότητες του plugin και τις πρόσβαση στις λεπτομέρειες που θέλουμε να έχουν οι χρήστες που θα το αξιοποιήσουν. Όπως βλέπουμε και πιο κάτω στην εικόνα έχουμε την δυνατότητα να κτίσουμε το plugin σε binary και source. Το binary build μπορούμε να επιλέξουμε ποια αρχεία θα συμπεριληφθούν μέσα στο πακέτο του plugin που θα δημιουργηθεί. Το Source build υπάρχει έτσι ώστε να μπορούμε να εξάγουμε αυτά τα αρχεία για να τα συμπεριλάβουμε σε άλλα plugins ή άλλα προγράμματα παρά στο binary plugin που έχουμε δημιουργήσει.



Σχήμα 5.11 : Διαμόρφωση δομής του plugin

5.3.5.4 Εξαγωγή του εφαρμόσιμου plugin

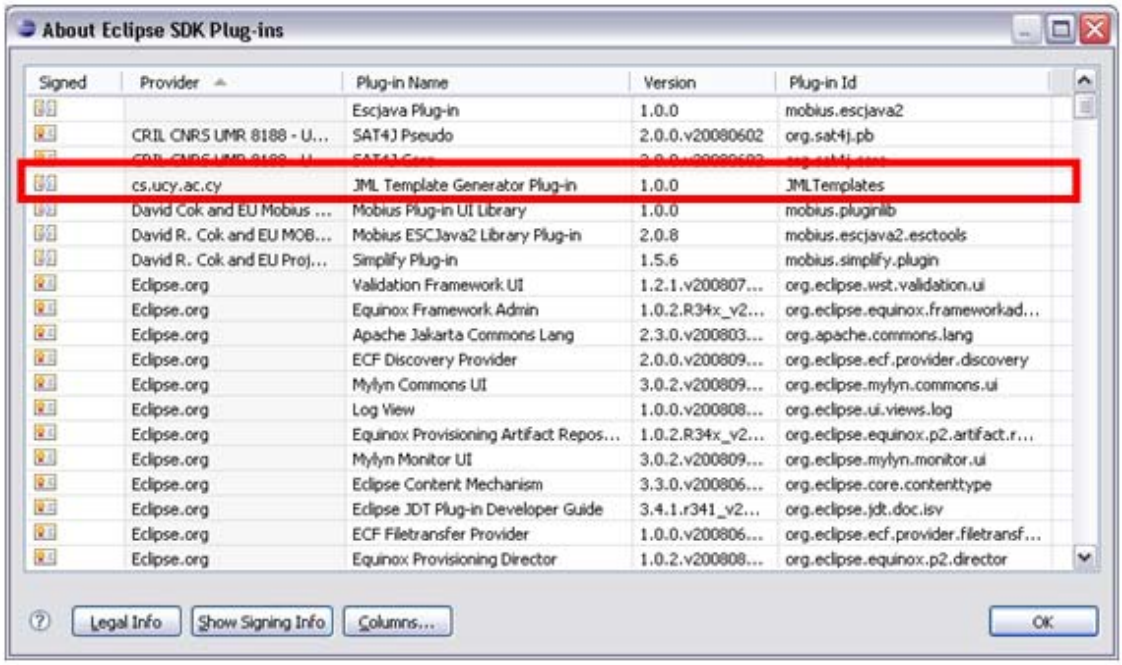
Το τελικό βήμα της εξαγωγής του εφαρμόσιμου plugin που έχουμε δημιουργήσει είναι το κτίσιμο του και η έκδοση του στην μορφή που επιθυμούμε. Όπως βλέπουμε και στην πιο κάτω εικόνα με τον οδηγό εξαγωγής του πακέτου δημιουργούμε ένα jar αρχείο στον επιθυμητό χώρο στο file system.



Σχήμα 5.12 : Εξαγωγή του εφαρμόσιμου plugin

5.3.5.5 Εγκατάσταση του plugin

Για την εγκατάσταση του plugin το μόνο πράγμα που έχουμε να κάνουμε είναι τα μεταφέρουμε το αρχείο .jar που έχουμε δημιουργήσει πιο πάνω στον κατάλογο plugins του καταλόγου εγκατάστασης του Eclipse. Κατά την εκκίνηση του Eclipse φορτώνονται όλα τα plugins που είναι αποθηκευμένα στο συγκεκριμένο κατάλογο και συμπεριλαμβάνονται στα λειτουργήσιμα της γενικής εφαρμογής του Eclipse. Για να βεβαιωθούμε ότι το Eclipse έχει αναγνωρίσει το plugin που έχουμε εγκαταστήσει μπορούμε να πάμε στην λίστα των εγκατεστημένων plugins που βρίσκεται στο Help→About→Plug-in Details. Στην περίπτωση του JML Templates Plugin μπορούμε να δούμε και στην εικόνα της λίστας των Plugins, την ύπαρξη του και λεπτομέρειες για την πιστοποίηση και το όνομα του δημιουργού του κάθε plugin, το όνομα του Plugin, την τρέχουσα εγκατεστημένη έκδοση και την ταυτότητα αναγνώρισης του.



