ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΔΙΕΠΑΦΕΣ ΕΦΑΡΜΟΓΩΝ & ΜΕΤΡΙΚΕΣ ΛΟΓΙΣΜΙΚΟΥ (API's & METRICS)

Του φοιτητή
Γανώση Δημήτρη
Αρ. Μητρώου: 03/2304

Επιβλέπων καθηγητής
Δεληγιάννης Ιγνάτιος

Θεσσαλονίκη 2010

Η εικόνα στο εξώφυλλο είναι ο πίνακας *Still Life with Chair Caning* (1912) του Pablo Picasso, το πρώτο έργο του συνθετικού κυβισμού.

Ο κυβισμός είναι ένα κολάζ από κομμάτια μιας εικόνας σε μορφή κύβου τα οποία επανασυνδέονται αυτόματα ή με τυχαία σειρά, δημιουργώντας ξανά την εικόνα η οποία όμως εκφράζει πλέον διαφορετικές οπτικές γωνίες των αντικειμένων της.

Η σύνδεση διαφόρων κομματιών για την συμπλήρωση ενός έργου αποτελεί και την ουσία των API's όταν ορισμένα εξ'αυτών είναι προκατασκευασμένα είτε ως αυτοτελή είτε ως συστατικά άλλου έργου.

Πρόλογος

Τα προγράμματα και οι εφαρμογές που αναπτύσσονται στην εποχή μας είναι ως προς το πλήθος τους, περισσότερα από ποτέ. Διαρκώς δημιουργούνται καινούργιες ανάγκες που πρέπει να καλυφθούν από λογισμικά το ταχύτερο δυνατόν. Αναμενόμενη είναι και η αναγκαία ύπαρξη τρόπων διευκόλυνσης της συγγραφής του κώδικα.

Οι προγραμματιστικές διεπαφές εφαρμογών εξυπηρετούν αυτή την ανάγκη και επιτρέπουν στους προγραμματιστές να αναπτύσσουν προγράμματα όχι από το μηδέν αλλά χρησιμοποιώντας βιβλιοθήκες που παρέχουν έτοιμες λειτουργίες.

Ο στόχος βέβαια πάντα παραμένει τα προγράμματα να είναι όσο το δυνατό περισσότερο αξιόπιστα, σταθερά, εύκολα συντηρήσιμα και συμβατά με προηγούμενες και επόμενες εκδόσεις. Αυτό απαιτεί την τήρηση σχεδιαστικών κανόνων και την επιβεβαίωση της ορθότητας της υλοποίησης μέσω των μετρικών λογισμικού.

Οι μετρικές λογισμικού διακρίνονται ανάλογα με τα χαρακτηριστικά τους και την λειτουργία τους. Παρέχουν την δυνατότητα ελέγχου της ορθότητας του κώδικα μέσω διαφόρων κριτηρίων, ώστε ο προγραμματιστής να ανατρέξει και να διορθώσει τυχόν ασυνέπειες εγκυροποιώντας το έργο του.

Ο συνδυασμός των προγραμματιστικών διεπαφών των εφαρμογών και των μετρικών λογισμικού αποτελούν σημαντικά εργαλεία για την λιγότερο δύσκολη ανάπτυξη έγκυρων προγραμμάτων. Η χρήση τους επίσης, που πλέον είναι απαραίτητη, αυξάνει τις πιθανότητες εμπορικής επιτυχίας του προϊόντος αλλά και της υπεροχής του προγραμματιστή.

Περίληψη

Η παρούσα πτυχιακή εργασία καλύπτει δύο ζητήματα: τις προγραμματιστικές διεπαφές εφαρμογών και τις μετρικές λογισμικού. Παρέχει πέρα από τους ορισμούς και τις λειτουργίες τους, μία σειρά παραδειγμάτων σε java, διαγράμματα και εμπειρικές μελέτες και για τα δύο θέματα.

Οι προγραμματιστικές διεπαφές εφαρμογών αναλύονται για τις χρήσεις τους και διαχωρίζονται ως προς τις ανάγκες που καλύπτουν. Μελετάται ο σχεδιασμός τους, η ανάγκη ύπαρξης δοκιμών σ'αυτές καθώς παρατίθενται και διάφορες οδηγίες έγκυρης ανάπτυξης διεπαφών. Επιπλέον παρουσιάζονται έννοιες και χαρακτηριστικά που αποτελούν κριτήρια ελέγχου της ποιότητας των διεπαφών.

Οι μετρικές λογισμικού κατηγοριοποιούνται βάσει χαρακτηριστικών και αναλύονται ως προς την λειτουργία τους. Ένα πλήθος αυτών, καλύπτουν θεωρητικά και εμπειρικά το σύνολο των ελέγχων σε προγράμματα, βιβλιοθήκες και διεπαφές. Παρουσιάζονται και αναλύονται καθιερωμένα σύνολα μετρικών που χρησιμοποιούνται για την εγκυρότητα του κώδικα από πλευρά μεγέθους, πολυπλοκότητας, συνοχής, σύζευξης και κληρονομικότητας.

Επιπλέον, παρατίθεται ένας οδηγός χρήσης του λογισμικού που χρησιμοποιείται για την εφαρμογή μετρικών καθώς και ένα παράδειγμα εκτέλεσης προγράμματος και εφαρμογής των μετρικών σε αυτό.

Abstract

The present thesis covers two subjects: Application Programming Interfaces and Metrics. It provides, further than their definitions and functions, a number of examples in java, diagrams and empiric studies on both subjects.

"Application Programming Interfaces" are analyzed for their uses and they are separated as for the needs they cover. Their design and the need for testing them are also studied. Furthermore various directives of valid development Interfaces are provided. Moreover are presented significances and characteristics that constitute criteria of quality control of Interfaces.

"Metrics" are categorized based on their characteristics and they are analyzed as for their function. A number of these cover theoretically and empirically the total of controls in programs, libraries and interfaces. Established suites of metrics that are used for the validity of code concerning size, complexity, coupling, cohesion and inheritance are presented and analyzed.

A manual is also provided for the software that is used for the application of metrics as well as an example of program implementation and of metrics' applications on it.

Ευχαριστίες

Θέλω να ευχαριστήσω τον προϊστάμενο του τμήματος Πληροφορικής και επιβλέποντα της πτυχιακής μου καθηγητή κ. Δεληγιάννη Ιγνάτιο αρχικά που με εμπιστεύτηκε και μου έδωσε την παρούσα πτυχιακή εργασία με απευθείας ανάθεση. Με την καθοδήγηση και την βοήθεια του, όχι μόνο καλύφθηκαν οι στόχοι της εργασίας αλλά επιπλέον μου έμαθε την σημασία της έρευνας. Σημαντικότερο όλων, μου μετέδωσε πάθος για την πληροφορική ώστε τα θέματα που με ενδιαφέρουν, να τα επεκτείνω μόνος μου και εκτός σχολής.

Επίσης θέλω να ευχαριστήσω τον επίκουρο καθηγητή εφαρμοσμένης πληροφορικής του Πανεπιστημίου Μακεδονίας κ. Χατζηγεωργίου Αλέξανδρο, που εξ αρχής στάθηκε αρωγός της πτυχιακής μου εργασίας. Η συμβολή του ήταν καθοριστική για την πορεία της πτυχιακής μου και η βοήθεια του αποτέλεσε αποσαφήνιση σε πολλά θέματα που αρχικά μου έμοιαζαν ακατανόητα και μη αντιμετωπίσιμα.

Ευρετήριο Περιεχομένων

Εισαγωγή

Οι στόχοι της πτυχιακής εργασίας είναι αφενός να μελετήσει τις προγραμματιστικές διεπαφές εφαρμογών πλήρως αφετέρου να αναλύσει πλήθος μετρικών λογισμικού ως προς την λειτουργία τους και την χρησιμότητα τους.

Στο πρώτο κεφάλαιο, που αφορά τις προγραμματιστικές διεπαφές εφαρμογών, αναλύονται η λειτουργία και τα οφέλη χρήσης τους με παραδείγματα και εμπειρικές μελέτες που έχουν διεξαχθεί στο παρελθόν. Αναλύεται εκτεταμένα ένα πλήθος σχεδιαστικών κανόνων με παραδείγματα κώδικα σε java. Παρουσιάζεται λεπτομερειακά η ανάγκη ύπαρξης δοκιμών σε κάθε πρόγραμμα αλλά με έμφαση στις διεπαφές. Παρέχεται μία σειρά συμβουλών από διάφορους καταξιωμένους προγραμματιστές για την πιο έγκυρη ανάπτυξη προγραμματιστικών διεπαφών εφαρμογών. Τέλος αναπτύσσονται τρόποι ελέγχου της ποιότητας των διεπαφών με την παράθεση βασικών εννοιών και χαρακτηριστικών που αποτελούν κριτήρια ελέγχου.

Το δεύτερο κεφάλαιο αφορά τις μετρικές λογισμικού. Αρχικά ορίζονται και διαχωρίζονται με βάση χαρακτηριστικά και έπειτα παρουσιάζονται αναλυτικά έννοιες που τα καθορίζουν σε διάφορες κατηγορίες. Μελετώνται δεκατέσσερις μετρικές με παραδείγματα, διαγράμματα, πειράματα και εμπειρικές μελέτες. Οι μετρικές που μελετώνται καλύπτουν κατά σειρά τα εξής θέματα: το μέγεθος, την πολυπλοκότητα, την κληρονομικότητα, την συνοχή, την σύζευξη και το ποσοστό κοινών ιδεών.

Επιπλέον παρουσιάζεται οδηγός χρήσης του λογισμικού που χρησιμοποιείται. Στον οδηγό χρήσης καλύπτονται όλες οι απαραίτητες ενέργειες για την λήψη και χρήση του Eclipse Galileo, της δημιουργίας ενός java έργου, της λήψης και εγκατάστασης του επιπλέον απαραίτητου προγράμματος για την εφαρμογή μετρικών. Επίσης παρουσιάζεται ένα παράδειγμα εκτέλεσης προγράμματος και ταυτόχρονης εφαρμογής μετρικών, καθώς και η εμφάνιση των τιμών της κάθε μετρικής.

Τέλος αναφέρονται τα συμπεράσματα και οι προτάσεις της πτυχιακής εργασίας, ως προς την αναγκαιότητα χρήσης των μετρικών και των προγραμματιστικών διεπαφών των εφαρμογών καθώς και της τήρησης των σχεδιαστικών κανόνων.

Κεφάλαιο 1

Προγραμματιστική Διεπαφή Εφαρμογών

Application **P**rogramming **I**nterface

Διεπαφή προγραμματισμού εφαρμογών καλούμε, την διεπαφή (interface), την οποία ένα υπολογιστικό σύστημα, βιβλιοθήκη ή διαδικτυακή εφαρμογή παρέχει, προκειμένου να επιτρέψει να γίνουν αιτήσεις από άλλα προγράμματα προς αυτό, ή/και ανταλλαγή δεδομένων με άλλα προγράμματα.

Ένας από τους πρωταρχικούς σκοπούς μιας διεπαφής API, είναι να διατυπώσει το σύνολο των λειτουργιών-υπηρεσιών που μπορεί να παρέχει ένα λειτουργικό σύστημα, μια διαδικτυακή υπηρεσία και λοιπά, σε άλλα προγράμματα χωρίς να γίνεται κάποια αναφορά στον κώδικα που υλοποιεί αυτές τις υπηρεσίες. Το API απλά ορίζει με ποιες εξωτερικές εντολές θα παρέχει την αμφίδρομη επικοινωνία με την υπηρεσία που θέλει να συνδεθεί προς αυτό χωρίς να αποκαλύπτει τον πηγαίο κώδικά του.

Ένα API παρομοιάζει το λογισμικό με υπηρεσία Software as a Service (SaaS), ώστε οι μηχανικοί λογισμικού να μην χρειάζεται να γράφουν από το μηδέν κάθε φορά κάποιο πρόγραμμα. Αντιθέτως η κατεύθυνση είναι να κατασκευαστεί ο πυρήνας μιας εφαρμογής που να μπορεί να επαναχρησιμοποιηθεί σε μια πληθώρα περιπτώσεων.

Οφέλη χρήσης των API’s

- Εξοικονόμηση χρόνου παρέχοντας λειτουργικότητα στον προγραμματιστή και δυνατότητα επαναχρησιμοποίησης του κώδικα
- Παροχή πληροφοριών των εφαρμογών με δυνατότητα τροποποίησης των χωρίς να επηρεάζεται ο κώδικας
- Προώθηση των εφαρμογών στους τελικούς χρήστες

Γενικευμένα API's με παραδείγματα

Στον πίνακα Gen_API_I παρουσιάζεται μια σειρά διεπαφών με παραδείγματα.

Πίνακας_API_I

Libraries	Math library "standard" library in C
Frameworks	.NET Framework Eclipse Framework
Development Kits	.NET Development Kit Java Development Kit
Toolkits	The GIMP Toolkit (GTK) Google Web Toolkit
APIs	Win32 APIs Google Map APIs

Τι δεν είναι τα API's

- Γλώσσα προγραμματισμού
- Εργαλεία
- Τεκμηρίωση (Documentation)
- Κώδικας παραδειγμάτων
- Πηγαίος κώδικας μιας εφαρμογής

Συμμετέχοντες – εμπλεκόμενοι

- **Σχεδιαστές API's**

Στόχοι τους: μεγιστοποίηση των εκδόσεων ενός API, ελαχιστοποίηση του κόστους ανάπτυξης και δυνατότητα συντήρησης του σε σύγχρονο πλαίσιο.

- **Χρήστες API's** είναι οι προγραμματιστές.

Στόχοι τους: να γράφουν κώδικα γρήγορα και ελεύθερα από λάθη, να επαναχρησιμοποιούν προγράμματα άλλων χρηστών και να τα εκτελούν γρήγορα και αποδοτικά.

- **Καταναλωτές API's**

Στόχοι τους: Προϊόντα με πληρότητα επιθυμητών χαρακτηριστικών χωρίς ελαττώματα, με σταθερότητα, συμπεριλαμβάνοντας τα βασικά εργαλεία

Μία πολύ σημαντική προγραμματιστική ενέργεια είναι η χρήση της διεπαφής προγραμματισμού εφαρμογής (API), πλαίσια λογισμικού (frameworks), εργαλεία (toolkits) και βιβλιοθήκες (libraries). Οι προγραμματιστές που υλοποιούν λειτουργικότητες χρειάζεται να καταλάβουν ποια API να χρησιμοποιήσουν και πως να τα συνδυάσουν. Οι προγραμματιστές που διαβάζουν ή τροποποιούν κώδικα πρέπει να κατανοήσουν πως λειτουργεί ο υπάρχων κώδικας ενός API, τι συσχετίσεις κάνει ο κώδικας και πως μπορεί να προσθέσει ή να αλλάξει τον κώδικα χωρίς να σπάσουν αυτές οι συσχετίσεις. Αυτές οι εργασίες είναι εξαιρετικά απαιτητικές με το αυξανόμενο πλήθος και μέγεθος των API. Υπάρχουν πολλές διαθέσιμες επιλογές ως προς το framework που θα χρησιμοποιηθεί και το κάθε framework με τη σειρά του έχει δεκάδες χιλιάδες κλάσεις και μεθόδους.

Ένας τρόπος για να επιλυθεί το πρόβλημα αυτό είναι ο σχεδιασμός API που να είναι πιο χρήσιμα σε καθορισμένες ομάδες. Καλά σχεδιασμένα API μπορούν να είναι πιο γρήγορα, πιο αποδοτικά και με λιγότερα σφάλματα. Παρόλα αυτά, ένας σχεδιαστής είναι δύσκολο να προβλέψει από την αρχή, την χρησιμότητα των API που σχεδιάζει και επίσης δύσκολο να την επηρεάσει μετά την διανομή του API εξαιτίας κυρίως θεμάτων συμβατότητας .

Για να μελετηθεί η χρησιμότητα ενός API θα πρέπει να αναγνωριστούν οι κοινές εργασίες που είναι σχεδιασμένο να παρέχει και να παρατηρηθεί η προσπάθεια που καταβάλλεται από τους προγραμματιστές για να εκτελέσουν αυτές τις εργασίες. Διενεργώντας αυτές τις μελέτες σε ένα εργαστήριο, καταγράφοντας τες και ζητώντας από τους προγραμματιστές να τις σχολιάζουν δυνατά καθώς τις εκτελούν, μπορεί κάποιος εύκολα να αναγνωρίσει τα προβλήματα που δημιουργούνται από την διαφορετικότητα των χρηστών καθώς και άλλες αιτίες προβλημάτων.

Όταν δημιουργείται ένα νέο API, οι σχεδιαστές πρέπει να πάρουν κάποιες αποφάσεις. Οι αποφάσεις αυτές επηρεάζουν την ποιότητα των τελικών API ως προς την απόδοση τους και άλλα ποιοτικά χαρακτηριστικά (όπως η ταχύτητα και η μνήμη), η δύναμη (η εκφρασιμότητα, η επεκτασιμότητα και η εξελιξιμότητα) και η χρησιμότητα (ικανότητα εκμάθησης, παραγωγικότητα και πρόβλεψη σφαλμάτων). Έμπειροι σχεδιαστές έχουν γράψει οδηγίες για τον σχεδιασμό των API, παρέχοντας διάφορες απόψεις για τις προαναφερθείσες αποφάσεις. Επιπλέον, εμπειρικές μελέτες έχουν μετρήσει την χρησιμότητα συγκεκριμένων σχεδιαστικών αποφάσεων.

Η χρήση των API γίνεται ολοένα και περισσότερο το μεγαλύτερο τμήμα του προγραμματισμού. Υπάρχουν πλέον περισσότερα API από ποτέ και το μέγεθος τους συνεχώς αυξάνεται. Μεγάλα API όπως της Microsoft .NET Framework ή της Java έχουν χιλιάδες κλάσεις με δεκάδες χιλιάδες μεθόδους που συνεχώς αυξάνονται με κάθε νέα έκδοση.

Όπως ήδη ανέφερα ένα κίνητρο για τα API είναι η βελτίωση της παραγωγικότητας των προγραμματιστών με την επαναχρησιμοποίηση όλο και περισσότερου κώδικα αντί της σύνταξης του από το μηδέν. Ωστόσο, η ύπαρξη πολλών API σημαίνει ότι η χρησιμότητα τους μπορεί να σταθεί εμπόδιο στην παραγωγικότητα των προγραμματιστών. Σε μερικές περιπτώσεις το να αποφασίσεις ποιο API θα χρησιμοποιήσεις μπορεί να είναι πιο χρονοβόρο και από το να γράψεις τον κώδικα από την αρχή.

Είναι δύσκολο να γραφτεί ένα καλό API το οποίο θα μπορεί να χρησιμοποιηθεί από το ευρύ κοινό και κυρίως σε διεθνές επίπεδο. Ο καθένας έχει το δικό του τρόπο κατανόησης και τον δικό του τρόπο να αναλύει τα προβλήματα. Η ικανοποίηση όλων αυτών των παραμέτρων είναι πραγματικά δύσκολη. Αυτή η διαδικασία γίνεται ακόμα δυσκολότερη όταν το API απευθύνεται σε παγκόσμιο επίπεδο, που χρειάζεται να αντιμετωπιστούν ποικίλες πολιτισμικές διαφορές.

Η κατάσταση στην οποία δεν έχουμε κάποιο στοιχείο λέγεται Cluelessness και είναι βασικό αξίωμα που επιτρέπει να αξιολογηθεί κατά πόσον μια συμβουλή είναι καλή ή όχι. Αν κάτι επιτρέπει στους ανθρώπους να επιτύχουν περισσότερα, ενώ γνωρίζουν λιγότερα, τότε αυτό είναι σπουδαίο. Είναι η καθημερινότητα των περισσοτέρων μας. Είναι το αποτέλεσμα της συγχώνευσης του ορθολογισμού και της υπάρχουσας εμπειρίας. Βρίσκεται παντού ακόμα και στον τρόπο που προγραμματίζουμε λογισμικό. Ως αποτέλεσμα, σχεδόν κάθε εξελιγμένη τεχνολογία, πρέπει να εμφανίζει στοιχεία αυτής της κατάστασης (cluelessness).

Αυτή η κατάσταση επηρεάζει τον τρόπο που αντιμετωπίζουμε την πραγματικότητα, τεκμηριώνει την σπουδαιότητα των δοκιμών και εξηγεί την αναγκαιότητα της δημιουργίας των API με σωστό τρόπο. Όλα αυτά συμβάλλουν στην αύξηση της cluelessness των συμβολομεταφραστών στις εφαρμογές μας, βελτιώνοντας παράλληλα την ποιότητά τους.

Το μοντέλο cluelessness βασίζεται στη χρήση μεγάλων δομικών στοιχείων που συλλέχθηκαν από έργα λογισμικού σε όλο τον κόσμο. Με την επαναχρησιμοποίηση όσο το δυνατόν περισσότερων και την μη συγγραφή των από το μηδέν, οι ομάδες προϊόντων μπορούν να εστιάσουν στην σημαντικότερη διαφοροποίηση: την πραγματική λογική της εφαρμογής τους. Δεν χρειάζεται πλέον να ξοδεύουν τον χρόνο τους στη δημιουργία και στην συγγραφή της υποδομής (infrastructure). Αντίθετα, μπορούν να επαναχρησιμοποιούν πλαίσια (frameworks) και άλλες χρήσιμες βιβλιοθήκες που παράγονται από άλλους. Κανείς δεν γράφει πλέον από το μηδέν ένα διακομιστή βάσης δεδομένων.

Οι άνθρωποι επαναχρησιμοποιούν εξυπηρετητές βάσεων δεδομένων που παρέχονται από καταξιωμένες εταιρίες ή από διάφορες εναλλακτικές λύσεις ανοικτού κώδικα. Παράλληλα η υλοποίηση ενός διακομιστή βάσης δεδομένων SQL για ιδιωτική χρήση θα πρέπει να θεωρηθεί ως μια κραυγαλέα σπατάλη πόρων. Το ίδιο συμβαίνει και σε πολλούς άλλους τομείς της μηχανικής λογισμικού.

Διαδικτυακοί διακομιστές και διακομιστές εφαρμογών, γλώσσες και βιβλιοθήκες, ολοκληρωμένα περιβάλλοντα ανάπτυξης (IDEs) γίνονται όλα καθιερωμένα δομικά στοιχεία. Ακριβώς όπως είναι τα προκατασκευασμένα έπιπλα του ΙΚΕΑ για τα διαμερίσματα που μπορούν να ληφθούν από το ράφι, να συναρμολογηθούν και στη συνέχεια γυαλισμένα όπως απαιτείται, να είναι έτοιμα προς χρήση . Η προσέγγιση αυτή μειώνει σε μεγάλο βαθμό το χρόνο ανάπτυξης που απαιτείται για την παραγωγή ενός συστήματος λογισμικού.

Κάθε συστατικό του API είναι ένα πρώτο βήμα προς την αύξηση της cluelessness σας. Δεν είναι απαραίτητο να κατανοήσουμε και να γνωρίζουμε όλα τα στοιχεία του κάθε συστατικού. Συνήθως είναι αρκετό να διαβάσουμε το εγχειρίδιο χρήσης του συστατικού και με την χρήση του, να συμπεριλάβουμε το API με επιτυχία στην εφαρμογή μας. Το API ελαχιστοποιεί την ανάγκη να κατανοήσουμε την κάθε λεπτομέρεια ενός συστατικού.

Μια τυπική εφαρμογή δεν αποτελείται από μία ή κάποιες βιβλιοθήκες. Οι εφαρμογές που έχουν αναπτυχθεί σήμερα κάνουν χρήση πολλών ανοικτού κώδικα βιβλιοθηκών. Η αρχή της διάθεσης των ανοικτού κώδικα βιβλιοθηκών έγινε με το Unix. Στην πραγματικότητα, κάθε μία από αυτές τις βιβλιοθήκες έχει το δικό της API και ως εκ τούτου, ο καθένας που γράφει ένα τέτοιο λογισμικό εντάσσεται στον σχεδιασμό των API είτε συνειδητά είτε όχι.

Οι βιβλιοθήκες ανοικτού κώδικα δεν είναι μόνο η κινητήρια δύναμη πίσω από την αυξανόμενη ανάγκη για ένα καλό API. Οι εμπορικοί πωλητές παράγουν επίσης πολλές τέτοιες βιβλιοθήκες και πλαίσια (frameworks). Πολλοί από αυτούς είτε υλοποιώντας κάποια υφιστάμενα πρότυπα, όπως η SQL είτε παρέχοντας τα δικά τους API's. Ωστόσο, οι κινήσεις αυτές, του ανοικτού κώδικα, με την απαιτούμενη αδειοδότηση έχουν γίνει η πρωταρχική πηγή των συστατικών για να χρησιμοποιούνται ως επαναχρησιμοποιήσιμα κατασκευαστικά στοιχεία. Οι ανοικτού κώδικα λύσεις είναι γνωστές από τους τελικούς χρήστες οι οποίοι μπορούν να τις χρησιμοποιούν χωρίς καμία χρέωση. Ωστόσο, επειδή δεν υπάρχουν περιορισμοί αδειών, είναι σημαντικό και για τους προγραμματιστές. Είναι εύκολο να πάρει ένα υπάρχον συστατικό και να το χρησιμοποιήσει σαν μέρος της δικής του εφαρμογής. Είναι σχεδόν πάντα, μόνο θέμα χρόνου πριν κάποιος το κάνει. Αυτό σημαίνει ότι η κάθε ανοικτού κώδικα εφαρμογή αργά ή γρήγορα θα χρειαστεί ένα API. Αυτά τα συστατικά αναπτύχθηκαν από μια σειρά διάφορων προγραμματιστών, από φοιτητές του πανεπιστημίου που ξεκίνησαν τα δικά τους έργα, από επαγγελματίες προγραμματιστές που βλέπουν τον συγκεκριμένο τρόπο ανάπτυξης ως μια επιχειρηματική ευκαιρία. Αυτοί οι άνθρωποι έχουν διαφορετικές ικανότητες και διαφορετικούς τρόπους εργασίας. Ανεξάρτητα αυτού, είναι σημαντικό να δημιουργούν καλά API's, διότι το API είναι το πρώτο βήμα για να λειτουργήσουν οι βιβλιοθήκες τους στην cluelessness. Όσο περισσότερες βιβλιοθήκες και πλαίσια έχουμε, τόσο το καλύτερο. Ωστόσο, είναι σχεδόν απαραίτητο για την επιτυχία της επαναχρησιμοποίησης της cluelessness να αφήσουμε τα API's να αντανακλούν το εσωτερικό πνεύμα αυτών των βιβλιοθηκών. Για να είναι αξιόπιστο ένα API, θα πρέπει να μειώνει τα περιθώρια της κακής χρήσης του και ταυτόχρονα να είναι έτοιμο να εξελιχθεί. Ελάχιστα API είναι τέλεια στην πρώτη τους έκδοση.

Η σημαντική προϋπόθεση εδώ είναι ο διαχωρισμός. Διαχωρισμός όσον αφορά την ανάγκη κανόνων για τον σχεδιασμό και την διατήρηση ενός API. Αν δεν υπήρχε διαχωρισμός και ολόκληρο το προϊόν ήταν κατασκευασμένο με υψηλή σύζευξη, τότε θα δημιουργείτο μία φορά και θα το ξεχνούσαμε, αφού δεν θα υπήρχε ανάγκη να ασχοληθούμε μαζί του. Τα πραγματικά εμπορικά προϊόντα όμως που αναπτύσσονται βασισμένα σε πρότυπα συναρτησιακά στοιχεία, με συστατική αρχιτεκτονική, σε ομάδες που δεν γνωρίζουν κατ 'ανάγκη ο ένας τον άλλο και λειτουργούν σύμφωνα με ανεξάρτητα χρονοδιαγράμματα για την υλοποίηση των σχεδίων τους, υπάρχει ανάγκη για ένα σταθερό συμβόλαιο βάσει του οποίου θα γίνεται η όλη επικοινωνία και συνεργασία.

Αντί να εξετάζουν, να μελετούν και να προσπαθούν να καταλάβουν ένα API, οι προγραμματιστές συχνά απλώς καλούν μια μέθοδο από ένα API και εάν λειτουργεί τότε είναι ευχαριστημένοι. Εάν όχι, τότε καλούν κάποια άλλη να δουν τι συμβαίνει. Μεταξύ των προγραμματιστών του NetBeans, αυτή η τακτική έχει γίνει γνωστή ως εμπειρικός προγραμματισμός: εκτέλεσε ένα πείραμα χρησιμοποιώντας το API, και αν αυτό δεν λειτουργήσει, δοκίμασε κάτι άλλο. Η εμπειρία προηγείται, η κατανόηση ακολουθεί, αλλά μερικές φορές δεν χρειάζεται καν. Αυτό επηρεάζει τη δομή του API. Το API θα πρέπει να αυτοτεκμηριώνεται. Δηλαδή να χρησιμοποιείται χωρίς καμία άλλη οδηγία χρήσης. Θα πρέπει να οδηγεί τον χρήστη στα στοιχεία με φυσικό τρόπο μέσω των εργασιών που πρέπει να εκτελεστούν. Οι λύσεις θα πρέπει να είναι εύκολα εντοπίσημες και παρόλο που ένα IDE προσφέρει διάφορες συμβουλές, θα πρέπει να είναι στο προσκήνιο χωρίς να αποσπά την προσοχή του χρήστη. Μόνο έτσι μπορεί τα πειράματα που χρησιμοποιούν ένα API να έχουν εμπειρική επιτυχία.

Σχεδιασμός των API's

Μην εκθέτετε περισσότερα από όσα θέλετε

Θεμελιώδη διλήμματα εμπλέκονται στην απόφαση αν κάτι πρέπει ή δεν πρέπει να αποτελεί μέρος ενός API. Η απάντηση σε αυτό, τουλάχιστον σύμφωνα με τις μεθοδολογίες προγραμματισμού, είναι η κατασκευή περιπτώσεων χρήσης. Μια έγκυρη περίπτωση χρήσης είναι η αιτιολόγηση μίας κλάσης ή μίας μεθόδου σε ένα API. Αν δεν υπάρχει περίπτωση χρήσης για μια συγκεκριμένη μέθοδο ή κλάση, ή αν η περίπτωση χρήσης είναι λίγο περίεργη, τότε είναι καλύτερα να αφήσουμε αυτό το στοιχείο εκτός του API. Ένας τρόπος να επιτευχθεί αυτό είναι να καταστεί ως ένα ιδιωτικό πακέτο. Όταν ένας χρήστης του API διαμαρτύρεται και ζητά από το στοιχείο να είναι ορατό, τότε έτσι μπορούμε να το κάνουμε ορατό. Ωστόσο, αυτό δεν πρέπει να γίνει νωρίτερα από την αίτηση του χρήστη και χωρίς κάποια πειστική περίπτωση χρήσης.

Έτσι, το πρώτο τμήμα της οδηγίας είναι να αρθούν όλα όσα μπορούν να αφαιρεθούν πριν την πρώτη έκδοση του API. Στην πραγματικότητα δεν είναι δυνατόν να καταργηθούν όλα. Πίσω από κάθε τέτοιο αλτρουισμό υπάρχει μια κρυφή ανάγκη, κάτι που οδηγεί και παρακινεί τον σχεδιασμό των API. Οι σχεδιαστές των API αισθάνονται αυτήν την ανάγκη για το έργο τους και είναι πρόθυμοι να την μοιραστούν. Ωστόσο η δημοσίευση ενός API είναι μόλις το πρώτο του βήμα. Με τη δημοσίευση αυτή, αυτόματα προκύπτει η ανάγκη για την παραγωγή και μετέπειτα εκδόσεων πιο αρμονικών που θα μπορούν να συντηρηθούν καλύτερα και να παρέχουν περισσότερα. Ωστόσο, είναι βασικό να αποφεύγονται οι υποσχέσεις ότι θα υπάρξει τρομακτική βελτίωση στο μέλλον. Η πιο απλή και αποτελεσματική λύση είναι να μείνει κρυφό το γεγονός ότι η ανάδραση με τους τελικούς χρήστες είναι η πολυτιμότερη βοήθεια των σχεδιαστών.

Η μέθοδος είναι καλύτερη της μεταβλητής

Είναι προτιμότερο να χρησιμοποιούνται μέθοδοι - τυπικά getters και setters για να υπάρχει πρόσβαση σε πεδία παρά αυτά τα πεδία να εκτίθενται άμεσα. Αυτό ισχύει πρώτον επειδή με την κλήση μιας μεθόδου μπορούμε να κάνουμε πολλά επιπλέον πράγματα, ενώ με την χρήση ενός πεδίου μπορούμε να γράψουμε ή να διαβάσουμε απλά την τιμή του. Όταν γίνεται χρήση της get μπορούμε να αρχικοποιούμε, να συγχρονίζουμε την πρόσβαση ή να συνθέτουμε την τιμή κάνοντας χρήση αλγορίθμων. Με την set από την άλλη πλευρά μπορούμε να ελέγχουμε την ορθότητα της δοθείσας τιμής ή να ενημερώνουμε τους χειριστές (listeners) όταν παρουσιάζεται κάποια αλλαγή. Ο άλλος λόγος προτίμησης των μεθόδων βρίσκεται στον προσδιορισμό των JVM (εικονικών μηχανών της Java). Εκεί είναι επιτρεπτή η μετακίνηση μιας μεθόδου από μία κλάση στην υπερκλάση της, με την διατήρηση της συμβατότητας της. Έτσι, μια μέθοδος που αρχικά εισήχθη ως Dimension javax.swing.JComponent.getPreferredSize (Dimension d) μπορεί να διαγραφεί σε μία νέα έκδοση και να μεταφερθεί στην Dimensionjava.awt.Component.getPreferredSize (Dimension d), αφού το Jcomponent είναι υποκλάση της Component. Αυτή η αλλαγή πραγματικά συνέβη στο JDK 1.2 και ήταν δυνατό επειδή το πεδίο ήταν σε ενθυλάκωση από την μέθοδο. Μία τέτοια λειτουργία δεν είναι δυνατή στα πεδία. Όταν ένα πεδίο ορίζεται σε μία κλάση πρέπει να μείνει εκεί για πάντα ώστε να διατηρείται η συμβατότητα του, επιπλέον λόγος ο οποίος εξηγεί γιατί να είναι τα πεδία ιδιωτικά.

Η μέθοδος αναδόμησης (factory) είναι καλύτερη του δομητή (constructor).

Η διευκόλυνση της μελλοντικής εξέλιξης ενός API είναι δυνατή, όταν εκτίθεται μία μέθοδος αναδόμησης αντί ενός δομητή Όταν ένας δομητής είναι διαθέσιμος σε ένα API, εξασφαλίζει όχι μόνο ότι θα δημιουργηθεί σε μία κλάση ένα εκχωρηθέν στοιχείο αλλά και ότι αυτό το στοιχείο θα είναι συγκεκριμένης κλάσης αφού δεν επιτρέπονται υποκλάσεις. Επιπλέον, ένα νέο στοιχείο θα δημιουργείται κάθε φορά. Εάν η μέθοδος αναδόμησης προωθηθεί αντί αυτού θα υπάρχει μεγαλύτερη ευελιξία. Η μέθοδος αναδόμησης είναι συνήθως μία στατική μέθοδος που δέχεται τις ίδιες παραμέτρους με τον δομητή και επιστρέφει ένα στοιχείο από την κλάση στην οποία έχει οριστεί και ο δομητής. Το πρώτο πλεονέκτημα της μεθόδου αναδόμησης είναι ότι δεν είναι απαραίτητο να επιστραφεί η ακριβής κλάση. Είναι δυνατόν να επιστραφεί η υποκλάση, επιτρέποντας έτσι τον πολυμορφισμό και πιθανώς την εκκαθάριση του κώδικα. Το δεύτερο πλεονέκτημα είναι η δυνατότητα απόκρυψης στοιχείων. Μπορούμε να κρύβουμε προηγούμενα αντικείμενα και να τα επαναχρησιμοποιούμε προς εξοικονόμηση μνήμης. Τρίτο πλεονέκτημα, υπάρχει καλύτερος συγχρονισμός με την μέδοθο αναδόμησης από ένα απλό δομητή που έχει περισσότερους περιορισμούς. Αυτό συμβαίνει επειδή η αναδόμηση μπορεί να συγχρονίζεται ως όλον, συμπεριλαμβάνοντας κώδικα πριν την δημιουργία των αντικειμένων, κώδικας που δημιουργεί τα στοιχεία καθώς και τον υπόλοιπο κώδικα που εκτελείται μετά από την δημιουργία του αντικειμένου. Αυτό δεν είναι δυνατόν να γίνει σε έναν δομητή. Κατά την επανασυγγραφή του NetBeans οι προγραμματιστές βρήκαν ακόμα ένα πλεονέκτημα των μεθόδων αναδόμησης: επιτρέπουν την επινόηση των παραμετροποιήσιμων τύπων για τα επιστρεφόμενα αντικείμενα, οι δομητές όχι.
Παράδειγμα αποτελεί η ακόλουθη κλάση στο API του NetBeans.

```
public final class Template extends Object {
   private final Class type;

   public Template(Class type) { this.type = type; }
   public Class getType() { return type; }


   public Template() { this(Object.class); }
}
```

Όταν μεταφέρθηκαν οι βιβλιοθήκες του NetBeans στο JDK 1.5, ήταν φυσικό να παραμετροποιηθεί το πρότυπο της κλάσης με έναν τύπο που αναγνώριζε το αντικείμενο της κλάσης. Και αυτό λειτούργησε καλά. . .

```
public final class Template<T> extends Object {
   private final Class<T> type;

   public Template(Class<T> type) { this.type = type; }
   public Class<T> getType() { return type; }

   // now what!?
   public Template() { this(Object.class); }
}
```

. . . μέχρι τον τελευταίο δομητή.