Σχήμα 5.13 : Εγκατάσταση του plugin στην κύρια πλατφόρμα Eclipse

Αφού έχουμε βεβαιωθεί για την σωστή του εγκατάσταση μπορούμε να το χρησιμοποιήσουμε ως ένα ακόμα εργαλείο στην πλατφόρμα ανάπτυξης λογισμικού Eclipse.

5.4 Συμπεράσματα

Ο αυτοματισμός της παραγωγής κώδικα είναι ένα από τα πιο σημαντικά εργαλεία στην Τεχνολογία λογισμικού. Παρατηρώντας όλα τα πλεονεκτήματα που προσφέρει η JML μεγάλο μέρος του κώδικα της μπορεί να αυτοματοποιηθεί για να εισάγεται εύκολα. Αυτό θα προσφέρει τη δυνατότητα στους χρήστες να την χρησιμοποιούν περισσότερο και το πιο σημαντικό δεν θα περιορίζονται στην εισαγωγή των προδιαγραφών στο τέλος της υλοποίησης αλλά θα μπορούν με απλές διαδικασίες να εισάγουν προδιαγραφές κατά την διάρκεια της υλοποίησης προσφέροντας έτσι καλύτερη και αυστηρή τεκμηρίωση της συμπεριφοράς του κώδικα χωρίς την παρουσία ασαφειών και έλλειψης κατανόησης από τους προγραμματιστές που δεν έχουν λάβει μέρος στη σχεδίαση.

Μελετώντας τις δυνατότητες που προσφέρει το Eclipse και το plugin που έχουμε δημιουργήσει μπορούμε να κατανοήσουμε ότι με την ανάπτυξη εφαρμογών που εξυπηρετούν τις ανάγκες κάθε χρήστη η συγγραφή και αποσφαλμάτωση του κώδικα γίνεται πιο εύκολη και αποδοτική. Με το πιο πάνω plugin έχει δημιουργηθεί μία βάση για έρευνα γύρω από τον αυτοματισμό των προδιαγραφών JML και την δημιουργία ενός framework με εφαρμογές και λειτουργίες που αποσκοπούν στον έλεγχο λογισμικού με αυτοματοποιημένες μεθόδους ελέγχου έχοντας ως βασικό άξονα την γλώσσα προδιαγραφών JML.

Τέλος παρατηρώντας την χρήση του plugin που έχω αναπτύξει μπορούμε εύκολα να συμπεράνουμε ότι με την χρήση του γίνεται πιο εύκολη η εισαγωγή πολλαπλών προδιαγραφών για την παράλληλη ανίχνευση περισσοτέρων από ένα προγραμματιστικών λαθών σε κάθε ενότητα. Ακόμα με το plugin μπορεί να γίνει εύκολα εισαγωγή νέων προτύπων λαθών ή και ακόμα να επεκταθεί ακόμα περισσότερο προσθέτοντας του λειτουργίες οι οποίες μπορούν να συμβάλουν με την σειρά τους στην εύκολη συγγραφή προδιαγραφών και την αυτοματοποίηση του ελέγχου τους.

5.5 Μελλοντική εργασία

Σαν μελλοντική εργασία μπορεί να επεκταθεί το υφιστάμενο πλέον plugin έτσι ώστε να προστεθούν πρότυπα για περισσότερα προγραμματιστικά λάθη. Ακόμα η δημιουργία λειτουργίας με την οποία ο χρήστης θα μπορεί να εισάγει εύκολα και αυτόματα δηλώσεις JML είναι ένα έργο το οποίο θα ωφελήσει ακόμα περισσότερο την εύκολη εισαγωγή προδιαγραφών. Το συγκεκριμένο εργαλείο μπορεί να περιλαμβάνει επιλογές

εισαγωγής για assertions, invariants, modifiers και expressions. Θα ήταν καλύτερο εάν οι συγκεκριμένες επιλογές να μην προσθέτονταν στο μενού για το γενικό right click στον Java Editor αλλά κατά το right click πάνω σε κάποια συγκεκριμένη μεταβλητή. Τέλος μία πιθανή μελλοντική εργασία θα ήταν η δημιουργία ενός παρόμοιο plugin το οποίο να συμπεριφερθεί σε μία άλλη πλατφόρμα ανάπτυξης λογισμικού για γλώσσα Java. Ένα τέτοιο περιβάλλον είναι η πλατφόρμα ανάπτυξης λογισμικού της Microsoft MS Visual Studio η οποία προσφέρει την δυνατότητα συγγραφής κώδικα Java, άρα μπορεί να υποστηρίξει έλεγχο των προδιαγραφών του κώδικα όταν είναι γραμμένες σε JML [33][34].