Θεωρείτο ότι θα δημιουργηθεί ένα στοιχείο του Template <αντικείμενο>, αλλά αυτό δεν μπορεί να γίνει με την Java 1.5. Η γλώσσα δεν είναι αρκετά ευέλικτη ώστε να εκφράσει αυτήν την κατάσταση. Αυτό θα μπορούσε να χαρακτηριστεί ως σχεδιαστικό σφάλμα. Ωστόσο, όταν σχεδιάστηκε αρχικά αυτή η κλάση, υπήρχαν πιο σημαντικά πράγματα που έπρεπε να επιλυθούν από την αναγνώριση του τι θα ταίριαζε στην Java 1.5 κυρίως σε συστήματα που δεν είχαν ακόμα σχεδιαστεί. Αργότερα, ήταν πολύ αργά. Η μόνη λύση που υπήρχε ήταν να αποδοκιμαστεί ο δομητής, να χρησιμοποιηθεί μια μετατροπή και να δοθεί η οδηγία στους χρήστες να χρησιμοποιούν τον άλλο δομητή, με την παραμετροποιημένη κλάση. Αν εξ'αρχής είχε ακολουθηθεί αυτή η συμβουλή, με την χρήση της μεθόδου αναδόμησης αντί του δομητή, η κατάσταση αυτή θα ήταν πολύ ευκολότερα αντιμετωπίσιμη. Θα δημιουργείτο μία create() μέθοδος αναδόμησης και θα καθιστούσε την μέθοδο, κάπως έτσι:

```
public final class Template<T> extends Object {
   private final Class<T> type;

   public Template(Class<T> type) { this.type = type; }
   public Class<T> getType() { return type; }

   @Deprecated
   @SuppressWarnings("unchecked")
   public Template() { this((Class<T>)Object.class); }

   public static Template<Object> create() {
       return new Template<Object>(Object.class);
   }
}
```

Αυτό το παράδειγμα δείχνει το μέγεθος της ευελιξίας των μεθόδων. Αυτό συμβαίνει επειδή οι μέθοδοι, συμπεριλαμβάνοντας τις μεθόδους αναδόμησης, δεν περιορίζονται από τον τύπο της εσωκλειόμενης κλάσης, σε αντίθεση με τους δομητές. Κι αυτό αποτελεί ακόμα έναν λόγο προτίμησης τους.

Κάνετε τα πάντα Final

Συχνά, οι άνθρωποι δεν σχεδιάζουν με τις υποκλάσεις στο μυαλό τους. Με αποτέλεσμα, ακούσιες και επικίνδυνες συνέπειες να απορρέουν από αυτό, καθώς το πλήθος των διαφορετικών τρόπων χρήσης των API αυξάνεται σημαντικά. Παράδειγμα αποτελεί η ακόλουθη μέθοδος σε μια υποκλάση:

```
public class Hello {
   public void hello() { System.out.println ("Hello"); }
}
```

Σε περίπτωση κλήσης της από εξωτερικό κώδικα μπορεί να προκληθούν προβλήματα.

```
public static void sayHello() {
    Hello hello = new Hello();
    hello.hello();
}
```

Εντωμεταξύ, μπορεί να υποστεί υπέρβαση και για μια άλλη εργασία:

```
private static class MyHello extends Hello {
   @Override
   public void hello() { System.out.println ("Hi"); }
}
```

Μπορεί να γίνει κι άλλη υπέρβαση με κλήση της super:

```
private static class SuperHello extends Hello {
// Override
public void hello() {
super.hello();
System.out.println("Hello once again");
        }
}
```

Είναι πολύ πιθανό, να εμφανιστεί ένα μεγαλύτερο φάσμα επιλογών από αυτό που προοριζόταν. Ωστόσο, η λύση είναι απλή: κάνοντας την κλάση Hello final!
Όταν γράφουμε ένα API χωρίς να θέλουμε να είναι επιτρεπτό στους χρήστες να κάνουν υποκλάσεις τις κλάσεις, είναι καλύτερο να απαγορεύεται αυτή η συμπεριφορά. Όπως παρουσιάζεται στο παράδειγμα της Hello, θα πρέπει μετά να παρασχεθούν τρείς διαφορετικοί τρόποι χρήσης των μεθόδων της κλάσης. Για να διευκολύνεται η μελλοντική εξέλιξη τους, είναι καλύτερα να απαγορευτεί αυτή η συμπεριφορά. Και πάλι, η απλούστερη λύση είναι να κάνουμε την κλάση final. Άλλες προσεγγίσεις περιλαμβάνουν private δομητές ή τις περισσότερες μεθόδους της κλάσης final ή private. Φυσικά αυτό λειτουργεί μόνο για κλάσεις. Αν αποφασίσουμε να χρησιμοποιήσουμε διεπαφές, δεν μπορούμε να απαγορεύσουμε «ξένες» υλοποιήσεις σε επίπεδο εικονικής μηχανής. Το μόνο που μπορούμε να κάνουμε είναι να ζητήσουμε από τους χρήστες να μη το κάνουν. Κάτι που δεν μας εξασφαλίζει εννοείται την μη υλοποίηση του. Συνεπώς, εκτός και αν υπάρχουν καταστροφικές συνέπειες, είναι προτιμότερο να αποφεύγονται οι υποκλάσεις σε επίπεδο εικονικής μηχανής χρησιμοποιώντας final κλάσεις.

Μην βάζετε τους Setters εκεί που δεν ανήκουν

Μην τοποθετείτε setters σε πραγματικά API. Με τον όρο πραγματικό API εννοείται η διεπαφή που πρέπει να υλοποιήσει κάτι. Εάν οι setters χρειάζονται, συνήθως δεν συμβαίνει αυτό, τότε ανήκουν μόνο στις βασικές κλάσεις. Ενδεικτικό παράδειγμα είναι το javax.swing.Action που παραβιάζει αυτόν τον κανόνα. Η μέθοδος setEnabled(boolean) δεν ανήκει στην διεπαφή Action. Πιθανώς να μην έπρεπε καν να υπήρχε αλλά έστω κι έτσι θα έπρεπε να είναι μία ιδιωτική μέθοδος στην AbstractAction, μια κλάση που υποστηρίζει την απλή υλοποίηση της διεπαφής. Η μέθοδος boolean isEnabled () είναι το API! Είναι η μέθοδος που πρέπει να κληθεί από όλους. Το ερώτημα που τίθεται είναι ποιος θα καλέσει την setEnabled (boolean).

Στην καλύτερη περίπτωση μόνο ο δημιουργός της Action, επειδή για όλους τους άλλους αποτελεί μια λεπτομέρεια υλοποίησης. Αν τυχαίνει η Action να είναι μια καθολική δράση για την οποία ευθύνεται ο χειριστής (listener), η setEnabled (boolean) είναι μία ιδιωτική final μέθοδος στην βασική κλάση όπως την AbstractAction. Εντούτοις, η Action.setEnabled είναι λανθασμένη για τους εξής λόγους:

• Υποτίθεται ότι άγνωστοι εξωτερικοί χρήστες θα καλέσουν αυτή την μέθοδο. Αυτό όμως είναι σίγουρο ότι δεν θα συμβεί.
• Δεν έχει κανένα νόημα για τις ευαίσθητων περιεχομένων ενέργειες, αν και η Action του API γενικά υστερεί εδώ.
• Δεν έχει νόημα να έχουμε μια μέθοδο setEnabled για μια ενέργεια που είναι πάντα ενεργοποιημένη. Οι ενέργειες αυτές απλώς επιθυμούν να επιστρέψουν μια τιμή true από την isEnabled ανεξάρτητα από οτιδήποτε που συμβαίνει γύρω τους.
• Δεν έχει νόημα να έχουμε μια ενέργεια της οποίας η ενεργοποίηση υπολογίζεται από μία isEnabled κατ'εντολή.

Αν θέλουμε να υπολογίζουμε την κατάσταση των ενεργειών μας όταν μας ζητείτε, τότε η setter δεν είναι η σωστή μέθοδος για χρήση. Αν θέλουμε να υλοποιήσουμε μία "pull" στρατηγική, τότε θα θέλουμε να παρέχουμε ένα αποτέλεσμα χωρίς όμως να υποκρινόμαστε ότι γνωρίζουμε τα πάντα για την καθολική κατάσταση της ενέργειας. Αντιθέτως, θέλουμε το σύστημα να μας ρωτάει για την κατάσταση του όταν η τιμή της isEnabled είναι απαραίτητη.

Αυτή η συμβουλή δεν έχει σκοπό να αμαυρώσει κάθε setter σε ένα API. Μερικές φορές είναι χρήσιμοι. Συμπερασματικά ο κανόνας εδώ είναι: Μεγάλη προσοχή για του περιττούς setters στα API.

Επιτρέπεται την πρόσβαση μόνο από «φίλο» κώδικα

Κατά την προσπάθεια να προστατευτεί η έκθεση των περισσοτέρων σε ένα API, ένας άλλος κανόνας είναι να δίνεται πρόσβαση σε ορισμένες λειτουργίες μόνο σε «φίλο» κώδικα. Λειτουργίες όπως η ικανότητα να ορίζεις μια κλάση ή να καλείς μία συγκεκριμμένη μέθοδο. Η Java από κατασκευής, περιορίζει τους «φίλους» μιας κλάσης σε αυτούς που ανήκουν στο ίδιο πακέτο. Αν θέλουμε να μοιραζόμαστε την λειτουργικότητα μεταξύ των κλάσεων στο ίδιο πακέτο, μπορούμε να χρησιμοποιούμαι τον ιδιωτικό τροποποιήτη πακέτου (package private modifier) όταν ορίζουμε έναν δομητή, ένα πεδίο ή μια μέθοδο. Έτσι θα διατηρηθεί η πρόσβαση μόνο σε «φίλους».

Μερικές φορές είναι χρήσιμο να επεκτείνουμε το πλήθος των φίλων μας σε περισσότερες κλάσεις. Για παράδειγμα, μπορεί να θέλουμε να ορίσουμε δύο πακέτα, ένα μόνο για το Αρι και το άλλο για την υλοποίηση του. Αν ακολουθήσουμε αυτή την προσέγγιση ο ακόλουθος κώδικα θα μας φανεί χρήσιμος. Έστω ότι έχουμε μία κλάση ως εξής:

```
public final class Item {
private int value;
private ChangeListener listener;
static {
Accessor.setDefault(new AccessorImpl());
}
/** Μόνο φίλοι μπορούν να δημιουργήσουν στιγμιότυπα. */
Item() {
}
/** Όλοι μπορούν να αλλάξουν την τιμή.
*/
public void setValue(int newValue) {
value = newValue;
ChangeListener l = listener;
if (l != null) {
l.stateChanged(new ChangeEvent(this));
}
}
/** Όλοι μπορούν να πάρουν την τιμή.
*/
public int getValue() {
return value;
}
/** Μόνο φίλοι μπορούν να "ακούσουν" τις αλλαγές.
*/
void addChangeListener(ChangeListener l) {
assert listener == null;
listener = l;
}
}
```

Αυτή η κλάση είναι μέρος του API αλλά δεν μπορεί να είναι άμεση ή να λειτουργήσει από εξωτερικές φιλικές κλάσεις που βρίσκονται στο πακέτο του API ή σε άλλα πακέτα. Μπορούμε όμως να ορίσουμε έναν Accessor στο πακέτο που δεν έχει το API.

```
public abstract class Accessor {
private static volatile Accessor DEFAULT;
public static Accessor getDefault() {
Accessor a = DEFAULT;
if (a != null) {
return a;
}
try {
Class.forName(
Item.class.getName(), true, Item.class.getClassLoader()
);
} catch (Exception ex) {
ex.printStackTrace();
}
return DEFAULT;
```

```
}
public static void setDefault(Accessor accessor) {
if (DEFAULT != null) {
throw new IllegalStateException();
}
DEFAULT = accessor;
}
public Accessor() {
}
protected abstract Item newItem();
protected abstract void addChangeListener(Item item, ChangeListener l);
}
```

Η Accessor έχει αφηρημένες μεθόδους με πρόσβαση σε όλη την φιλική λειτουργικότητα της κλάσης Item, με στατικό πεδίο για να παίρνει το στοιχείο του. Το βασικό κόλπο εδώ είναι να υλοποιείται η Accessor με μη δημόσια κλάση στο πακέτο API.

```
final class AccessorImpl extends Accessor {
protected Item newItem() {
return new Item();
}
protected void addChangeListener(Item item, ChangeListener l) {
item.addChangeListener(l);
}
}
```

Καταχωρούμε την Accessor σαν εξ κατασκευής στοιχείο που ασχολείται με το api.Item για πρώτη φορά. Αυτό γίνεται προσθέτοντας έναν στατικό αρχικοποιητή στην κλάση Item.

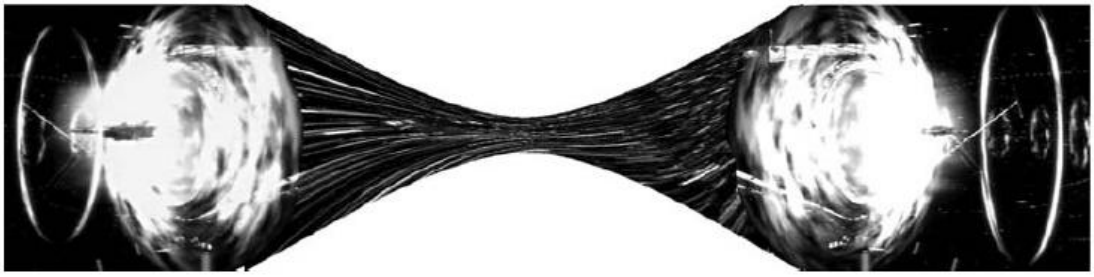
```
static {
Accessor.setDefault(new AccessorImpl());
}
```

Τώρα η φίλη κλάση μπορεί να χρησιμοποιεί την Accessor για να έχει πρόσβαση στην κρυφή λειτουργικότητα του πακέτου

```
Item item = Accessor.getDefault().newItem();
assertNotNull("Some item is really created", item);
Accessor.getDefault().addChangeListener(item, this);
```

Μελλοντικές εκδόσεις της Java μπορεί να παρέχουν έναν εναλλακτικό τροποποιητή πρόσβασης που θα επιτρέπει κοινή πρόσβαση σε πολλαπλά πακέτα που θα μεταγλωττίζονται μαζί. Ωστόσο, δεν υπάρχει λόγος αναμονής. Ο νέος τροποποιητής πρόσβασης θα απαιτήσει πιθανόν αλλαγές στην εικονική μηχανή, η οποία θα αποτρέψει την πιθανή χρήση του κώδικα από το να λειτουργεί σε παλαιότερες εκδόσεις της Java, συμπεριλαμβανομένης της Java 6. Το μοτίβο της εν λόγω πρόσβασης με τις απαιτούμενες αντικαταστάσεις θα μπορεί να τρέχει σε οποιαδήποτε έκδοση της JVM, επιτυγχάνοντας παράλληλα σχεδόν το ίδιο αποτέλεσμα. Το σχεδόν μπαίνει στην πρόταση επειδή υπάρχει κάτι που δεν μπορεί να αντιστοιχιστεί απευθείας με αυτό το πρότυπο: ο νέος τροποποιητής πρόσβασης επιτρέπει τον κώδικα ενός φιλικού

πακέτου να επεκταθεί σε κλάσεις του πακέτου API, το οποίο τα άλλα πακέτα δεν θα μπορούν να δουν. Εντούτοις, αυτό το είδος του περιορισμού είναι εύκολο να εξαλειφθεί: αντί να χρησιμοποιούνται υποκλάσεις, χρησιμοποιούνται μέθοδοι ανάθεσης. Ο Tulach αναφέρεται σε αυτό το είδος των σχεδιαστικών προτύπων των API ως τηλεδιεπαφή, όπως απεικονίζονται στο παρακάτω σχήμα. Παρόμοιο με ένα διαστημόπλοιο, που ταξιδεύει στο διάστημα από μία ανοικτή πύλη προς μία άλλη και είναι αόρατο από τον εκτός αυτού χώρο. Το εν λόγω πρότυπο επιτρέπει σε δύο ξεχωριστά μέρη να επικοινωνούν έτσι ώστε κανένας άλλος να μην μπορεί να παρατηρεί την διάδραση αυτή κι ας είναι γνωστό ότι υπάρχει.



Τηλεδιεπαφή Σχήμα_I

Η «φιλική» accessor αποτελεί παράδειγμα ενός τέτοιου προτύπου σχεδιασμού API. Οι δύο πλευρές πρέπει να επικοινωνούν ενώ κανένα κομμάτι του εξωτερικού κώδικα δεν πρέπει να παρακολουθεί αυτήν την αλληλεπίδραση. Είναι σαν να διεξάγεται στο «διάστημα».

Δώστε στον δημιουργό του αντικειμένου περισσότερα δικαιώματα.

Δεν πρέπει να έχουν όλοι οι χρήστες ενός αντικειμένου πρόσβαση σε όλη την λειτουργικότητα. «Κάποιος κώδικας είναι πιο ίσος από άλλον». Υπάρχουν επίσης περιπτώσεις που ένα κομμάτι του κώδικα χρειάζεται περισσότερα πλεονεκτήματα από κάποιο άλλο κομμάτι κώδικα.

Τα αντικείμενα ταξιδεύουν μεταξύ διαφόρων τμημάτων του συστήματος και για να εξασφαλιστεί βασική σταθερότητα των εσωτερικών καταστάσεων δεν είναι σοφό να αφήνουμε κάθε τμήμα του κώδικα να τροποποιεί το εκάστοτε αντικείμενο. Σ'αυτήν την περίπτωση, σύμμαχοι αποτελούν οι τροποποιητές πρόσβασης. Αν δεν θέλουμε να είναι κάτι ορατό σε εξωτερικό κώδικα, τότε δεν πρέπει να κάνουμε την λειτουργικότητα αυτή δημόσια ή προστατευόμενη. Ωστόσο, αυτό στρέφει τον κώδικα στον διαχειριστή, καθιστώντας τον οποιοδήποτε άλλον, απλό χρήστη. Όσον αφορά τα οικιακά συστήματα δεν υπάρχει πρόβλημα αλλά στον σχεδιασμό των API δεν είναι σωστό επειδή δεν θέλουμε το API να είναι διαχειριστής. Αντιθέτως, θέλουμε το API να είναι μια βιβλιοθήκη που θα εξυπηρετεί τους διαχειριστές όπως και τους απλούς χρήστες.

Μια κοινή προσπάθεια για το διαχωρισμό των δικαιωμάτων του δημιουργού ενός API είναι η δημιουργία μιας διεπαφής API ή μιας κλάσης μαζί με μία υποστηρικτική κλάση που την επεκτείνει. Η υποστηρικτική αυτή κλάση παρέχει επιπρόσθετους ελέγχους, κυρίως για τον δημιουργό. Όλες οι άλλες μέθοδοι του API υποτίθεται ότι χρησιμοποιούν την βασική διεπαφή της κλάσης καθώς οι υλοποιητές χρησιμοποιούν την υποστηρικτική κλάση για να υλοποιούν εύκολα και να έχουν πρόσβαση σε άλλες μεθόδους που δεν είναι διαθέσιμες σε μη προνομιακούς χρήστες. Αυτή ήταν η ιδέα στο javax.swing.text.Document και στην υποστηριζόμενη της κλάση javax.swing.text.AbstractDocument. Σχετικά αποδοτικό, αφού η μετατροπή της AbstractDocument σε υποκλάση είναι απλούστερη από την υλοποίηση όλων των μεθόδων στην διεπαφή Document. Επιπλέον η AbstractDocument περιλαμβάνει μεθόδους με πρόσβαση μόνο στους υλοποιητές όπως την ιδιωτική writeLock και την writeUnlock. Κάτι που είναι λογικό αλλά περιορίζει την πρόσβαση μόνο στις υποκλάσεις. Επομένως είναι φυσικό να βρεθεί κώδικας όπως ο ακόλουθος που εκθέτει την μέθοδο «creator only» σε όλο τον κώδικα του ίδιου πακέτου, το οποίο πολύ συχνά είναι και αναγκαίο.

```
public class MyDocument extends AbstractDocument {
public MyDocument() {super(new StringContent());}
final void writeLockAccess() {
writeLock();
}
}
```

Δεν ενδείκνυται να υποχρεώνεις τους χρήστες των API της AbstractDocument να αντιγράφουν μεθόδους όπως εδώ. Ένα καλύτερα σχεδιασμένο API θα εξάλειφε αυτή την κατάσταση. Ωστόσο, αυτό είναι ένα στιλιστικό θέμα και κατ'επέκταση δεν αποτελεί το κύριο πρόβλημα.

Το μεγαλύτερο πρόβλημα με AbstractDocument είναι ότι περιλαμβάνει ένα πλήθος άλλων χρήσιμων μεθόδων που παρουσιάζουν μεγάλο ενδιαφέρον για τους μη προνομιούχους χρήστες. Αυτές περιλαμβάνουν τις δημόσια προσβάσιμες readLock, readUnlock, την getListeners ή μεθόδους όπως η replace η οποία προστέθηκε αργότερα με αποτέλεσμα να μην εφαρμόζει τέλεια στην διεπαφή Document. Συμπερασματικά, υπάρχουν αρκετές ενδιαφέρουσες μεθόδους που οι χρήστες καλούν στην javax.swing.text API. Οι χρήστες παίρνουν ένα στιγμιότυπο του document, το μετατρέπουν σε AbstractDocument και καλούν αυτές τις μεθόδους. Σχετικά με την ασφάλεια, αν ήταν εύκολο για του μη διαχειριστές να εισάγονται σε διακομιστές όπως να καλούν μη προνομιακές μεθόδους της AbstractDocument, όλοι οι διακομιστές στο διαδίκτυο θα ήταν ήδη παραβιασμένοι. Αυτή είναι και η αιτία που το συγκεκριμμένο μοτίβο της διεπαφής API, με την υποστηρικτική κλάση, δεν επιλύει το πρόβλημα της προνομιακής πρόσβασης.

Υπάρχει ένα μέλος σε κάθε κλάση της Java, όπως και σε άλλες αντικειμενοστραφείς γλώσσες που μπορεί να κληθεί μόνο μία φορά από τον δημιουργό του αντικειμένου. Ο δομητής. Αυτό μας δίνει την βάση για ένα άλλο απλό διαχωρισμό της προνομιακής κατάστασης: οτιδήποτε μπορεί να ελέγχθει στον προνομιακό κώδικα πρέπει να συμπεριληφθεί, όταν δημιουργείται το αντικείμενο._Αυτό επιλύει σημαντικά προβλήματα. Ο προνομιακός κώδικας μπορεί να εισχωρεί μέσα στο αντικείμενο κατά την διάρκεια της δημιουργίας του. Επιπλέον, εν μέρη επιλύει το πρόβλημα των συνδυαστικών API's. Όταν οι χρήστες των API έχουν κατ'αναφορά ένα στιγμιότυπο ενός αντικειμένου, δεν συνηθίζεται να διαβάζουν την τεκμηρίωση του δομητή. Εξάλλου συχνά δεν τους παρέχεται πλέον στα σύγχρονα περιβάλλοντα ανάπτυξης. Επιπροσθέτως, οι μέθοδοι αναδόμησης και οι δομητές είναι ευκολονόητα έργα. Αυτή η λύση

είναι φυσική και απλή επειδή δεν εφευρίσκει κάτι νέο. Εν ολίγοις, αυτή είναι ίσως η πιο ενδεδειγμένη λύση.

Ωστόσο, τίθεται το θέμα της εξέλιξης και της περίπτωσης να θέλει κάποιος να προσθέσει νέες μεθόδους στον προνομιακό κώδικα. Σε τέτοιες περιπτώσεις, θα πρέπει να προστεθεί ένας δομητής ή μία μέθοδος αναδόμησης που να δέχονται τις πρόσθετες παραμέτρους. Το πιο περίπλοκο θέμα εδώ είναι ότι το ενδεχόμενο πλήθος των παραμέτρων σε αυτές τις μεθόδους μπορεί να είναι τεράστιο. Μεγάλες λίστες παραμέτρων, κυρίως ίδιου τύπου δεν είναι ποτέ ευχάριστες. Είναι εύκολο να δημιουργηθεί ένας νέος τύπος και να παραληφθεί μία παράμετρος αλλά αυτό δεν είναι πρόβλημα στην αρχή. Ωστόσο, όταν εμπλέκονται τα API η κατάσταση περιπλέκεται. Είναι επιθυμητό να υπάρχει μια έτοιμη λύση για αυτές τις περιπτώσεις, όπως η αναμεμειγμένη προσέγγιση της μεθόδου αναδόμησης και της setter προσέγγισης.

Η λύση είναι η εξής:

```
public static Executor create(Configuration config) {
return new Fair(config);
}
public static final class Configuration {
boolean fair;
int maxWaiters = -1;
public void setFair(boolean fair) {
this.fair = fair;
}
public void setMaxWaiters(int max) {
this.maxWaiters = max;
}
}
```

Με την προσέγγιση αυτή, μπορούμε να σταματήσουμε την προσθήκη νέων μεθόδων αναδόμησης και στην περίπτωση που απαιτείται, να προσθέσουμε setters στην κλάση Configuration. Η πρόσθεση τέτοιων μεθόδων είναι ασφαλής, αφού η κλάση είναι final και η μόνη χρήση της είναι σαν παράμετρος στην μέθοδο αναδόμησης create. Η χρησιμότητα της περιορίζεται εξωτερικά καθώς σκοπίμως δεν υπάρχουν getters. Αυτός είναι ο λόγος που η πρόσθεση νέων μεθόδων μπορεί να επηρεάσει μόνο εσωτερικά την μέθοδο αναδόμησης η οποία είναι υπό τον έλεγχο του API. Υπάρχει όμως και η δυνατότητα να επεκτείνουμε το εσωτερικό της έτσι ώστε να χειριζόμαστε τις επιπλέον παραλλαγές στην διαμόρφωση της εισόδου.

Επιπλέον, η εισαγωγή της κλάσης Configuration, επιλύει το τελευταίο σε εκκρεμότητα θέμα. Αυτό της ικανότητας του προνομιακού κώδικα να εκτελεί τις προνομιακές του λειτουργίες μετά την πρόσβαση του μη προνομιακού κώδικα στο αντικείμενο. Αυτό ήταν αδύνατο με το μοτίβο του setReadOnly καθώς και με την μέθοδο αναδόμησης. Έτσι, ο προνομιακός κώδικας μπορεί να διατηρεί την αναφορά του με το στιγμιότυπο της Configuration. Η μετατροπή δεν βοηθάει επειδή αυτά είναι δύο διαφορετικά αντικείμενα. Η Configuration δρα με έναν μυστικό τρόπο που δεν μπορεί να αποκαλυφθεί μέσω του API. Επομένως αυτή η λύση είναι απόλυτα ασφαλής.

Παρόλο που η Java δεν έχει κάποιον ειδικό τροποποιητή πρόσβασης για τα αντικείμενα, είναι εφικτό να δημιουργηθεί ένα API που θα εξασφαλίζει αυτήν την σύμβαση. Υπάρχει μία σειρά λύσεων με πιο αποδοτική την εξής: έχοντας δύο κλάσεις, η μία δίνει προνομιακή πρόσβαση στον δημιουργό και η άλλη παρέχει δημόσια το API για κοινή χρήση.

Μην εκθέτετε το βάθος της κληρονομικότητας

Όταν γράφετε ένα API, το μέγεθος και η συμπεριφορά της μεταγωγής είναι άγνωστο στον συγγραφέα του κώδικα. Αυτό μπορεί να θεωρηθεί ως πλεονέκτημα, αυξάνοντας τον αριθμό των τρόπων που ένα κομμάτι του κώδικα μπορεί να επαναχρησιμοποιηθεί. Ωστόσο, μερικές εκδόσεις αργότερα, αν δεν είναι γνωστοί οι πραγματικοί συμμετέχοντες της μεταγωγής μπορεί να αποβεί καταστροφικό. Επιτρέποντας απλά ένα άγνωστο πλήθος χρηστών να συμμετέχουν στον κώδικα, μπορεί να καταστήσει την μεταγωγή μη συντηρήσιμη.

Η έκθεση του βάθους της ιεραρχίας των κλάσεων από μόνη της δεν αρκεί για να βελτιώσει την χρηστικότητα του API. Αποτελεί κοινή παραδοχή ότι η κλάση Human είναι υποκλάση της Mammal αλλά πόσο πολύ μπορεί να χρησιμοποιηθεί αυτό ως γεγονός στην σχεδίαση των API. Ο ισχυρισμός ότι το JButton είναι υποκλάση του AbstractButton το οποίο είναι υποκλάση του JComponent που με τη σειρά του είναι υποκλάση του Container κι αυτό υποκλάση του Component, φυσικά και δεν μας βοηθάει. Οι υποκλάσεις εδώ είναι απλά μια λεπτομέρεια υλοποίησης. Είναι μια μεταγωγική κατάσταση που μας επιτρέπει να γράφουμε κώδικα με απλό τρόπο. Ωστόσο, δεν σχετίζεται με την έννοια του API. Αν σχετίζεται, τότε θα ήταν φυσιολογικό να χρησιμοποιηθεί το Jbutton ως Container. Δηλαδή, είναι δυνατό να του προστεθούν επιπλέον συστατικά. Θα ήταν πιθανό να λειτουργήσουμε έτσι αλλά δεν αποτελεί την προορισμένη εκ του API χρήση του JButton.

Σε αυτό το σημείο αναδεικνύεται ένα κοινό ελάττωμα των API στις αντικειμενοστραφείς γλώσσες. Εάν η υλοποίηση ωφελείται από την υποκλάση μιας κλάσης, δεν αναμένεται ότι όλοι οι άλλοι, το API ολόκληρο και οι χρήστες του, θα ωφεληθούν από αυτήν την επέκταση. Κάτι τέτοιο δεν πρόκειται να γίνει αν η υποκλάση υπάρχει για να υλοποιεί απλά μια κατάσταση μεταγωγής. Ο κανόνας εδώ είναι ο εξής: αντί για μεγάλες και ιεραρχικές κλάσεις, ορίστε το πραγματικό API της εφαρμογής και αφήστε τους χρήστες να το υλοποιήσουν. Εάν κάτι είναι υποκλάση ή υποδιεπαφή, μπορεί να κληρονομηθεί σε κάποιο σημείο της βασικής κλάσης ή της διεπαφής.

Πρέπει πάντα να υπάρχει η παρόμοια ανησυχία του Frame που επεκτείνεται απο το Component. Δίνοντας όλους τους κανόνες των API, αυτή η σχέση σημαίνει ότι το Frame μπορεί να χρησιμοποιείται όπου χρησιμοποιείται και το Component, ακόμα κι αν αυτό δεν αληθεύει πάντα. Το γεγονός ότι το Frame κληρονομείται από το Component είναι μια λεπτομέρεια υλοποίησης, μια μεταγωγική κατάσταση για να επαναχρησιμοποιείται ο κώδικας του Component. Αυτό όμως δεν σχετίζεται με τα πραγματικά API. Αυτός ο τύπος της αντικειμενοστραφούς επαναχρησιμοποίησης, που καλύτερα κατά τον Tulach ονομάζεται μη χρησιμοποίηση, είναι κοινός κυρίως σε περιπτώσεις βαθιάς κληρονομικότητας κλάσεων ή διεπαφών. Αυτή είναι και η αιτία που όταν μια ιεραρχία κληρονομικότητας είναι βαθύτερη από δύο επίπεδα είναι χρήσιμο να αναρωτιόμαστε αν αυτό γίνεται για το API ή για την επαναχρησιμοποίηση του κώδικα. Αν γίνεται για το δεύτερο, τότε θα πρέπει να ξαναγράψουμε το API πιο συνεκτικά.

Δοκιμές

Ο σχεδιασμός του λογισμικού διαρκώς αλλάζει. Οι προγραμματιστές αναπτύσσουν εφαρμογές τελείως διαφορετικά από την αρχή της δεκαετίας και αυτό οφείλεται σε μεγάλο βαθμό στην ύπαρξη βιβλιοθηκών ανοικτού κώδικα. Η διάθεση αυτών, στρέφει τους προγραμματιστές σε μεταγλωττιστές υφιστάμενης λειτουργικότητας.

Μία άλλη σημαντική και σχετική αλλαγή είναι το πλήθος των πρότυπων εφαρμογών (modular applications). Οι πρότυπες εφαρμογές μπορούν να παραδοθούν με αποδεκτή ενσωμάτωση των επιμέρους συστατικών τους, καθώς συνεχίζουν να παρέχουν την εμπειρία των χρηστών σε μία εκ των σημαντικότερων αλλαγών της δεκαετίας, το αυξανόμενο πλήθος των δοκιμών. Έχει γίνει πλέον κοινός τόπος η σημασία των δοκιμών και πλήθος αυτόματων δοκιμών έχουν παρουσιαστεί, σε σημείο πλέον να θεωρείται για τους προγραμματιστές εθισμός.

Η βασική στρατηγική για τις περισσότερες δοκιμές και κυρίως τον έλεγχο μονάδων, γίνεται με ψεύτικα αντικείμενα (mock objects). Το ψεύτικο αντικείμενο είναι κάτι παραπάνω από μία ψεύτικη υλοποίηση μιας διεπαφής που δημιουργείτε για λόγους δοκιμών. Για παράδειγμα, όταν μία πραγματική εφαρμογή πρόκειται να συνδεθεί με μια βάση δεδομένων μέσω της υλοποίησης της διεπαφής Connection, οι μονάδες δοκιμής χρησιμοποιούν μια ψεύτικη υλοποίηση η οποία δεν κάνει στην πραγματικότητα την σύνδεση αλλά προσομοιώνει αξιόπιστα την συμπεριφορά της πραγματικής εφαρμογής. Με τη χρήση αυτού του είδους των υλοποιήσεων, μπορούμε να επικεντρωθούμε στην λογική της πραγματικής εφαρμογής και να εξαλείψουμε τυχαίες επιδράσεις στην σύνδεση της βάσης. Αυτό επιπλέον απλοποιεί την δοκιμή, αφού η εκτέλεση δεν απαιτεί πραγματική βάση δεδομένων αλλά απλά μια προσομοίωση στην μνήμη.

Από τη σκοπιά ενός συγγραφέα API, η σημαντικότερη πτυχή των API είναι να προσφέρει ένα τρόπο για τη δημιουργία ψεύτικων αντικειμένων για τις σημαντικότερες διεπαφές. Είτε θα πρέπει να είναι δυνατόν να γραφεί μια υλοποίηση αυτών των διεπαφών, είτε το API οφείλει να παρέχει κάποιες δικές τους υλοποιήσεις. Φυσικά, αυτό ισχύει μόνο για τα API, όπου η υλοποίηση των ψεύτικων αντικειμένων κάνει την διαφορά. Για παράδειγμα, κανείς δεν θα γράψει ένα ψεύτικο αντικείμενο για String, αφού το String αποτελεί καθαρή και απλή βιβλιοθήκη, κάνει τη δουλειά του και την κάνει σωστά. Ο άλλος λόγος είναι ότι η εκτέλεση όλων των String είναι αυτοδύναμες. Δεν απαιτούν οποιοδήποτε περιβάλλον και συνεπώς είναι κατάλληλες για να δοκιμάζονται μεμονωμένα χωρίς την παρουσία ψεύτικων αντικειμένων.

Όταν τα API’s εκθέτουν μόνο όσα χρειάζεται, μπορεί να υπάρξει πρόβλημα με την φάση της δοκιμής τους. Μερικές φορές οι χρήστες των API δεν χρειάζονται τις υποκλάσεις. Κατ’επάκταση, οι υποκλάσεις ενδέχεται να αφανιστούν. Αυτό σημαίνει ότι δεν θα υπάρχει η δυνατότητα δημιουργίας ψεύτικων αντικειμένων στα API, γεγονός που περιορίζει σημαντικά τις δοκιμές των εφαρμογών που κάνουν χρήση των API, με συνέπεια τα API να μην παρουσιάζουν ενδιαφέρων για τον χρήστη.

Μερικές φορές αυτός μπορεί να είναι ο λόγος για τον οποίο οι άνθρωποι προτιμούν να χρησιμοποιούν τις διασυνδέσεις ενός API. Ωστόσο, οι διασυνδέσεις ενδέχεται να αντιμετωπίσουν προβλήματα εξέλιξης, επειδή απαγορεύετε η προσθήκη μεθόδων σε μελλοντικές εκδόσεις τους. Έτσι, η δυνατότητα δημιουργίας ψεύτικων αντικειμένων είναι σημαντική, αλλά όχι πιο σημαντική από το ίδιο το API που μπορεί να ανακάμψει από τα λάθη του και να αυξήσει την ικανοποίηση των χρηστών του.

Η λύση σε όλα αυτά είναι να παρασχεθεί όχι μόνο ένα κανονικό API, αλλά και ένα API μόνο για δοκιμές. Τα δοκιμαστικά API είναι μια επέκταση των κανονικών. Δεν προορίζονται να χρησιμοποιηθούν σε πραγματικά προϊόντα και εφαρμογές, αλλά μόνο για την εκτέλεση δοκιμών. Τα δοκιμαστικά API μπορεί να περιέχουν τις δικές τους εφαρμογές ή τουλάχιστον το μέσο για τη δημιουργία ψεύτικων αντικειμένων για τις κλάσεις του API. Μπορεί επίσης να περιέχει άλλα χρήσιμα εργαλεία που εκθέτουν τις εσωτερικές λεπτομέρειες του API. Αυτό είναι εφικτό διότι οι δοκιμές συνήθως βρίσκονται στο ίδιο πακέτο με τις κλάσεις του API. Οι δοκιμές μπορούν να χρησιμοποιήσουν το πακέτο με τις ιδιωτικές μεθόδους που δεν είναι διαθέσιμες για τους χρήστες του κανονικού API.