6 Έλεγχος προδιαγραφών JML και Γενετικοί Αλγόριθμοι

6.1 Εισαγωγή

Μελετώντας την συμπεριφορά του ελέγχου των προδιαγραφών κατά το black box testing με το εργαλείο JET [13] παρατηρούμε ότι μία ελεγχόμενη μέθοδος δοκιμάζεται με ένα συγκεκριμένο αριθμό TC τα οποία δημιουργούνται με μία τυχαία μέθοδο βάσει των παραμέτρων της συγκεκριμένης μεθόδου παρατηρούνται δύο σοβαρά προβλήματα.

- Λόγω της τυχαίας δημιουργίας των παραμέτρων μίας μεθόδου παρατηρείται συχνά δημιουργία TC τα οποία δεν έχουν καμία νόημα (meaningless) εφόσον δεν πληρούν τα preconditions που ορίστηκαν για την συγκεκριμένη μέθοδο. Επιπλέον με την αύξηση της πολυπλοκότητας της μεθόδου και ειδικότερα του αριθμού των παραμέτρων, δημιουργούνται όλο και περισσότερα meaningless TC έτσι παρουσιάζεται η ανάγκη παραγωγής περισσοτέρων για να επιτευχθεί ο ικανοποιητικός αριθμός για τον ορθό έλεγχο της μεθόδου.

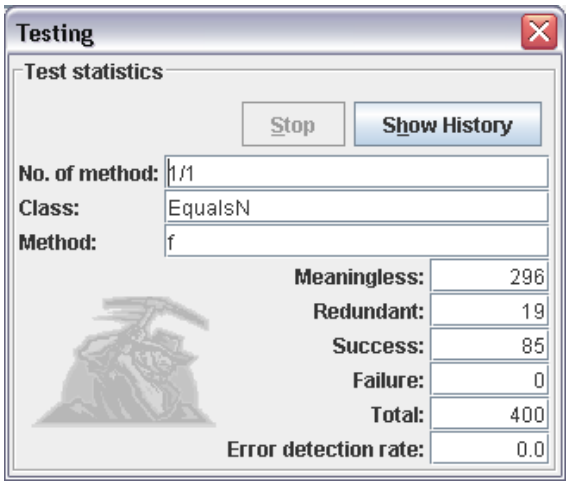
Όπως βλέπουμε και στις εικόνες πιο κάτω στον κώδικα της συνάρτησης f έχουμε μόνο μία παράμετρο θέτοντας όμως και μία προδιαγραφή JML ότι η τιμή πρέπει να είναι μεγαλύτερη του 0 τότε κατά την τυχαία παραγωγή 400 τιμών της παραμέτρου για τα TC δημιουργούνται 296 τιμές που δεν πληρούν τα preconditions, 19 τιμές που έχουν ήδη ελεγχθεί και μόνο 85 που να πληρούν τις προδιαγραφές.

```
public class EqualsN
{
    private /*@spec_public*/ int n;
     //Initialize this search problem.


    // doc comment inherited
    //@ requires j > 0;
    //@ ensures \result == (j > 0);
    public boolean f(int j) {
        return (j > 0);
    }

    //doc comment and specification inherited
    protected String className() {
        return "EqualsN";
    }
}
```

Σχήμα 6.1 : Κώδικας κλάσης ελέγχου(1)



Σχήμα 6.2 : Παράδειγμα ελέγχου JET (1)

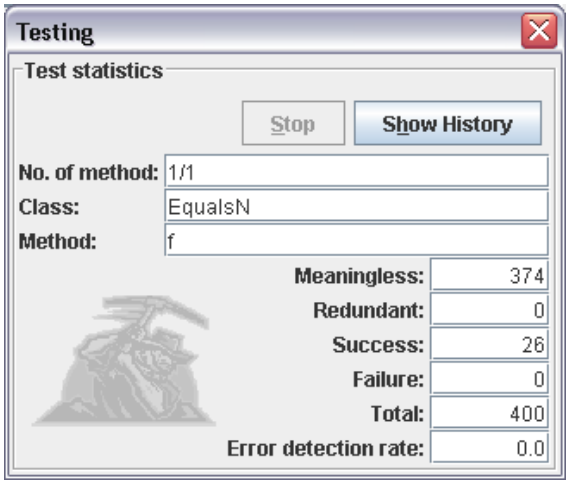
Το πρόβλημα όπως παρατηρούμε πιο κάτω έχει μεγαλώσει σημαντικά. Με την πρόσθεση ακόμα 2 παραμέτρων και θέσπιση προδιαγραφών που τις αφορούν βλέπουμε ότι ο αριθμός των TC τα οποία δεν πληρούν τις προδιαγραφές έχει εκτοξευθεί στις 374 ενώ ο αριθμός των επιτυχημένων έχει πέσει στα 26.

```
public class EqualsN
{
    private /*@spec_public*/ int n;
     //Initialize this search problem.


    // doc comment inherited
    //@ requires j > 0;
    //@ requires f > (j+1);
    //@ requires k > (j-f);
    //@ ensures \result == (j > 0);
    public boolean f(int j,int f,int k) {
        return j > 0;
    }

    //doc comment and specification inherited
    protected String className() {
        return "EqualsN";
    }
```

Σχήμα 6.3 : Κώδικας κλάσης ελέγχου(2)