Εδώ τίθεται το ερώτημα πως μας βοηθάει ο διαχωρισμός ενός API από ένα δοκιμαστικό API. Γιατί να μην μπορούν οι χρήστες απλά να συμπεριλάβουν τα δοκιμαστικά jar στην διαδρομή και μετά να τα χρησιμοποιήσουν στις κανονικές τους εφαρμογές. Παρόλο που δεν υπάρχει τρόπος να εμποδιστούν για κάτι τέτοιο, κάποιες απλές επισημάνσεις μπορεί να τους προειδοποιήσουν για να μη το κάνουν. Πρώτον, μπορούμε να τονίσουμε ότι η σταθερότητα ενος πραγματικού API είναι σημαντικότερη της σταθερότητας των δοκιμαστικών API.

Δεύτερον, ότι τα δοκιμαστικά API είναι πιθανό να είναι πιο «κρυφά». Για παράδειγμα, αν υπάρχει κάποιο διαφορετικό jar είναι πιθανό ότι θα τεκμηριωθεί σε ένα διαφορετικό αρχείο. Οι προγραμματιστές που χρησιμοποιούν μόνο την βιβλιοθήκη του API μπορεί ούτε καν να γνωρίζουν για το δοκιμαστικό API. Δεν πρέπει να αποπροσανατολιστούν από την παρουσία μεθόδων στο API που έχουν ασαφή σκοπό. Ένας τρόπος για να μην χρησιμοποιήσουν οι προγραμματιστές το δοκιμαστικό API σε μια πραγματική εφαρμογή είναι ο εξής: ο δοκιμαστικός κώδικας συνήθως εκτελείτε με ενεργοποιημένα σήματα, ενώ ο παραγωγικός κώδικας με απενεργοποιημένα. Άρα μπορούμε να κατευθύνουμε τις μεθόδους μας στο δοκιμαστικό API με κριτήριο ποια σήματα είναι ενεργά και σε αντίθετη περίπτωση να προκαλούμε μία εξαίρεση. Ένας κώδικας με τον οποίο μπορεί να γίνει αυτό είναι ο εξής:

```
boolean assertionsOn = false;
assert assertionsOn = true;
if (assertionsOn) {
throw new IllegalStateException("Αυτή είναι μία δοκιμαστική μέθοδος!");
}
```

Παρόλο που είναι ασαφής και ακραίος ισχύει αλλά δεν είναι υποχρεωτικός να συμπεριληφθεί. Το σημαντικότερο εδώ είναι ότι οι δοκιμές γίνονται όλο και πιο αναγκαίες, όλο και πιο σημαντικές. Αν κάνουμε εφαρμογές χωρίς δοκιμές είναι σαν να παρέχουμε στους ανταγωνιστές μας την δουλειά μας μισοτελειωμένη, την ευκαιρία να την ολοκληρώσουν και να την καρπωθούν.

Συμβουλές

Το API πρέπει να είναι όμορφο

Ένα από τα βασικότερα χαρακτηριστικά της ομορφιάς ενός API είναι η καθολική απόρριψη των αποδοκιμασιών. Με τον όρο αποδοκιμασία (deprecation) στην πληροφορική εννοούμε την επισήμανση ενός συστατικού ως παρωχημένο για αποθαρρυνθεί η χρήση του. Ένα ποσοστό χρηστών προτιμούν να απομακρύνουν ένα στοιχείο από το API παρά να το αποδοκιμάζουν και πράγματι ένα API γεμάτο αποδοκιμασίες δεν είναι επιθυμητό. Ωστόσο, η αποδοκιμασία είναι μία πολύ μικρότερη εργασία που πρέπει να γίνει συγκριτικά με την δουλειά που απαιτείται για την μεταφορά του σε ένα καινούργιο API, όταν η παλαιά έκδοση του API αποσυρθεί. Το κόστος αυτό συνδέεται με το αποδοκιμαζόμενο στοιχείο που παραμένει και μας αποσπά καθώς διαβάζουμε την τεκμηρίωση του API. Αυτό όμως είναι προτιμότερο από την ανασφάλεια που προκύπτει όταν αφαιρούμε κάτι από ένα API. Στην πραγματικότητα, όπως η διάσπαση κάθε συμβατότητας, οι αφαιρέσεις από ένα API ανέρχονται σε τιμωρίες για αυτούς που χρησιμοποιούν την βιβλιοθήκη μας κι αυτό είναι το τελευταίο που θέλει ο υπεύθυνος πωλήσεων του API. Το χαρακτηριστικό αυτό και γενικότερα η ομορφιά ενος API, δεν είναι το σημαντικότερο αλλά απόκτα μεγάλη σημασία όταν ξεκινάει η διαδικασία πώλησης του.

Το API πρέπει να είναι σωστό

Μια άλλη συμβουλή που μπορεί να παραβιαστεί είναι ο μύθος ότι ένα καλό API, πρέπει πάντοτε να είναι σωστό. Ένα σωστό API είναι σίγουρα πιο χρήσιμο. Ένα API που είναι εύκολο στη χρήση του είναι επίσης καλύτερο από ένα που έχει πολλούς τρόπους για να χρησιμοποιηθεί καταχρηστικά απαιτώντας τον χρήστη του να ψάξει μέσω της τεκμηρίωσης για να βρει το σωστό τρόπο να το χρησιμοποιήσει. Εντούτοις, μπορεί να δημιουργηθούν πολλά προβλήματα θυσιάζοντας την ευκολία χρήσης για την ορθότητα. Η ευκολία στη χρήση είναι μερικές φορές πιο σημαντική από την ορθότητα. Η Java δεν έχει κάνει κάποιες βασικές μεθόδους ακόμα, όπως η απλοποίηση του να διαβάζεις και να γράφεις σε αρχεία. Η αιτία αυτής της δεκαετούς καθυστέρησης είναι ότι πρέπει να πιστοποιηθεί ότι είναι σωστές πριν κυκλοφορήσουν, επειδή ότι μπαίνει σε μία γλώσσα δεν ξαναβγαίνει. Η C# έχει κάνει ήδη πολλές διανομές με τέτοιες μεθόδους.

Το API πρέπει να είναι απλό

Ανάλογο πρόβλημα τίθεται, αν δούμε το API μόνο από την πλευρά ενός νεοεισερχόμενου στα API. Τα συναισθήματα και οι συμπεριφορές που απορρέουν από την πρώτη συνάντηση με τα API είναι καταλυτικά. Φυσικά και δεν θέλουμε να προκαλέσουμε απογοήτευση σχεδιάζοντας αναλόγως, χωρίς βέβαια να ξεχνάμε ότι η πλειονότητα των χρηστών θα είναι επαγγελματίες.

Συνοπτικά να αναφέρω ότι κατά τον σχεδιασμό των API πρέπει να είμαστε ιδιαίτερα προσεκτικοί για την απλότητα των πραγμάτων καθώς όταν προκύπτει πολυπλοκότητα, αυτή κάνει τα πράγματα ανέφικτα. Είναι επιθυμητό να κρύβουμε τις πολύπλοκες λειτουργίες. Αυτοί που τις χρειάζονται είναι ήδη σε θέση να τις ανακαλύψουν και όσοι από τους νεοεισερχόμενους επιθυμούν να τις βρουν, τους παρέχετε η δυνατότητα μέσω της τεκμηρίωσης. Κρύβοντας την

πολυπλοκότητα, αποφεύγουμε να τους τρομάξουμε στα πρώτα τους βήματα, την συνέχεια των οποίων θέλουμε κι εμείς για να ανακαλύψουν και να ασχοληθούν με τις βιβλιοθήκες μας.

Η βασική λύση εδώ, πέρα από το «κρύψιμο» της πολυπλοκότητας είναι να διαχωρίσουμε την απλή λειτουργικότητα από την πολύπλοκη βάζοντάς τες σε διαφορετικά πακέτα. Για παράδειγμα, το βασικό δικτυακό API μπορεί να είναι στο πακέτο net και οι κλάσεις για πιο προχωρημένες δικτυακές εργασίες στο net.stream υποπακέτο.

Στα εργαστήρια του NetBeans δέχονται πολλά e-mail που τους ζητάνε να κάνουν πιο εξειδικευμένα πράγματα και εμμέσως τους κατηγορούν ότι θυσιάζουν την χρησιμότητα των API για την απλότητα τους. Η επίσημη απάντηση τους, που δεν είναι ένα στεγνό όχι, είναι: είστε πολύ προχωρημένοι και δεν παρέχουμε υποστήριξη για τόσο εξειδικευμένες περιπτώσεις χρήσης. Παροτρύνοντας έτσι τους χρήστες να ψάξουν μόνοι τους την απάντηση στο ερώτημα τους. Να ανακαλύψουν, να σχεδιάσουν και να τεκμηριώσουν την λύση του. Όταν γίνει αυτό, τότε δέχονται να την κρίνουν, να την αποδεχτούν και να την ενσωματώσουν.

Το API πρέπει να είναι αποδοτικό

Ο κύριος λόγος για την διάρρηξη της συμβατότητας με παλαιότερες εκδόσεις είναι η απόδοση. Αν το API είναι κακοσχεδιασμένο, οι χρήστες του μπορεί να καλούν μεθόδους με μικρή ταχύτητα ή να δημιουργούν περιττά αντικείμενα. Η απόδοση είναι δύσκολο να διορθωθεί σε ένα API χωρίς την αλλαγή της υπάρχουσας συμπεριφοράς του. Από την άλλη πλευρά, κατά τον σχεδιασμό ενός API, δεν πρέπει να είμαστε υπεραισιόδοξοι ή να παραβιάζουμε τους κανόνες σχεδίασης για να επιτύχουμε υψηλή απόδοση. Στην περίπτωση αυτή, το κέρδος μπορεί να είναι εικαζόμενο ή προσωρινό. Αυτό συμβαίνει κυρίως στην java ή σε άλλες δυναμικά μεταγλωττιζόμενες γλώσσες. Το πρόγραμμα που προκύπτει μετά την μεταγλώττιση δεν είναι ακριβώς αυτό που βλέπει ο υπολογιστής. Χρειάζεται ακόμα να ερμηνευθεί η να μεταγλωττιστεί δυναμικά. Ο κώδικας μηχανής που παράγεται από τέτοιες μεταγλωττίσεις βελτιώνεται συνεχώς. Η χορήγηση μιας σειράς προσωρινών αντικειμένων φαίνεται ως απειλή για τον συλλέκτη σκουπιδιών όταν ξεκινάει η java. Ωστόσο, εξαιτίας της υλοποίησης του συλλέκτη σκουπιδιών, τα αντικείμενα αυτά παραμένουν στην «νέα γενιά», μέρος της μνήμης από το οποίο μπορούν να αποκτηθούν ανώδυνα. Αν υπάρχει βελτιστοποιημένη εφαρμογή ή API, τότε δεν θα χρειάζονται προσωρινά αντικείμενα. Φυσικά όσα λιγότερα αντικείμενα δημιουργούνται τόση λιγότερη δουλειά θα έχει ο συλλέκτης σκουπιδιών.

Άλλη περίπτωση που μπορεί η βελτίωση της απόδοσης να είναι πρώιμη, τουλάχιστον στην περίπτωση της java, είναι όταν αντικαθίστανται οι μέθοδοι get ή set με άμεσα εκτεθειμένα fields. Αυτή μπορεί να είναι μία αναγκαία τεχνική σε συστήματα όπως το .NET αλλά στην java είναι περιττή. Αποτυγχάνει να βελτιστοποιήσει την απόδοση επειδή ο δυναμικός μεταγλωττιστής είναι ικανός να εξαλείφει την επιβάρυνση (overhead) των get και set αντικαθιστώντας κλήσεις μεθόδων, με το σώμα του καλών (inlining), μετατρέποντας τα πάντα σε απλή πρόσβαση των fields. Η αντικατάσταση αυτή βελτιώνει την ταχύτητα και μειώνει την χρήση της μνήμης κατά τον χρόνο εκτέλεσης. Επιπλέον, αν εκθέσουμε άμεσα τα fields, περιορίζουμε την μελλοντική βελτιστοποίηση των API. Εν συντομία, πρέπει να είμαστε ανήσυχοι όσον αφορά τις πρώιμες βελτιστοποιήσεις επειδή αυτές στον κόσμο των API μπορεί να αποδειχθούν η ρίζα του κακού.

Το API πρέπει να είναι 100% συμβατό

Όταν τα προϊόντα σχεδιάζονται για νέα πρότυπα και λειτουργούν, διαβάζουν, γράφουν, παίζουν παλαιότερα πρότυπα, τότε λέμε ότι παρέχουν συμβατότητα προς τα πίσω (Backward compatibility). Όταν έχουμε backward συμβατότητα, τότε δεν ανησυχούμε για αναβαθμίσεις. Οι τελικοί χρήστες και οι μεταγλωττιστές επιθυμούν να στηρίζονται σε αυτή την συμβατότητα όταν χρησιμοποιούν πολλές βιβλιοθήκες μαζί. Η επίτευξη της είναι επιθυμητή και αναγκαία. Ωστόσο, με την τρέχουσα κατάσταση των έργων λογισμικού, η συμβατότητα με παλαιότερες εκδόσεις δεν είναι πάντα εφικτή. Ένας από τους κύριους στόχους είναι να επιτευχθεί σε κάθε βιβλιοθήκη και να περιγράφει τεχνικές που βοήθησαν στην πρόληψη σφαλμάτων. Ωστόσο, η επίτευξη της απόλυτης συμβατότητας είναι ζήτημα στάσης απέναντι στην σχεδίαση.

Η απόλυτη συμβατότητα είναι ακριβή σε κόστος και δύσκολο να επιτευχθεί. Στην πραγματικότητα, ακόμη κι ένα μικρό σφάλμα μπορεί να διαρρήξει την συμβατότητα. Για παράδειγμα, θα έχουμε μία μη συμβατή αλλαγή εάν το API μας αντέξει από μία εξαίρεση όταν πριν από αυτή προκλήθηκε μία NullPointerException. Στην περίπτωση αυτή, ο κώδικας που διερράγη από αυτήν την ανάρμοστη αλλαγή της συμπεριφοράς θα είναι περίεργος καθώς θα περιμένει μία NullPointerException να προκληθεί. Ωστόσο, θα ήταν ένας έγκυρος κώδικας για την java και μία έγκυρη χρήση για τα API. Η μη υποστήριξη όμως αυτής της χρήσης σε νεότερη έκδοση θα σήμαινε ότι δεν υπάρχει πια 100% συμβατότητα.

Η διόρθωση σφαλμάτων είναι σημαντική. Αποτελεί την αιτία της αναβάθμισης. Κάθε διόρθωση σφάλματος που δεν εκθέτει ένδειξη, σε χρόνο μεταγλώττισης, για το πότε πρέπει να εφαρμοστεί, απειλεί την προς τα πίσω συμβατότητα. Συμπερασματικά, κάθε διόρθωση σφαλμάτων είναι και μια απειλή για την απόλυτη συμβατότητα. Αν και οι κορυφαίοι σχεδιαστές των API απέδειξαν ότι μπορεί να επιτευχθεί, το ερώτημα είναι πόσο πρακτικά εφικτό είναι να διατηρηθεί και επιπλέον όχι με αφόρητο οικονομικό κόστος.

Ως αποτέλεσμα, στις περισσότερες περιπτώσεις, η επίτευξη της 99 τοις εκατό συμβατότητας είναι αρκετή. Είναι δύσκολο όμως να ορίσουμε τι εμπίπτει εντός του εύρους του 99 τοις εκατό συμβατότητας και τι αναφέρεται στο υπόλοιπο 1 τοις εκατό. Μόλις κάποιο είδος ασυμβατότητας επιτραπεί, ο χρήστης είναι πιθανό να το εκμεταλλευτεί και να αρχίσει την επιστροφή μηδενικών τιμών από μεθόδους ενώ ποτέ πριν δεν το έκανε, να προκαλεί εξαιρέσεις σε περιπτώσεις που παλιά δεν το έκανε, να διαβάζει το περιεχόμενο αρχείων διαμόρφωσης (configuration files) που παλιά δεν τα άνοιγε και να αλλάζει ονόματα κλάσεων ή να μετακινεί μεθόδους επηρεάζοντας έτσι την λειτουργικότητα τους. Για αυτές τις πιθανές περιπτώσεις και την δική μας ασφάλεια είναι καλύτερα να παραμένει ο στόχος της απόλυτης συμβατότητας.

Η απόλυτη συμβατότητα σημαίνει ότι αν υπάρχει κάτι στο API που μπορεί να χρησιμοποιηθεί καταχρηστικά, θα καταχραστεί. Ωστόσο, όταν γνωρίζεις τα πάντα για το API έχεις πρόσβαση σε όλες τις εκδόσεις του προτού αρχίσεις να εκμεταλλεύεσαι την ασυμβατότητα. Επιπλέον μπορείς να δεις όχι μόνο το API αλλά και τα εσωτερικά του στοιχεία. Αυτό είναι σύνηθες για χρήστες βιβλιοθηκών ανοικτού κώδικα αλλά δεν είναι σύνηθες και να διαβάζουν τον πηγαίο κώδικα για όλες τις εκδόσεις του API. Ο μόνος λόγος για να γίνει αυτό είναι να ψάξεις γραμμή προς γραμμή για διαφορές στις εκδόσεις των API ώστε να βρεις τυχόν προβλήματα συμβατότητας. Η περίπτωση αυτή όμως δεν είναι συχνή για τους χρήστες.

Το ακόλουθο παράδειγμα δείχνει μία αμυντική πρόταση εξέλιξης:

```
public class Arithmetica {
public int sumTwo(int one, int second) {
return one + second;
}
public int sumAll(int... numbers) {
if (numbers.length == 0) {
return 0;
}
int sum = numbers[0];
for (int i = 1; i < numbers.length; i++) {
sum = sumTwo(sum, numbers[i]);
}
return sum;
}
public int sumRange(int from, int to) {
if (Boolean.getBoolean("arithmetica.v2")) {
return sumRange2(from, to);
} else {
return sumRange1(from, to);
}
}
private int sumRange1(int from, int to) {
int len = to - from;
if (len < 0) {
len = -len;
from = to;
}
int[] array = new int[len + 1];
for (int i = 0; i <= len; i++) {
array[i] = from + i;
}
return sumAll(array);
}
private int sumRange2(int from, int to) {
return (from + to) * (Math.abs(to - from) + 1) / 2;
}
}
```

Στον παραπάνω κώδικα, η μέθοδος sumRange χρησιμοποιούσε έναν αργό αλγόριθμο, ο οποίος ήταν περισσότερο χρήσιμος για τον υπολογισμό παραγοντικών μετά την εσωτερική υποκλάση. Απλά ξαναγράφοντας την υλοποίηση για χρήση πιο γρήγορου αλγορίθμου διαρρηγνύουμε την προς τα πίσω συμβατότητα. Αν αποφασίσουμε όμως να κάνουμε την συμπεριφορά του συμβατή και πάλι, εκτός κι αν μας ζητηθεί ρητά το αντίθετο, προσθέτουμε την Boolean ιδιότητα getBoolean ("arithmetica.v2"), και αλλάζουμε την συμπεριφορά της μεθόδου. Αυτή είναι τουλάχιστον 99% συμβατότητα αλλά όχι 100%. Παραμένει ακόμα πιθανό να παρουσιαστούν διαφορές στις δυο εκδόσεις.

Η 99% συμβατότητα προς προηγούμενη έκδοση του API προσπαθεί να βρεί την ισορροπία μεταξύ των αναγκαίων διορθώσεων επί των προβλημάτων και της πρακτικής τους επίδρασης επί της ασυμβατότητας. Η αλλαγή του API μπορεί να θεωρηθεί αποδεκτή εάν η πιθανότητα να εκμεταλλευτεί κάποιος την ασυμβατότητα, καθώς γνωρίζει μόνο την παρούσα έκδοση, είναι σχεδόν μηδέν.

Το API πρέπει να είναι συμμετρικό

Είναι κοινώς αποδεκτό ότι όταν τα πράγματα είναι σε μορφή ζευγών θεωρούνται πιο κομψά, πιο συμμετρικά. Όπως επίσης ότι ασυνείδητα όλοι οι άνθρωποι έχουμε συνδέσει την ομορφιά και την κομψότητα με την αλήθεια. Μερικές φορές παρόμοιες κρίσεις εφαρμόζονται στην ορθότητα των API. Στην περίπτωση ύπαρξης δύο API με την ίδια ποιότητα, ευκολία χρήσης και συντήρησης, η επιλογή μεταξύ των δύο θα κριθεί από την συμμετρία τους. Η συμμετρία ενός API βοηθάει στην αποδοχή και κατανόηση του καθώς χρειάζεται να έχουμε κατά νου μόνο το ήμισυ του API, το άλλο ήμισυ μπορεί να συναχθεί από το πρώτο.

Ωστόσο, ένα API δεν είναι κακοσχεδιασμένο επειδή απλά δεν είναι συμμετρικό. Μπορούμε να έχουμε μεθόδους get χωρίς μεθόδους set ή set χωρίς get ή οι set μέθοδοι να είναι προστατευόμενες και οι get δημόσιες. Να σημειώσω όμως ότι δεν θεωρείται σημαντικό πρόβλημα η ύπαρξη ασυμμετρίας, κυρίως όταν ενισχύει κάποιον άλλο σχεδιαστικό κανόνα όπως της απλότητας.

Τρόποι ελέγχου της ποιότητας των API

Ικανότητα κατανόησης

Αυτοί που χρησιμοποιούν ένα API πρέπει να το καταλαβαίνουν. Αυτή είναι μία γενική δήλωση αλλά και η σημαντικότερη. Το API όπως έχω ήδη αναφέρει, είναι μία επικοινωνία μεταξύ του προγραμματιστή που γράφει το API και των προγραμματιστών που το υλοποιούν. Αν αυτοί οι δύο δεν επικοινωνούν, τότε υπάρχει πρόβλημα στον τρόπο που χρησιμοποιείται το κανάλι επικοινωνίας, το οποίο είναι το API.

Ο σχεδιαστής των API χρειάζεται να καταλαβαίνει τον κοινό τόπο του πλήθους στο οποίο απευθύνεται και να χρησιμοποιεί αυτήν την γνώση κατά τον σχεδιασμό των API. Όταν γράφουμε σε java, είναι ασφαλές να περιμένουμε ότι οι χρήστες γνωρίζουν έννοιες που χρησιμοποιούνται στις περισσότερες από τις βιβλιοθήκες της java. Χρησιμοποιώντας παρόμοιες κλάσεις και όρους μειώνουμε την απαίτηση μάθησης νέων πραγμάτων από τους χρήστες με αποτέλεσμα να τους είναι πιο κατανοητό και ευχάριστο το API.

Όταν κατασκευάζουμε άγνωστες δομές υπάρχει μία απλή λύση, βασιζόμενη στην παρατήρηση ότι οι περισσότεροι χρήστες των API πρώτα ψάχνουν για υπάρχουσες εφαρμογές που κάνουν κάτι παρόμοιο με την εργασίας τους, αντιγράφουν την εφαρμογή και την τροποποιούν για να καλύπτει τις ανάγκες τους. Όσα περισσότερα παραδείγματα υπάρχουν με τις περιπτώσεις χρήσης του, τόσο αυξάνεται η πιθανότητα οι χρήστες να βρουν κάτι κοντά στις ανάγκες τους. Όσο πιο πρωτότυπη είναι η χρήση του API, τόσο μεγαλύτερο θα είναι και το όφελος που θα αποκομιστεί από το εύρος των παραδειγμάτων.

Σταθερότητα

Μια άλλη σημαντική πτυχή ενός API που μπορεί εύκολα να ελεγχθεί είναι το κατά πόσον είναι σταθερό και συνεπές. Εάν οι χρήστες ένός API πρέπει να επενδύσουν χρόνο για να μάθουν μια έννοια, είναι σημαντικό η έννοια αυτή να εφαρμόζεται με σταθερότητα σε όλο το API.

Από την πλευρά του κύκλου ζωής των API, είναι σημαντικό να ακολουθείται ότι και στην ανάπτυξη του ή τουλάχιστον όσα τηρούνται για την εξέλιξη του. Είναι δυσάρεστο να εντοπίζονται τμήματα του API που υλοποιούνται με διαφορετικό τρόπο από άλλα. Για παράδειγμα η περίπτωση της διεπαφής org.w3c.dom αποτελούσε μέρος του JDK της έκδοσης 1.3 και παρόλο που υπήρχε προειδοποίηση ότι ήταν «δημόσια διεύθυνση του διαδικτύου» κανένας δεν παραδεχόταν ότι αυτή η διεπαφή δεν μπορεί να υλοποιηθεί από τους χρήστες όπως οι άλλες διεπαφές των API της java. Η υλοποίηση της java.lang.Runnable θεωρείτο ασφαλής. Δηλαδή, κανένας δεν θα πρόσθετε μία νέα μέθοδο στην java.lang.Runnable, επειδή υπήρχαν πάρα πολλοί χρήστες που την υλοποιούσαν. Ωστόσο, η προσθήκη νέων μεθόδων στην org.w3c.dom συνέβη σε νεότερες εκδόσεις. Αυτό είναι προβληματικό επειδή αυτοί που είχαν υλοποιήσει την διεπαφή δεν μπορούσαν πλέον να την μεταγλωττίσουν. Η συμβατότητα των διεπαφών δυστυχώς δεν θεωρείται δεδομένη και υπάρχουν πολλές περιπτώσεις που οι χρήστες ενημερώνονται για την ασυμβατότητα, όταν ανακαλύπτουν ότι δεν μπορούν να μεταγλωττίσουν τον κώδικα τους ο οποίος ήταν απόλυτα νόμιμος σε νεότερες εκδόσεις. Αυτή η μορφή εξέλιξης, εκτός και εάν ανακοινώνεται εκ των προτέρων, είναι η χειρότερη προδοσία της εμπιστοσύνης των χρηστών.

Ανακαλυψιμότητα

Αποτελεί χείριστη περίπτωση η εύρεση ενός έργου που ισχυρίζεται ότι παρέχει ένα API το οποίο λύνει το πρόβλημα που μας απασχολεί και μας παραπέμπει σε ένα Javadoc με πέντε πακέτα και τριάντα κλάσεις, χωρίς σημείο εισόδου ή κάποιον οδηγό. Παράδειγμα αυτής της περίπτωσης είναι το Javadoc για το java.awt.Image. Είναι η γνωστή κλάση που αντιπροσωπεύει γραφικά με τα οποίο μπορούμε να σχεδιάσουμε στην οθόνη ή να αποθηκεύσουμε σε διάφορους τύπους εικόνες. Είναι μία αφηρημένη κλάση άλλα αυτό που θέλουν οι περισσότεροι χρήστες είναι να φορτώσουν μία εικόνα σε έναν δίσκο. Περιέχει την εξής οδηγία: Η εικόνα θα πρέπει να ληφθεί με έναν συγκεκριμένης πλατφόρμας τρόπο. Ο έμπειρος αναγνώστης της Java τεκμηρίωσης, πρέπει να προσέξει την υποκλάση BufferedImage και να κοιτάξει εκεί. Κενές εικόνες αυτής της κλάσης θα πρέπει να κατασκευαστούν αλλά δεν υπάρχει κάποια οδηγία για το πως θα φορτωθεί μία εικόνα από έναν δίσκο, πως θα σχεδιάσει κάποιος επί της οθόνης ή πως θα γίνει αποθήκευση. Είναι ένα αντιπροσωπευτικό παράδειγμα τεκμηρίωσης που δεν παρέχει βοήθεια στον χρήστη για να καταλάβει την χρήση του αλλά ούτε ένα σημείο εισόδου για να ωφεληθεί. Στις περισσότερες περιπτώσεις, οι κλάσεις δεν είναι στο ενδιαφέρων των χρηστών API. Το ενδιαφέρων τους έγκειται στο να γίνει η δουλειά τους. Γι'αυτό είναι πιο χρήσιμο να παρουσιάζονται παραδείγματα χρήσης ώστε να επιλέγουν την περίπτωση που είναι πιο κοντά στα ζητούμενα τους. Αυτό εν μέρη εξηγεί και την επιτυχία των έργων ανοικτού κώδικα στα οποία απλά αντιγράφεις τον κώδικα που χρειάζεσαι. Στην πραγματικότητα ο κώδικας λειτουργεί και ως τεκμηρίωση παρέχοντας καθοδήγηση. Παράδειγμα ενός έργου ανοικτού κώδικα αποτελεί το WWW που οι σχεδιαστές του αντέγραφαν παραδείγματα κώδικα από τους φυλλομετρητές τους.

Ανεξάρτητα του τύπου του API που παρέχεται πρέπει να υπάρχει ένα σημείο εισόδου που θα κατευθύνει τους χρήστες ώστε να λύσουν τα προβλήματα τους. Καθώς οι χρήστες δεν συνηθίζεται να ασχολούνται με τις κλάσεις των API, είναι σημαντικό να οργανώνονται τα σημεία εισόδου βασισμένα στις πραγματικές ή τουλάχιστον στους προβλεπόμενες εργασίες.

Ευκολία στις απλές εργασίες

Μερικές φορές τα API απευθύνονται σε διαφορετικές ομάδες χρηστών. Το πρώτο και βασικότερο λάθος είναι να τίθενται θέματα που το ενδιαφέρων τους διαχέεται σε διαφορετικά τμήματα στο API. Αυτό δυσχεράνει την ανακαλυψιμότητα, καθώς οι χρήστες ενδιαφέρονται μόνο για μια πτυχή του API η οποία θα κρατήσει και την προσοχή τους από άλλες πτυχές που σχεδιάστηκαν για διαφορετικές ομάδες χρηστών. Μια κοινή προσέγγιση είναι ο διαχωρισμός του API σε δύο ή περισσότερα τμήματα. Ένα που θα στοχεύει στους καλούντες και το άλλο σε διαφορετικό πακέτο για τους παρασχεθέντες του API. Μια άλλη προσέγγιση είναι η δημιουργία διαφορετικών διεπαφών για διαφορετικές ομάδες χρηστών, διαχωρισμένα σε διαφορετικά πακέτα. Οι καλούντες των API απλά χρησιμοποιούν το javax.naming και το javax.naming.event, καθώς οι υλοποιητές ενδιαφέρονται περισσότερο για το javax.naming.spi. Αυτού του είδους ο διαχωρισμός είναι πιο σημαντικός από την τεκμηρίωση του API στο ότι παρέχει αυτόματα πεδίο δράσης. Οι διαφορετικές ομάδες χρηστών API μπορούν εύκολα να εστιάσουν την μελέτη και την προσοχή τους σε περιοχές ενδιαφέροντος για αυτούς, χωρίς να αναρωτιούνται αν είναι κάτι που τους απασχολεί για κάθε κλάση που βλέπουν στην τεκμηρίωση. Ο λανθασμένος όμως διαχωρισμός ή το να απευθυνθεί σε λάθος ομάδα χρηστών είναι κάτι που μειώνει σημαντικά την χρησιμότητα των API.

Διαφύλαξη της επένδυσης

Είναι σημαντικό για οποιονδήποτε που σχεδιάζει ένα API να μεταχειρίζεται τους συνεργάτες του δίκαια δηλαδή τους χρήστες του. Οι χρήστες των API δημιουργούν περισσότερες χρήσεις των API κι αντίστροφα. Όσοι περισσότεροι χρησιμοποιούν την βιβλιοθήκη JUnit, τόσο πιο σημαντική γίνεται καθώς και ο τρόπος που αναπτύχθηκε. Για να γίνει κάποιος χρήστης μια βιβλιοθήκης πρέπει να την καταλάβει και να κρίνει ότι μπορεί να κάνει την δουλειά του ευκολότερα. Επιπροσθέτως για τα API, οι χρήστες πρέπει να πεισθούν ότι η δουλειά τους θα διαφυλαχθεί στο μέλλον και από τις νεότερες εκδόσεις των βιβλιοθηκών. Η συγγραφή κώδικα σε μια βιβλιοθήκη είναι επένδυση του χρόνου, της μελέτης, της προσπάθειας, ενίοτε και των χρημάτων μας. Η πρώτη ευθύνη ενός σχεδιαστή API είναι να διαφυλάξει αυτή την επένδυση του χρήστη. Κάθε νέα έκδοση βιβλιοθήκης πρέπει να διασφαλίζει την συνέχεια της εκτέλεσης ή σε αντίθετη περίπτωση, την παροχή αναβάθμισης ώστε να εκτελείτε και πάλι ο κώδικας. Η προσδοκία της ανάγκης των νέων εκδόσεων πρέπει να υφίσταται εξ αρχής για την αντιμετώπιση πιθανής ασυμβατότητας.

Δυνατότητα διόρθωσης και επαναμεταγλώττισης έχουν περισσότερο, ως πιο ευέλικτες, οι λιγότερο αυστηρές ρυθμίσεις. Αυτό συνηθίζεται σε ανοικτού κώδικα προγράμματα και συστήματα όπως το Linux. Σ'αυτές τις ρυθμίσεις δεν είναι αναγκαία η απόλυτη συμβατότητα, αρκεί να επαναμεταγλωττίζετε η νέα έκδοση. Εάν επιτευχθεί αυτό και οι προσδοκίες επαληθευτούν, το στοίχημα της διαφύλαξης έχει κερδηθεί.

Από την άλλη πλευρά της διαφύλαξης είναι πρόθεση των περισσοτέρων να γίνει το API καλύτερο και ομορφότερο. Η μετονομασία των μεθόδων να είναι αυτονόητη, η αναδιάρθρωση των μοτίβων του API να είναι πιο κατανοητή και ούτω καθεξής. Αυτές οι δραστηριότητες είναι καλοδεχούμενες στην πρώτη έκδοση του API. Ωστόσο, μετά την πρώτη έκδοση, δεν γίνεται να αντισταθμιστούν τα προβλήματα που σχετίζονται με την διάρρηξη του υφιστάμενου κώδικα του API. Μία τέτοια στάση, δεν διαφυλάσσει την επένδυση των χρηστών του API.

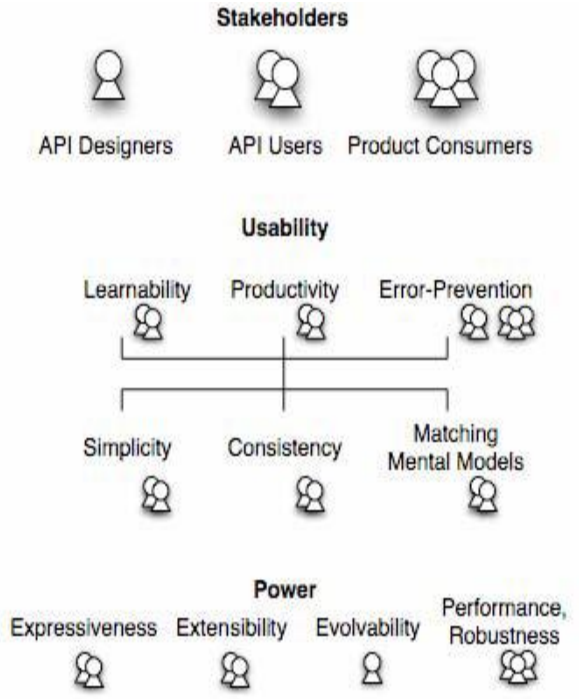
Χρησιμότητα

Αναφέρεται στην χρήση των API's κατά την δημιουργία και το debugging του κώδικα. Περιλαμβάνει ως έννοια την ευκολία εκμάθησης, την αποδοτικότητα, την προστασία σφαλμάτων, την απλότητα, την σταθερότητα και τον βαθμό καταλληλότητας των API's στους χρήστες. Η χρησιμότητα επηρεάζει περισσότερο τους χρήστες ενώ η προστασία σφαλμάτων τους καταναλωτές

Δύναμη

Η δύναμη περιλαμβάνει: την εκφραστικότητα των API's, την επεκτασιμότητα (ώστε οι χρήστες να κάνουν ένα εξειδικευμένο πρόγραμμα), την εξελιξιμότητα (για update και νέες εκδόσεις), την εκτέλεση (ταχύτητα και χρήση μνήμης), την ευρωστία και την χωρίς λάθη υλοποίηση των API's. Η δύναμη επηρεάζει περισσότερο τους χρήστες και τους καταναλωτές ενώ η εξελιξιμότητα τους σχεδιαστές.

Στο Σχήμα_ΙΙ εμφανίζονται οι εμπλεκόμενοι, η χρησιμότητα και η δύναμη με τα χαρακτηριστικά τους.



Σχήμα_II

Επίλογος

Οι μηχανικοί λογισμικού διδάσκονται να μετρούν τα πάντα. Όλες οι μετρήσεις που θα παρουσιαστούν είναι μία ανταμοιβή για την τήρηση των σχεδιαστικών κανόνων στο μέγιστο βαθμό και την επιβεβαίωση της ορθότητας τους. Η ποιότητα των API's υπολογίζεται από διάφορες σκοπιές, με πολλούς τρόπους και αυστηρά κριτήρια.

Στο τέλος του κεφαλαίου 1 παρουσιάστηκαν κάποιες βασικές έννοιες που αποτελούν τρόπους ελέγχου της ποιότητας των API's, στο επόμενο κεφάλαιο θα μελετηθεί και θα αναλυθεί η λειτουργία ενός πλήθους μετρικών.

Κεφάλαιο 2 Μετρικές Μέθοδοι

Εισαγωγή

Ο λόρδος Kelvin κάποτε είπε:

Όταν μπορείς να μετρήσεις αυτό για το οποίο μιλάς και μπορείς να το περιγράψεις με αριθμούς ξέρεις κάτι γι' αυτό, αλλά όταν δεν μπορείς να μετρήσεις, όταν δεν μπορείς να το περιγράψεις με αριθμούς, η γνώση σου είναι ανεπαρκής: μπορεί να είναι η αρχή της γνώσης, αλλά ελάχιστα έχεις προοδεύσει στις σκέψεις σου, στο στάδιο μίας επιστήμης.

Τα λόγια του λόρδου Kelvin άρχισαν να κερδίζουν έδαφος στη κοινότητα ανάπτυξης λογισμικού τα τελευταία 10 χρόνια. Πριν όμως προχωρήσουμε στη περιγραφή των λόγων χρήσης μετρήσεων και μετρικών καλό θα ήταν να ορίσουμε τι είναι αυτά.