Σχήμα 6.4 : Παράδειγμα ελέγχου JET (2)

Μία λύση για το πρόβλημα των redundant TC για μία παράμετρο είναι η δημιουργία πισίνας αριθμών (pooling) έτσι ώστε με την επιλογή μίας τιμής να αφαιρείται από την πισίνα για να μην ξαναχρησιμοποιηθεί. Στην περίπτωση των περισσοτέρων παραμέτρων η πιθανότητα να δημιουργήσουμε το ίδιο Test Case είναι πολύ μικρή έτσι δεν χρειάζεται η δημιουργία πισίνας.
Όσο αφορά το πρόβλημα των χωρίς νόημα σεναρίων μπορεί να δημιουργηθεί από πριν το πεδίο τιμών των παραμέτρων σύμφωνα με τις προδιαγραφές JML που έχουν καθοριστεί αλλά και αυτή η λύση αφορά μόνο απλές προδιαγραφές.

- Το δεύτερο πρόβλημα που παρατηρείται στον έλεγχος white box των συγκεκριμένων TC που δημιουργούνται. Λόγω της τυχαίας δημιουργίας TC δεν μπορεί να μας εγγυηθεί ο ελεγκτής ότι όλα τα JML statements που υπάρχουν στον κώδικα έχουν καλυφθεί για να ξέρουμε ότι έστω και για μία τιμή η προδιαγραφή έχει ικανοποιηθεί. Με την χρήση ενός αλγορίθμου coverage μπορούμε να δούμε στο επίπεδο της Java ότι αφορά την κάλυψη του κώδικα και τα υποσύνολα της.

Σε αυτή την ενότητα της έρευνας θα μελετηθεί μία καλύτερη προσέγγιση στην δημιουργία των σεναρίων έτσι ώστε ο έλεγχος των προδιαγραφών του λογισμικού να καλύπτει όλα τις δηλώσεις JML που εμφανίζονται στο πρόγραμμα.

6.2 Προτεινόμενη Προσέγγιση

6.2.1 Κριτήρια Αξιολόγησης νέας προσέγγισης

Ο έλεγχος ενός λογισμικού είναι ακριβός τόσο από πλευρά χρόνου αλλά και από απαιτήσεις δυναμικού. Παρόλο που συνήθως απαιτεί περισσότερο από το μισό του συνολικού κόστους ανάπτυξης του λογισμικού, δεν διεξάγεται σωστά και τα αποτελέσματα πολλές φορές δεν είναι ικανοποιητικά. Ο έλεγχος λογισμικού όμως δεν μπορεί να μην αποτελεί αναπόσπαστο κομμάτι της ανάπτυξης και ολοκλήρωσης ενός λογισμικού, εφόσον είναι ο πιο κύριος τρόπος ελέγχου της εκπλήρωσης των απαιτήσεων του λογισμικού. Σε αυτή την ενότητα θα μελετήσουμε την προσέγγιση του ελέγχου των απαιτήσεων ενός λογισμικού με την χρήση της JML συγκρίνοντας δύο προσεγγίσεις:

- Τον έλεγχο της κάλυψης των δηλώσεων JML με την χρήση τυχαίας παραγωγής σεναρίων δοκιμής.
- Τον έλεγχο της κάλυψης των δηλώσεων JML με την χρήση παραγωγής σεναρίων δοκιμής βασισμένη στους γενετικούς αλγορίθμους.

Αρχικά θα πρέπει να οριστούν τα χαρακτηριστικά της συγκεκριμένης προσέγγισης. Θα μελετηθούν και θα προταθούν, τα εργαλεία τα οποία θα χρησιμοποιηθούν για τον γενετικό αλγόριθμο παραγωγής σεναρίων ελέγχου, και η μέθοδος καταμέτρησης δηλώσεών JML για αξιολόγηση του πιο πάνω αλγορίθμου.

6.2.2 Γενετικός αλγόριθμος

Ο γενετικός αλγόριθμος (ΓΑ)θα λαμβάνει ως είσοδο ένα προκαθορισμένο αριθμό Test Cases τα οποία έχουν είδη ελεγχτεί ότι πληρούν τις προδιαγραφές preconditions της ελεγχόμενης μεθόδου. Τα ποιο πάνω Genes θα αποτελούν και τον Initial Population του αλγορίθμου. Έτσι μπορούμε να ορίσουμε τις συνθήκες εξέλιξης του αλγορίθμου. Τα αρχικά χρωμοσώματα του ΓΑ θα έχουν ως πρωταρχικό στόχο να ελέγξουν το branch coverage που παρέχουν. Στην συνέχεια θα πρέπει να αξιολογούν το κάθε branch όσο αφορά την περιεκτικότητα του σε JML statements. Με αυτό τον τρόπο θα προσδιοριστεί και το Fitness function του αλγορίθμου το οποίο θα αξιολογεί το κάθε TC για να εφαρμοστούν πάνω του οι γενετικές τροποποιήσεις.