Αρχικά με τον όρο μέτρηση (measure) στα πλαίσια της επιστήμης λογισμικού εννοούμε τη ποσοτική ένδειξη της έκτασης, της ποσότητας, των διαστάσεων, του όγκου ή του μεγέθους κάποιου χαρακτηριστικού ενός προϊόντος ή διεργασίας. Ενώ, ο όρος μετρική (metric) όπως μας δίνεται από το καθιερωμένο γλωσσάρι όρων λογισμικού της IEEE [IEEE93] ορίζεται ως η ποσοτική μέτρηση του βαθμού στον οποίο ένα σύστημα, συνιστώσα ή διεργασία κατέχει ένα χαρακτηριστικό. Τέλος, μία ένδειξη (indicator) είναι μία μετρική ή συνδυασμός μετρικών που μας παρέχει πολύτιμες γνώσεις σε μία διεργασία λογισμικού, σε μία συνολική δουλειά ή στο ίδιο το προϊόν. Πρακτικά, η έννοια της ένδειξης είναι βοηθητική για να μπορεί να γίνει περιγραφή καταστάσεων σε ένα έργο (project) που βοηθούν τον επικεφαλής (manager) να πάρει αποφάσεις για βελτίωση.

Πάνω σε αυτή τη λογική έχουμε πολλά είδη μετρήσεων και μετρικών. Τόσο στο επίπεδο του προϊόντος όσο και στο επίπεδο του συνολικού έργου. Μετρήσεις στο επίπεδο της διεργασίας. Ο μόνος ορθολογιστικός τρόπος για τη βελτίωση διεργασιών είναι η μέτρηση συγκεκριμένων χαρακτηριστικών της διαδικασίας, η ανάπτυξη ενός συνόλου μετρικών πάνω σε αυτά τα χαρακτηριστικά και τέλος η χρήση αυτών των μετρικών για τη παροχή ενδείξεων που θα οδηγήσουν στο στρατηγικό σχεδιασμό μίας βελτίωσης. Επίσης, με την ίδια λογική μετρήσεις γίνονται και στο επίπεδο του έργου. Επιπλέον, μετρήσεις στο λογισμικό αλλά και αντικειμενοστραφείς μετρήσεις είναι παρόντες. Μετρήσεις για τη ποιότητα λογισμικού κρίνονται αναγκαίες. Μετρικές για ανάλυση μοντέλων αλλά και για δοκιμές θεωρούνται βασικές.

Παρά τη μεγάλη ποικιλία μετρήσεων και μετρικών οι περισσότερες από αυτές είναι υποκειμενικές. Για την επίτευξη ενός καλού βαθμού αντικειμενικότητας χρειάζεται η χρήση ενός συνόλου από μετρικές, ενώ ποτέ δεν πρέπει να βασίζουμε αποφάσεις σε μία μετρική που θεωρούμε καλύτερη, αλλά στο σύνολο των αποτελεσμάτων όλων των μετρήσεων.

Επίσης, δεν πρέπει να κρύβουμε τις αρνητικές μετρήσεις, εάν εμφανίζονται προβληματικές περιοχές αυτό είναι καλό καθώς υπάρχει περιθώριο βελτίωσης. Επιπλέον, δεν πρέπει να χρησιμοποιούμε συγκεκριμένη μετρική για να αξιολογήσουμε τη προσπάθεια μίας ομάδας ή ενός ατόμου που ασχολείται με το εκάστοτε έργο.

Οι μετρικές του κάθε λογισμικού εξαρτώνται από τα μοναδικά χαρακτηριστικά του εκάστοτε λογισμικού. Τα αντικειμενοστραφή λογισμικά είναι τελείως διαφορετικά από τα λογισμικά που αναπτύσσονται με χρήση συμβατικών μεθόδων. Για αυτό το λόγο, οι μετρικές για αντικειμενοστραφή συστήματα πρέπει να συμβαδίζουν με τα χαρακτηριστικά που διακρίνονται από αντικειμενοστρέφεια.

Διαχωρισμός μετρικών ανά κατηγόριες - χαρακτηριστικά

Ο Berard όρισε πέντε χαρακτηριστικά που οδηγούν σε εξειδικευμένες μετρικές: τοπικά (localization), ενθυλάκωση, απόκρυψη πληροφοριών, κληρονομικότητα, τεχνικές αφαίρεσης αντικειμένων (object abstraction techniques).

Τοπικά (Localization)

Χαρακτηριστικό του λογισμικού που δείχνει τον τρόπο με τον οποίο η πληροφορία είναι συγκεντρωμένη σε ένα πρόγραμμα. Για παράδειγμα καθοδηγούμενες από τα δεδομένα μέθοδοι συγκεντρώνουν την πληροφορία γύρω από συγκεκριμένες δομές δεδομένων.

Επειδή τα συμβατικά λογισμικά δίνουν έμφαση σε λειτουργίες όπως ο μηχανισμός της τοπικότητας, οι μετρικές επικεντρώνουν στην εσωτερική δομή, την πολυπλοκότητα των λειτουργιών (μήκος προγράμματος, συνοχή ή κυκλωματική πολυπλοκότητα) ή στον τρόπο με τον οποίο οι λειτουργίες συνδέονται μεταξύ τους (συνοχή προγραμμάτων).

Δεδομένου ότι η κλάση είναι η κύρια μονάδα ενός αντικειμενοστραφούς συστήματος, η τοπικότητα είναι βασισμένη στα αντικείμενα. Ως εκ τούτου οι μετρικές πρέπει να εφαρμόζονται στην κλάση (αντικείμενο) ως πλήρης οντότητα. Εν αντίθεση, οι σχέσεις μεταξύ λειτουργιών και κλάσεων δεν είναι υποχρεωτικά μία προς μία. Επομένως οι μετρικές εκφράζουν τον τρόπο με τον οποίο οι κλάσεις που συνεργάζονται, πρέπει να είναι ικανές να ικανοποιούν σχέσεις ένας προς πολλά και πολλά προς ένα.

Ενθυλάκωση (Encapsulation)

Ο Berard ορίζει την ενθυλάκωση ως «την συσκευασμένη (ή υποχρεωτικά συγκεντρωμένη) συλλογή στοιχείων. Παραδείγματα ενθυλάκωσης χαμηλού επιπέδου που περιλαμβάνουν εγγραφές και πίνακες, υποπρογράμματα (διαδικασίες, λειτουργίες, υπορουτίνες) είναι μεσαίου επιπέδου μηχανισμοί ενθυλάκωσης».

Η ενθυλάκωση επηρεάζει τις μετρικές αλλάζοντας την εστίαση της προσοχής τους από την μέτρηση ενός μεμονωμένου τμήματος κώδικα σε πακέτα δεδομένων.

Απόκρυψη πληροφοριών (Information Hiding)

Η απόκρυψη πληροφοριών εμποδίζει (ή κρύβει) τις λειτουργικές λεπτομέρειες ενός προγραμματιστικού συστατικού (component). Μόνο η πληροφορία που είναι απαραίτητο να έχει πρόσβαση στο συστατικό προωθείται σε άλλα συστατικά που επιθυμούν να το προσβάσουν.

Ένα καλά σχεδιασμένο ΟΟ σύστημα πρέπει να ενισχύει την απόκρυψη πληροφοριών. Ως εκ τούτου οι μετρικές που παρέχουν μία ένδειξη του βαθμού απόκρυψης που έχει επιτευχθεί μπορούν να παράσχουν ένδειξη του βαθμού της ποιότητας του ΟΟ σχεδιασμού.

Κληρονομικότητα (Inheritance)

Η κληρονομικότητα είναι ο μηχανισμός που ενεργοποιεί τις ευθύνες ενός αντικειμένου να αναπαραχθεί σε άλλα αντικείμενα. Υφίσταται σε όλα τα επίπεδα της ιεραρχίας των κλάσεων. Επειδή η κληρονομικότητα αποτελεί ένα κεντρικό χαρακτηριστικό σε πολλά αντικειμενοστραφή συστήματα, πολλές μετρικές επικεντρώνονται σε αυτό.

Αφαίρεση (Abstraction)

Berard «Αφαίρεση είναι μια σχετική έννοια. Καθώς οδεύουμε προς υψηλότερα επίπεδα αφαίρεσης αγνοούμε όλο και περισσότερες λεπτομέρειες, παρέχουμε μία γενικότερη οπτική της έννοιας ή του στοιχείου ενώ όσο πηγαίνουμε προς χαμηλότερα επίπεδα αφαίρεσης εισάγουμε περισσότερες λεπτομέρειες».

Επειδή μια τάξη είναι μια αφαίρεση που μπορεί να εξεταστεί από πολλά διαφορετικά επίπεδα και με πολλούς διαφορετικούς τρόπους, οι μετρικές αντιπροσωπεύουν αφαιρέσεις από την πλευρά της μέτρησης των τάξεων (αριθμός των στοιχείων ανά κλάση/εφαρμογή, πλήθος των παραμετροποιήσιμων τάξεων ανά εφαρμογή, αναλογία των παραμετροποιήσιμων τάξεων σε μη παραμετροποιήσιμες τάξεις).

Μετρικές για σχεδιασμό αντικειμενοστραφών μοντέλων

Σχετικά με την αντικειμενοστραφή σχεδίαση έχουν ειπωθεί και γραφτεί πολλά υποκειμενικά πράγματα που ενδεχομένως να δημιουργούν μία συγκεχυμένη άποψη. Ένας έμπειρος σχεδιαστής γνωρίζει πως να αποδώσει χαρακτηριστικά σε ένα ΟΟ σύστημα με τα οποία θα υλοποιήσει αποδοτικά τις απαιτήσεις των πελατών. Ωστόσο καθώς ένα ΟΟ σχεδιαστικό μοντέλο μεγαλώνει σε μέγεθος και πολυπλοκότητα, τόσα περισσότερα χαρακτηριστικά μπορούν να ωφελήσουν συγχρόνως και έναν σχεδιαστή (ο οποίος αποκτά πρόσθετη διορατικότητα) και έναν αρχάριο (όπου λαμβάνει μέτρηση της ποιότητας του προγράμματος του).

Σε μια λεπτομερή μελέτη των μετρικών για ΟΟ συστήματα, ο Whitmire περιγράφει εννέα διαφορετικά και μετρήσιμα χαρακτηριστικά ΟΟ σχεδίασης τα οποία θεωρούνται εννέα γενικότερες κατηγορίες των μετρικών.

Μέγεθος

Το μέγεθος ορίζεται βάσει τεσσάρων κριτηρίων: τον πληθυσμό, την ένταση, το μήκος, και την λειτουργικότητα. Ο πληθυσμός μετράται λαμβάνοντας μια στατική μέτρηση των ΟΟ οντοτήτων όπως των κλάσεων. Η ένταση είναι ταυτόσημη με τον πληθυσμό αλλά συλλέγεται δυναμικά σε μια δεδομένη στιγμή του χρόνου. Το μήκος είναι μία μέτρηση από αλληλοσυνδεόμενα στοιχεία σχεδιασμού (π.χ., το βάθος της κληρονομικότητας είναι μία μέτρηση του μήκους). Η λειτουργικότητα παρέχει μια έμμεση ένδειξη της αξίας που παρέχεται στον πελάτη από μία ΟΟ εφαρμογή.

Πολυπλοκότητα

Ο Whitmire ορίζει την πολυπλοκότητα από την πλευρά των δομικών χαρακτηριστικών εξετάζοντας πως οι κλάσεις ενός ΟΟ σχεδίου είναι αλληλένδετες μεταξύ τους.

Σύζευξη

Οι φυσικές συνδέσεις μεταξύ των στοιχείων ενός ΟΟ σχεδίου (το πλήθος των συνεργαζόμενων κλάσεων ή ο αριθμός των μηνυμάτων που μεσολαβεί μεταξύ των αντικειμένων) αντιπροσωπεύουν την σύζευξη ενός ΟΟ συστήματος.

Επάρκεια (Sufficiency)

Ο Whitmire ορίζει την επάρκεια ως «τον βαθμό στον οποίο μία αφαίρεση κατέχει τα χαρακτηριστικά που απαιτούνται από αυτήν ή τον βαθμό στον οποίο ένα σχεδιαστικό συστατικό κατέχει χαρακτηριστικά της δικής του αφαίρεσης, από την πλευρά της τρέχουσας εφαρμογής». Ένα σχεδιαστικό συστατικό (μια κλάση) είναι επαρκές όταν αναπαριστά πλήρως όλες τις ιδιότητες της εφαρμογής.

Πληρότητα (Completeness)

Η μόνη διαφορά μεταξύ πληρότητας και επάρκειας είναι «το σύνολο των χαρακτηριστικών με το οποίο συγκρίνουμε την αφαίρεση ή το σχεδιαστικό συστατικό, Whitmire». Επειδή το κριτήριο της επάρκειας πληρείται με πολλούς τρόπους, έχει έμμεση επίπτωση στον βαθμό με τον οποίο η αφαίρεση ή το σχεδιαστικό συστατικό μπορεί να επαναχρησιμοποιηθεί.

Συνοχή (Cohesion)

Το αντικειμενοστραφές συστατικό, όπως και το αντίστοιχο του στα συμβατικά λογισμικά, πρέπει να σχεδιάζεται με τέτοιο τρόπο ώστε οι εργασίες του να εκτελούνται μαζί, προς την επίτευξη ενός καλά σχεδιασμένου σκοπού. Η συνοχή μιας κλάσης καθορίζεται από την εξέταση του βαθμού με τον οποίο «το σύνολο των ιδιοτήτων που διαθέτει είναι μέρος του προβλήματος ή του σχεδίου», Whitmire.

Ομοιότητα

Ο βαθμός στον οποίο δύο ή περισσότερες κλάσεις είναι όμοιες από την άποψη της δομής, της λειτουργίας, τη συμπεριφοράς ή του σκοπού, υποδεικνύεται από αυτή τη μέτρηση.

Αρχέγονο (Primitiveness)

Ένα χαρακτηριστικό που είναι παρόμοιο με την ομοιότητα, το αρχέγονο είναι ο βαθμός στον οποίο μια εργασία είναι ατομική και δεν μπορεί να κατασκευαστεί έξω από μια ακολουθία άλλων εργασιών που περιέχονται σε μια τάξη. Μια κλάση που παρουσιάζει υψηλό βαθμό αυτού του χαρακτηριστικού ενθυλακώνει μόνο αρχέγονες εργασίες.

Αστάθεια (Volatility)

Οι σχεδιαστικές αλλαγές μπορεί να προκύψουν όταν οι απαιτήσεις τροποποιούνται ή όταν συμβαίνουν αλλαγές σε άλλα σημεία της εφαρμογής, με αποτέλεσμα την υποχρεωτική προσαρμογή του σχεδιαστικού συστατικού. Η αστάθεια ενός ΟΟ σχεδιαστικού συστατικού είναι η μέτρηση των πιθανοτήτων ότι θα συμβούν αλλαγές.

Οι μετρικές εφαρμόζονται όχι μόνο σε ΟΟ σχεδιαστικά μοντέλα άλλα και σε μοντέλα ανάλυσης. Παρακάτω παρουσιάζω μετρικές για την ποιότητα σε επίπεδο ΟΟ κλάσεων, σε επίπεδο εργασιών και σε επίπεδο δοκιμής.

Μετρικές αντικειμενοστραφών κλάσεων

Η κλάση είναι κυρίαρχο μέλος ενός ΟΟ συστήματος. Ως εκ τούτου, οι μετρικές για μεμονωμένες, ιεραρχικές και συνεργαζόμενες κλάσεις, είναι ανεκτίμητες για έναν μηχανικό λογισμικού που πρέπει να αξιολογήσει την ποιότητα των σχεδιασμών του. Η κλάση είναι συχνά ο γονέας των υποκλάσεων (που καλούνται παιδιά) και κληρονομούν ιδιότητες και λειτουργίες. Καθένα από τα χαρακτηριστικά αυτά μπορούν να χρησιμοποιηθούν ως βάση για την εφαρμογή μετρήσεων.

Μία από τις πιο γνωστές συλλογές ΟΟ μετρικών έχει προταθεί από τους Chidamber και Kemerer. Συχνά αναφέρεται ως CK ομάδα μετρικών, στην οποία οι δημιουργοί της έχουν συμπεριλάβει έξι μετρικές βασισμένες σε κλάσεις, τις οποίες θα παραθέσω αναφορικά και ορισμένες εξ'αυτών θα αναλύσω αργότερα.

Weighted methods per class (WMC)

Depth of the inheritance tree (DIT)

Number of children (NOC)

Coupling between object classes (CBO)

Response for a class (RFC)

Lack of cohesion in methods (LCOM)

Μετρικές των Lorenz και Kidd

Στο βιβλίο τους ‘Αντικειμενοστραφείς μετρικές’ το 1994, οι Lorenz και Kidd διαχωρίζουν τις μετρικές που βασίζονται σε κλάσεις σε τέσσερις κατηγορίες: στο μέγεθος, στην κληρονομικότητα, στις εσωτερικές και στις εξωτερικές. Οι μετρικές που ανήκουν στην κατηγορία μεγέθους επικεντρώνονται στην μέτρηση των χαρακτηριστικών και των εργασιών για μεμονωμένες κλάσεις και στον μέσο όρο των τιμών για το σύνολο των ΟΟ συστημάτων. Οι μετρικές της κληρονομικότητας εστιάζουν στον τρόπο με τον οποίο οι λειτουργίες της κλάσης ιεραρχικά μπορούν να επαναχρησιμοποιηθούν. Οι εσωτερικές μετρικές αναφέρονται στην συνοχή και σε θέματα που αφορούν των κώδικα ενώ οι εξωτερικές εξετάζουν την σύζευξη και την επαναχρησιμοποίηση.

Αναφορικά ορισμένες μετρικές των Lorenz και Kidd:

- Class size (CS)
- Number of operations overridden by a subclass (NOO)
- Number of operations added by a subclass (NOA)
- Specialization index (SI)

Επειδή η κλάση αποτελεί το κύριο μέλος των ΟΟ συστημάτων, είναι λίγες οι μετρικές που έχουν προταθεί για εργασίες που υπάρχουν εκτός κλάσεων. Οι Churcher και Shepperd αναφορικά με αυτό το θέμα είχαν δηλώσει:

Αποτελέσματα πρόσφατων ερευνών υποδεικνύουν ότι οι μέθοδοι τείνουν να είναι μικρές τόσο σε πλήθος ορισμάτων όσο και σε βαθμό πολυπλοκότητας, προτείνοντας ότι μια συνδετική δομή του συστήματος ενδέχεται να είναι πιο σημαντική από το περιεχόμενο των μεμονωμένων τμημάτων κώδικα του προγράμματος.

Ωστόσο, κάποιες σημαντικές πληροφορίες μπορεί να αποκομιστούν από την εξέταση του μέσου όρου των χαρακτηριστικών των μεθόδων. Οι Lorenz και Kidd είχαν προτείνει τρείς μετρικές:

Average operation size (OSavg)

Operation complexity (OC)

Average number of parameters per operation (NPavg)

Η ομάδα μετρικών MOOD

Οι Harrison, Counsell και Nithi, το 1998, πρότειναν μία συλλογή μετρικών για αντικειμενοστραφή σχεδίαση που παρέχει ποσοτικούς δείκτες για ΟΟ χαρακτηριστικά σχεδιασμού.