6.2.3 Χαρακτηριστικά Γενετικού Αλγορίθμου

6.2.3.1 Genes

Όλες οι παράμετροι εισόδου μίας μεθόδου και όλα τα χαρακτηριστικά που τις διέπουν. Οι παράμετροι αυτοί χαρακτηρίζονται από το όνομα της τοπικής μεταβλητής και τον τύπο της, όπως και το πεδίο τιμών που τις προσδιορίζει.

6.2.3.2 Αρχικός Πληθυσμός

Ένας συγκεκριμένος αριθμός genes οι οποίος δημιουργείται τυχαία και οι τιμές που λαμβάνουν οι παράμετροι ορίζονται από το πεδίο τιμών που ικανοποιεί τα preconditions της ελεγχόμενης μεθόδου.

6.2.3.3 Fitness Function

Η αξιολόγηση των genes λαμβάνοντας υπόψη τόσο την κάλυψη που προσφέρουν τα TC όσο αφορά τα braches της ροής του κώδικα αλλά και την κάλυψη των δηλώσεων JML μέσα στις ενότητες που ορίζονται από τα branches.

6.2.3.4 Γενετικές τροποποιήσεις στα genes

Οι συναρτήσεις crossover και mutation που εφαρμόζονται στα genes μετά την αξιολόγηση τους με το fitness function θα περιορίζονται από τα preconditions της μεθόδου που ελέγχεται έτσι ώστε οι τιμές που λαμβάνονται από τον αλγόριθμο να βρίσκονται εντός των κατάλληλων ορίων.

6.2.3.5 Αξιολόγηση Χρωμοσώματος (Gene Set)

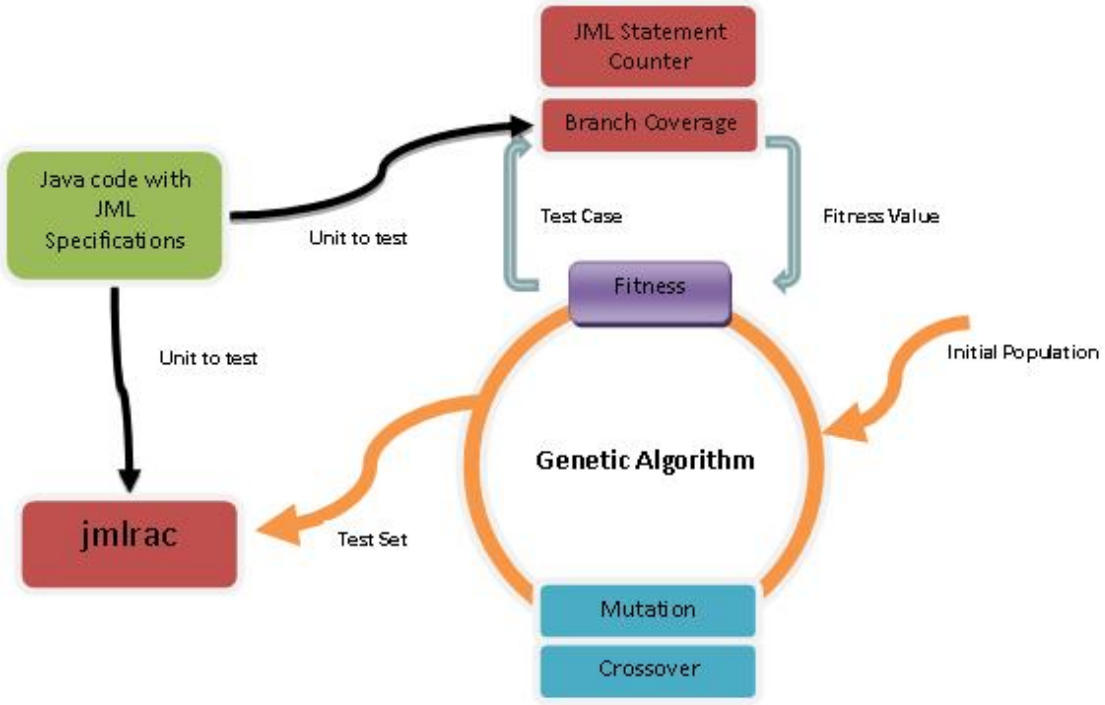
Μετά την ολοκλήρωση του χρωμοσώματος θα αξιολογηθεί η κάλυψη των branches από τον συγκεκριμένο set σε σύγκριση με την παραγωγή τυχαίων Test Cases όπως είδαμε στην προσέγγιση του εργαλείου JET.

6.2.3.6 Συνθήκες τερματισμού παραγωγής των Test Cases

- Με την επίτευξη της μέγιστης δυνατής κάλυψης από τον αλγόριθμο για όλα τα branches, τα οποία περιέχουν προδιαγραφές JML μέσα στις ενότητες που περιγράφουν
- Όταν δημιουργηθεί ένας συγκεκριμένος αριθμός από συνεχόμενα Test Cases που δεν παράγουν καλύτερο Fitness Value από το τρέχον πιο ψηλό.

6.2.4 Ροή Εργασιών Προτεινόμενης προσέγγισης

Στο πιο κάτω σχήμα μπορούμε να δούμε την ροή των εργασιών που διεξάγονται κατά την πιο πάνω προσέγγιση μέχρι να ολοκληρωθεί η διαδικασία αυτόματης παραγωγής Test Cases. Κατ' αυτή την διαδικασία ο ΓΑ παίρνει ως όρισμα ένα αρχικό αριθμό από Test Cases τα οποία κατ' επανάληψη αξιολογεί και τροποποιεί. Η αξιολόγηση γίνεται με ένα εργαλείο που καταμετρά την κάλυψη που πετυχαίνεται όσον αφορά το branch coverage και στην συνέχεια ελέγχεται η ύπαρξη δηλώσεων JML μέσα στην ενότητα που καθορίζει η συγκεκριμένη διακλάδωση. Στην συνέχεια επιστρέφεται στον αλγόριθμο η τιμή αξιολόγησης βάση της οποίας θα διενεργηθούν οι γενετικές τροποποιήσεις στο τρέχον TC και στην συνέχεια θα επαναληφθεί η διαδικασία. Τέλος αφού ολοκληρωθεί ολόκληρη η διαδικασία έχει πλέον δημιουργηθεί ένα Test Set το οποίο μπορούμε να ελέγξουμε τον πηγαίο κώδικα στο σύνολο του με το jmlrac για παραβάσεις των προδιαγραφών JML.



Σχήμα 6.5 : Ροή εργασιών προτεινόμενης προσέγγισης

6.2.5 Διαθέσιμα εργαλεία

Υπάρχουν αρκετά διαθέσιμα εργαλεία τα οποία μπορούν να χρησιμοποιηθούν σε διάφορα σημεία της προσέγγισης μας. Εκτός από τα εργαλεία της JML χρειαζόμαστε ένα εργαλείο το οποίο μπορεί να μας προσφέρει τις ιδιότητες των γενετικών αλγορίθμων αλλά και να υλοποιεί συναρτήσεις ελέγχου για branch coverage για την αξιολόγηση του κάθε TC μέσα στο fitness function. Ένα ολοκληρωμένο εργαλείο είναι το προτεινόμενο από τους Α. Ανδρέου και Α. Σοφοκλέους, που ορίζει ένα framework για δυναμική παραγωγή Test Cases ο οποίος έχει ως κριτήριο εξέλιξης το edge/condition coverage [9]. Ακόμα για τον έλεγχο των JML statements υπάρχουν τα υπάρχοντα εργαλεία της JML που υπάγονται στο jmlunit και μπορούν να προσφέρουν δυνατότητες unit testing με βάσει τις προδιαγραφές JML [2].

6.2.6 Προσαρμογή εργαλείων ATCGS & BPAS

Το Framework που έχει προταθεί από τους Α. Ανδρέου και Α. Σοφοκλέους αποτελείται από δύο βασικά προγράμματα, το Basic Program Analyser System (BPAS) και το Automatic Test Case Generation System (ATCGS). Το πρώτο πρόγραμμα έχει την δυνατότητα να αναλύει ένα πρόγραμμα γραμμένο σε Java και στην συνέχεια μέσα από αυτή την ανάλυση να δημιουργεί τους γράφους ροών ελέγχου, να ανιχνεύει τον κώδικα που έχει καλυφθεί και στην συνέχεια να παρουσιάζει όλα τα δεδομένα στους προηγούμενους γράφους. Από την άλλη το ATCGS ανακτά τα δεδομένα του προγράμματος χρησιμοποιώντας το BPAS και στην συνέχεια αναλύει το πεδίο τιμών της εισόδου και καθορίζει ένα αριθμό από TC τα οποία έχουν την καλύτερη κάλυψη ως προς το edge/condition coverage. Το πιο πάνω περιλαμβάνει τους αλγόριθμους Batch Optimistic (BO) και Close-Up (CU). Ο πρώτος υλοποιεί ένα ειδικά σχεδιασμένο γενετικό αλγόριθμο για την εξέλιξη των Test Cases με την βοήθεια του ολοκληρωμένου γράφου ροής ελέγχου. Ο CU τρέχει μετά τον BO και σε συγκεκριμένα μονοπάτια έτσι ώστε να βρεθούν Test Cases για τα στοιχεία που δεν έχουν ακόμα καλυφθεί.

Για να χρησιμοποιηθεί η προσέγγιση που προτείνεται στη συγκεκριμένη ΑΔΕ θα πρέπει να γίνουν μερικές μετατροπές στα πιο πάνω προγράμματα έτσι ώστε να εξυπηρετούνται οι διαδικασίες έτσι όπως έχουν οριστεί σε προηγούμενα κεφάλαια. Σε βασικές γραμμές θα πρέπει μέσα στο πρόγραμμα ATCGS να προστεθούν επιπλέον έλεγχοι για τα καλυπτόμενα JML Statements όπως επίσης και περιορισμοί στην λήψη τιμών για την

δημιουργία των TC μέσα από το πεδίο τιμών που ορίζουν τα preconditions της ελεγχόμενης μεθόδου. Η προηγούμενη μετατροπή μπορεί να γίνει με την ενσωμάτωση στον αλγόριθμο παραγωγής TC μία πισίνα τιμών από όπου θα επιλέγονται οι τιμές για κάθε TC, η οποία θα δημιουργηθεί με την χρήση του jmlrac για την ανίχνευση των preconditions της μεθόδου. Στον BPAS θα πρέπει να μετατραπεί η ροή ελέγχου έτσι ώστε να μην περιλαμβάνει διακλαδώσεις που δεν περιέχουν μέσα δηλώσεις JML. Αυτός ο έλεγχος μπορεί να γίνει και σε αυτή την περίπτωση με το jmlrac εφόσον μπορούμε να ελέγξουμε τον αριθμό των ελεγμένων δηλώσεων και συγκεκριμένα την καταμέτρηση των ικανοποιήσεων και ανικανοποίητων προδιαγραφών JML.

6.3 Πιθανά προβλήματα

Παρά τις ευεργετικές ιδιότητες και την επίλυση των προβλημάτων που προσφέρει η νέα αυτή προσέγγιση σε σχέση με την υπάρχουσα, λόγω της ανάγκης για επιπλέον λειτουργίες μπορεί να παρουσιαστούν καινούρια προβλήματα ως προς την υλοποίηση και τον σχεδιασμό της νέας προσέγγισης.

- Ο έλεγχος των JML statements μπορεί να γίνει μόνο με την καταγραφή κατά τον έλεγχο τους και όχι με την καταγραφή της ύπαρξης τους. Άρα εάν δεν ελεγχθούν κατά την διάρκεια εκτέλεσης σε κάποια διακλάδωση δεν μπορούμε να είμαστε σίγουροι ότι δεν υπάρχουν δηλώσεις JML στο συγκεκριμένο σημείο όπως και όταν υπολογίζουμε το Fitness Value για ένα Test Case.
- Για τον έλεγχο των Test Cases με την χρήση του jmlunit έτσι ώστε να προσδιορίσουμε την ύπαρξη των δηλώσεων JML απαιτείται από τα TC να είναι στην συμβατή μορφή σύμφωνα με το JUnit Framework. Ο αλγόριθμος του ATCGS κωδικοποιεί τα TC με άλλη μορφή έτσι ένα TC δεν είναι συμβατό και με τα δύο Frameworks.

6.4 Συμπεράσματα

Κατά την σύγκριση των δύο προσεγγίσεων, αυτής που παρουσιάζεται στο συγκεκριμένο έγγραφο και αυτής που ακολουθείται στο εργαλείο JET είναι εύκολο να συμπεράνουμε ότι τα κυριότερα προβλήματα που παρουσιάζονται από την χρήση του JET έχουν λυθεί. Πλέον δεν υπάρχουν "meaningless" Test Cases εφόσον οι τιμές των παραμέτρων δίνονται σε όλες τις περιπτώσεις μέσα από το πεδίο τιμών που ορίζουν τα preconditions της ελεγχόμενης μεθόδου. Επίσης με την νέα προσέγγιση μπορούμε να εγγυηθούμε σε μεγάλο βαθμό ότι κατά τον έλεγχου του λογισμικού θα παραχθούν αρκετά Test Cases τα οποία θα μπορούν να καλύψουν το μεγαλύτερο μέρος των δηλώσεων JML μέσα στον κώδικα.

Μέσα από την αξιολόγηση που έγινε για σύγκριση ενός αλγόριθμου τυχαίας παραγωγής Test Cases και του προτεινόμενου πακέτου των Α. Ανδρέου και Α. Σοφοκλέους είναι εύκολο να παρατηρήσουμε ότι ενώ πετυχαίνουμε μεγαλύτερη κάλυψη με κριτήριο το edge/condition χρειαζόμαστε περισσότερο χρόνο και περισσότερα Test Cases για να πετύχουμε τα επιθυμητά αποτελέσματα. Όμως λαμβάνοντας υπόψη ότι ο έλεγχος γίνεται πλέον πιο μεθοδικά και πιο περιεκτικά μπορούμε να κατανοήσουμε ότι αξίζει να εκμεταλλευτούμε αυτή την ανταλλαγή προς όφελος μας εφόσον θα προσφέρει στο τέλος ένα καλύτερο λογισμικό.

Με την πρόσθεση των επιπλέον λειτουργιών για την εκμετάλλευση της γλώσσας προδιαγραφών JML ο βαθμός αποτελεσματικότητας και αποδοτικότητας των παραγόμενων TC για τον έλεγχο του λογισμικού γίνεται ακόμα πιο μεγάλος. Η παραγωγή σεναρίων δοκιμής με γνώμονα την κάλυψη των δηλώσεων της JML μέσα στον κώδικα μπορεί να προσφέρει ένα πολύτιμο εργαλείο για τους προγραμματιστές της Java τόσο στο στάδιο της επαλήθευσης και επικύρωσης του λογισμικού όσο και στα στάδια σχεδιασμού και υλοποίησης.

6.5 Μελλοντική εργασία

Σαν μελλοντική εργασία θα ήταν ωφέλιμο να δημιουργηθεί ένα API για μετατροπή των παραγόμενων Test Cases από το ATCGS σε JUnit συμβατό έτσι ώστε να μπορεί να γίνεται σωστά ο έλεγχος για την ύπαρξη και ικανοποίηση των JML δηλώσεων.

Ακόμα για να αποφευχθεί το πρόβλημα της ανικανότητας να παρέχουμε εγγύηση στον χρήστη ότι έχουν καλυφθεί όλες οι δηλώσεις JML μπορεί να δημιουργηθεί ένας λεξικός αναλυτής ο οποίος θα τα τρέχει κατά την λειτουργία του BPAS έτσι ώστε να καταγράφονται όλες οι δηλώσεις που βρίσκονται σε κάθε διακλάδωση του γράφου ροής ελέγχου

Τέλος μπορεί να αναπτυχθεί ένα λογισμικό ή ακόμα ένα plugin στο Eclipse το οποίο θα προσφέρει την υλοποίηση της πιο πάνω προσέγγισης σε συνδυασμό με το plugin που έχω δημιουργήσει στο πρώτο μέρος της ΑΔΕ, έτσι ώστε ένας προγραμματιστής Java να έχει την δυνατότητα για άμεσο έλεγχο των προδιαγραφών του πηγαίου κώδικα με τις ευκολίες της χρήσης που προσφέρει τόσο η ίδια η JML όσο και τα προτεινόμενα εργαλεία που έχουν παρουσιαστεί στην τρέχουσα ΑΔΕ.

Βιβλιογραφία

Άρθρα

[1] Gary T. Leavens, Yoonsik Cheon, “Design by Contract with JML”, September 2006

[2] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T, “An overview of JML tools and applications”, 2004

[3] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll, “Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2”, 2001

[4] Du, Wenliang and Mathur, Aditya P., “Categorization of Software Errors that led to Security Breaches”,1998

[5] Κωνσταντίνος Ιωάννου, ΕΛΕΓΧΟΣ ΠΡΟΔΙΑΓΡΑΦΩΝ ΛΟΓΙΣΜΙΚΟΥ ΣΕ ΕΠΙΠΕΔΟ ΠΗΓΑΙΟΥ ΚΩΔΙΚΑ ΜΕ ΤΗ ΧΡΗΣΗ ΤΗΣ JAVA MODELING LANGUAGE(JML), 2008

[6] K. Rustan M. Leino, Greg Nelson, and James B. Saxe, “ESC/Java User's Manual”,2000

[7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby, “JML: A Notation for Detailed Design”,1999

[8] Iowa State University, Department of Computer Science, ”Tools, examples, and documentation for the Java Modeling Language (JML)”

[9] Α.Α. Σοφοκλέους, Α.Σ. Ανδρέου, “Automatic, evolutionary test data generation for dynamic software testing”, 2008

[10] Chris Aniszczyk, “Plug-in development 101: The fundamentals”, 2008

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual (DRAFT), May 2008

[12] Meyer B., “Design By Contract. The Eiffel Method”,1998

[13] Yoonsik Cheon, Myoung Yee Kim, and Ashaveena Perumandla, “A Complete Automation of Unit Testing for Java Programs”, 2005

[14] Gilbert Thomas Laycock, “The theory and practice of specification based testing”,1993
[15] Nilesh Parekh, “Software Testing – White Box Testing Strategy”, 2005
[16] Nilesh Parekh, “Software Testing – Black Box Testing Strategy”, 2005
[17] Gary T. Leavens, “An Overview of Larch/C++: Behavioral Specifications for C++ Modules”,1996

Ιστοσελίδες

[18] http://en.wikipedia.org/wiki/Software_testing
[19] http://en.wikipedia.org/wiki/Code_coverage
[20] http://en.wikipedia.org/wiki/Model-based_testing
[21] http://en.wikipedia.org/wiki/Dead_code
[22] http://www.cs.ucf.edu/~leavens/JML
[23] http://www.eiffel.com/
[24] http://www.eecs.ucf.edu/~leavens/larch-faq.html
[25] http://users.ecs.soton.ac.uk/mjb/refcalc-tut/home.html
[26] http://en.wikipedia.org/wiki/Computer_bug
[27] http://en.wikipedia.org/wiki/Plug-in_(computing)
[28] http://en.wikipedia.org/wiki/Eclipse_(software)
[29] http://www.jdg2e.com/ch21.actions.table/doc/index.html
[30] http://pm.ethz.ch/research/universes/tools/eclipse/
[31] http://autojml.sourceforge.net/
[32] http://www.sireum.org/
[33] http://en.wikipedia.org/wiki/Microsoft_Visual_Studio
[34] http://en.wikipedia.org/wiki/J_Sharp
[35] http://www.sct.ethz.ch/research/universes/tools/eclipse/