Ορισμένες εκ των οποίων είναι:

- Method inheritance factor (MIF)
- Coupling factor (CF)
- Polymorphism factor (PF)

Μετρικές αντικειμενοστραφούς δοκιμών

Οι μετρικές που έχουν αναφερθεί έως τώρα παρέχουν μία ένδειξη της ποιότητας του σχεδιασμού. Επιπλέον παρέχουν μία γενικότερη ένδειξη του πλήθους των δοκιμών που απαιτούνται για την εξέταση ενός ΟΟ συστήματος.

Ο Binder είχε προτείνει μία πλειάδα μετρικών που είχαν άμεση επίδραση στην δοκιμή (testing) των ΟΟ συστημάτων. Οι μετρικές είναι οργανωμένες σε κατηγορίες που αντιπροσωπεύουν σημαντικά σχεδιαστικά χαρακτηριστικά.

- Ενθυλάκωση
- Lack of cohesion in methods (LCOM)
- Percent public and protected (PAP)
- Public access to data members (PAD)
- Κληρονομικότητα
- Number of root classes (NOR)
- Fan-in (FIN)
- Number of children (NOC) and Depth of Inheritance tree (DIT)

Εν αντίθεση αυτών των μετρικών ο Binder το 1994, όρισε μετρικές για την πολυπλοκότητα των κλάσεων και τον πολυμορφισμό. Οι μετρικές που ανήκουν στην κατηγορία της πολυπλοκότητας περιλαμβάνουν τρεις CK μετρικές: την WMC, την CBO και την RFC. Οι μετρικές που σχετίζονται με τον πολυμορφισμό είναι άκρως εξειδικευμένες και υπερβαίνουν τον σκοπό της παρούσας εργασίας.

Μετρικές αντικειμενοστραφών έργων

Σχετικά με τις μετρικές των αντικειμενοστραφών έργων για να παρασχεθούν πληροφορίες κατά την διάρκεια υλοποίησης των, η παρουσία ειδικών ΟΟ μετρικών είναι υπαρκτή.

Η πρώτη ενέργεια που εκτελείται από τον επικεφαλής του έργου είναι ο σχεδιασμός, και μία από τις αρχικές του εργασίες είναι η εκτίμηση του έργου. Λαμβάνοντας υπόψη την εξελικτική διαδικασία του έργου, επανερχόμαστε στον σχεδιασμό μετά από κάθε επανάληψη του λογισμικού.

Ένα βασικό θέμα που αντιμετωπίζει ένας επικεφαλής έργου κατά τη διάρκεια του σχεδιασμού είναι η εκτίμηση του μεγέθους του λογισμικού. Το μέγεθος είναι ανάλογο της προσπάθειας που θα καταβληθεί από τον σχεδιαστή και της διάρκειας που θα απαιτηθεί.

Οι ακόλουθες ΟΟ μετρικές του Lorenz 1994 μπορούν να υποδείξουν το μέγεθος του λογισμικού:

- Number of scenario scripts (NSS)
- Number of key classes (NKC)
- Number of subsystems (NSUB)

Οι μετρικές NSS, NKC και NSUB μπορούν να συλλεχθούν από παλαιότερα ΟΟ έργα και να συσχετιστούν με την καταβαλλόμενη προσπάθεια ως επέκταση του έργου είτε ως όλον είτε ως μεμονωμένα έργα. Αυτά τα δεδομένα μπορούν επιπλέον να χρησιμοποιηθούν με τις σχεδιαστικές μετρικές για να υπολογίσουμε τις παραγωγικές μετρικές όπως τον μέσο όρο των κλάσεων ανά προγραμματιστή ή τον μέσο όρο των μεθόδων ανά εργατομήνα. Συλλογικά αυτές οι μετρικές μπορούν να χρησιμοποιηθούν για να υπολογιστεί η καταβαλλόμενη προσπάθεια, η διάρκεια, το ανθρώπινο δυναμικό που απαιτείται και άλλες χρήσιμες πληροφορίες για το εκάστοτε έργο.

Στην συνέχεια του κεφαλαίου αναλύονται και μελετώνται δέκα μετρικές ως προς τον τρόπο αλλά και την αιτία της εκάστοτε λειτουργία τους, τα αποτελέσματα τους επί συγκεκριμένων προγραμμάτων με παραδείγματα και διαγράμματα για την πληρέστερη κατανόηση τους. Οι μετρικές που επιλέχτηκαν είναι ένα επαρκές αντιπροσωπευτικό δείγμα όλων των κατηγοριών και χρήσεων των μετρικών μεθόδων λογισμικού.

Μετρική μεγέθους: LOC

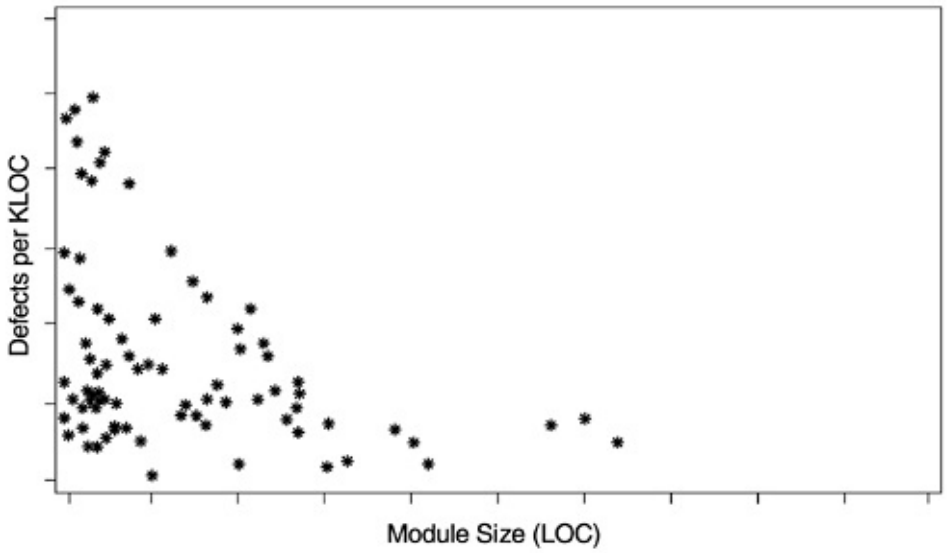
Η μετρική Lines Of Code (γραμμές του κώδικα) είναι η πιο διαδεδομένη μετρική για το μέγεθος των προγραμμάτων. Φαίνεται ότι είναι εύκολη και ακριβής αλλά υπάρχει πληθώρα διαφορετικών ορισμών για την μέτρηση των γραμμών του κώδικα ενός συγκεκριμένου προγράμματος. Αυτές οι διαφορές οφείλονται στην λογική της μέτρησης ή όχι των κενών γραμμών, των σχολίων, των μη εκτελέσιμων εντολών, των πολλαπλών δηλώσεων ανά γραμμή και των πολλαπλών γραμμών ανά δήλωση καθώς και της απόφασης για το αν θα μετρηθούν οι επαναχρησιμοποιούμενες γραμμές του κώδικα. Ο πιο κοινός ορισμός της LOC είναι αυτός που μετρά κάθε γραμμή του κώδικα εκτός των κενών γραμμών και των σχολίων, ανεξάρτητα από των αριθμό των δηλώσεων ανά γραμμή. (Boehm-81,Jones86).

Η LOC θεωρείται χρήσιμη ως δυνατότητα πρόβλεψης της πολυπλοκότητας του προγράμματος, της συνολικής αναπτυξιακής προσπάθειας και της προγραμματιστικής εκτέλεσης (αποσφαλμάτωση, παραγωγικότητα). Μειονεκτεί όμως από το προφανές πρόβλημα ότι το αποτέλεσμα της δεν μπορεί να υπολογιστεί πριν την ολοκλήρωση της σύνταξης του κώδικα. Πληθώρα μελετών επιχείρησαν να τεκμηριώσουν αυτές τις σχέσεις με ενδεικτικότερες αυτές των Woodfield-81 συγκρίνοντας την LOC, Mc-Cabe's v(G) και Halstead's ως δείκτης της επιδεξιότητας των προγραμματιστών και του Cutris-79 συγκρίνοντας την LOC με άλλες μετρικές ως δείκτη της επίδοσης των προγραμματιστών.

Επειδή ο μετρητής LOC αντιπροσωπεύει το μέγεθος του προγράμματος και την πολυπλοκότητα, όσες περισσότερες γραμμές κώδικα έχει ένα πρόγραμμα, τόσο περισσότερα ελαττώματα αναμένονται. Πιο αξιοπερίεργη είναι η διαπίστωση των ερευνητών ότι τα ελαττώματα στην πυκνότητα (ελαττώματα ανά KiloLOC) είναι άρρηκτα συνδεδεμένα με τον μετρητή LOC. Πρόσφατες μελέτες επισημαίνουν μια αντίστροφη σχέση: όσο μεγαλύτερο είναι το μέγεθος ενός τμήματος του προγράμματος (module), τόσο μικρότερο είναι το ποσοστό των ελαττωμάτων. Για παράδειγμα, οι Vasili και Perricone το 1984 εξέτασαν τμήματα προγραμμάτων γραμμένα σε Fortran με λιγότερες από 200 γραμμές κώδικα το μέγιστο του καθενός και βρέθηκε υψηλότερη πυκνότητα ελαττωμάτων στα μικρότερα τμήματα του κώδικα. Η Shen με τους συναδέλφους της το 1985 μελέτησαν λογισμικό που έχει γραφτεί σε Pascal και Assembly και βρέθηκε μια αντίστροφη σχέση που υπήρχε άνω των 500 γραμμών. Δεδομένου ότι μεγαλύτερα τμήματα προγραμμάτων είναι γενικά πιο πολύπλοκα, το χαμηλότερο ποσοστό ελαττωμάτων είναι κάπως παράλογο. Η ερμηνεία αυτών των ευρημάτων στηρίζεται στην εξήγηση των λαθών διεπαφής: Τα σφάλματα στις διεπαφές είναι ασύνδετα με το μέγεθος των τμημάτων του κώδικα και μικρότερα τμήματα προγραμμάτων υποδηλώνουν υψηλότερη πυκνότητα σφαλμάτων, εξαιτίας των συμπυκνωμένων δηλώσεων που προκαλούν υψηλότερη πολυπλοκότητα συγκριτικά μ'αυτήν του μεγέθους και κατ'επέκταση περισσότερα σφάλματα.

Πρόσφατες μελέτες εστιάζουν σε καμπυλόγραμμες σχέσεις μεταξύ των γραμμών του κώδικα και του βαθμού σφαλμάτων: Η πυκνότητα των σφαλμάτων μειώνεται με το μέγεθος και κύρτεται προς τα επάνω ξανά στην ουρά, όταν το τμήμα του κώδικα αυξάνεται, όπως παρουσιάζετε και στον Πίνακας_ΙΙ που ακολουθεί.

Πίνακας_ΙΙ



Για παράδειγμα, ο Withrow το 1990 μελέτησε προγράμματα που είχαν γραφτεί σε Ada για ένα μεγάλο έργο στο Unisys και επιβεβαίωσε την καμπυλωτή σχέση μεταξύ της πυκνότητας των σφαλμάτων (κατά τη διάρκεια της επίσημης δοκιμής και της τελικής φάσης) και του μεγέθους των τμημάτων του κώδικα. Συγκεκριμένα, σε δείγμα 362 τμημάτων κώδικα με ένα ευρύ φάσμα μεγεθών (από 63 γραμμές έως άνω των 1.000 γραμμών κώδικα), ο Withrow βρήκε την χαμηλότερη πυκνότητα ελαττωμάτων στην κατηγορία των περίπου 250 γραμμών. Η αιτία της αυξανόμενης ουράς είναι άμεσα διαθέσιμη. Όταν το μέγεθος του τμήματος του κώδικα γίνεται πολύ μεγάλο, η πολυπλοκότητα του αυξάνεται σε ένα επίπεδο που υπερβαίνει την δυνατότητα του ελέγχου και την πλήρους κατανόησης του από τον προγραμματιστή. Αυτή η νέα διαπίστωση είναι επίσης συνεπής με προηγούμενες μελέτες που δεν αντιμετώπισαν την πυκνότητα των ελαττωμάτων σε πολύ μεγάλα τμήματα κώδικα.

Το καμπυλόγραμμο μοντέλο μεταξύ μεγέθους και πυκνότητας ελαττωμάτων ανοίγει νέους δρόμους όσον αφορά την ποιότητα της μηχανικής λογισμικού. Αυτό σημαίνει ότι ενδεχομένως να βρεθεί ένα ιδανικό μέγεθος προγράμματος που να οδηγεί στο ελάχιστο ποσοστό σφαλμάτων.

Μια τέτοια βελτιστοποίηση μπορεί να εξαρτάται από τη γλώσσα προγραμματισμού, το έργο, το ζητούμενο εμπορικό προϊόν και το περιβάλλον, όμως προφανώς χρειάζονται πολλές περισσότερες εμπειρικές έρευνες. Ωστόσο, όταν μια εμπειρική βελτιστοποίηση προέλθει από εύλογες μεθόδους (π.χ., βάσει της προηγούμενης έκδοσης του ίδιου προϊόντος, ή βάσει ενός παρόμοιου προϊόντος με την ίδια ομάδα ανάπτυξης), μπορεί να χρησιμοποιηθεί ως κατευθυντήρια γραμμή για την ανάπτυξη νέων τμημάτων κώδικα

Πίνακας_ΙΙΙ

Καμπυλόγραμμη σχέση ανάμεσα σε επίπεδο σφαλμάτων και μέγεθος τμήματος κώδικα Withrow (1990)	
Μέγιστο πλήθος γραμμών κώδικα	Μέσος όρος σφαλμάτων ανά 1,000
63	1.5
100	1.4
158	0.9
251	0.5
398	1.1
630	1.9
1000	1.3
> 1000	1.4

Η μέτρηση του πλήθους των γραμμών του κώδικα επιτρέπεται να γίνεται και με λογική LOC αντί με φυσική (άμεση μέτρηση από πηγαίο αρχείο).

Δύο σημαντικά πλεονεκτήματα της λογικής LOC κατά της φυσικής είναι:

1) Το ύφος της σύνταξης του κώδικα δεν επηρεάζει την λογική LOC. Παραδείγματος χάριν η LOC δεν μεταβάλλεται κατά την κλήση μιας μεθόδου που είναι υπερβολικά συμπυκνωμένη σε κάποιες γραμμές εξαιτίας των πολλών ορισμάτων της.

2) Η λογική LOC είναι ανεξάρτητη από την γλώσσα προγραμματισμού. Μέθοδοι, ορίσματα και μεταβλητές που συλλέγονται από διαφορετικές γλώσσες προγραμματισμού είναι συγκρίσιμες και μπορούν να επεξεργαστούν.

Σημειωτέον ότι η LOC ενός τύπου είναι το το άθροισμα των μεθόδων της LOC, η LOC για ένα πεδίο ονομάτων είναι το άθροισμα των τύπων της LOC, η LOC για γλώσσα μηχανής (assembly) είναι το άθροισμα της LOC των πεδίων ονομάτων και η LOC για μια εφαρμογή είναι το άθροισμα των LOC των γλωσσών μηχανής.

Παρατηρήσεις:

1) Διεπαφές, αφηρημένες μέθοδοι και απαριθμήσεις έχουν LOC ίσο με το 0. Μόνο συγκεκριμένος εκτελέσιμος κώδικας θεωρείται υπολογίσιμος με τη LOC.

2) Πεδία ονομάτων, τύποι, πεδία και δηλώσεις μεθόδων δεν θεωρούνται γραμμή κώδικα, επειδή δεν έχουν αντίστοιχη ακολουθία σημείων.

3) Όταν ο C# ή ο VB.NET μεταγλωττιστής συναντά μία αρχικοποίηση στο πρόγραμμα, δημιουργεί μία ακολουθία σημείων για κάθε στοιχείο του δομητή (μέθοδος κατασκευής) και η ίδια παρατήρηση ισχύει για αρχικοποιήσεις στατικών δομητών ή πεδίων.

4) Η LOC που υπολογίζεται από μια ανώνυμη μέθοδο δεν συγχέεται από την LOC των δηλώσεων των εξωτερικών μεθόδων.

Συστάσεις:

Μέθοδοι όπου το πλήθος των γραμμών είναι μεγαλύτερο από 20 είναι δύσκολο να κατανοηθούν και να διατηρηθούν. Μέθοδοι όπου το πλήθος των εντολών είναι μεγαλύτερο από 40 είναι εξαιρετικά περίπλοκες και θα πρέπει να διαιρεθούν σε μικρότερες μεθόδους (εκτός εάν αυτές δημιουργούνται αυτόματα από ένα εργαλείο).

Ακολουθεί παράδειγμα με το πρόγραμμα 1 το οποίο παρουσιάζεται στο παράρτημα.

Τα αποτελέσματα της μετρικής LOC μετά από έλεγχο στο συγκεκριμένο πρόγραμμα εμφανίζεται στον Πίνακα_IV:

Πίνακας_IV

Overall		
Symbol	**Count**	**Definition**
Source Files	1	Source Files
Directories	1	Directories
LOC	251	Lines of Code
BLOC	50	Blank Lines of Code
SLOC-P	190	Physical Executable Lines of Code
SLOC-L	144	Logical Executable Lines of Code
MVG	27	McCabe VG Complexity
C&SLOC	5	Code and Comment Lines of Code
CLOC	11	Comment Only Lines of Code
CWORD	65	Commentary Words
HCLOC	0	Header Comment Lines of Code
HCWORD	0	Header Commentary Words

Οι γραμμές του κώδικα είναι 251, οι κενές γραμμές είναι 50, η φυσική LOC είναι 190 ,η λογική LOC είναι 144 και αρκετά ακόμα αποτελέσματα που εντάσσονται στην μετρική LOC.

Μετρική πολυπλοκότητας CC (Cyclomatic Complexity)

Τα κριτήρια μέτρησης της πολυπλοκότητας βασίσθηκαν στην μελέτη του McCabe (1976), ο οποίος ανέπτυξε ένα σύστημα το οποίο αποκάλεσε Cyclomatic Complexity (CC) και μπορεί να χρησιμοποιηθεί προκειμένου να αξιολογήσει την πολυπλοκότητα ενός αλγορίθμου εντός μιας μεθόδου. Το σύστημα του McCabe (1976) καταμετρά τον αριθμό των περιπτώσεων δοκιμών (test cases) οι οποίες απαιτούνται προκειμένου να δοκιμασθεί ικανοποιητικά μια μέθοδος. Αναπαριστά την γνωστική πολυπλοκότητα της κλάσης. Μετρά το πλήθος των πιθανών μονοπατιών σε ένα αλγόριθμο υπολογίζοντας τις διακριτές περιοχές του διαγράμματος ροής, δηλαδή των αριθμό των if, for και while στο σώμα της μεθόδου. Αντιπροσωπεύει τον αριθμό των ανεξάρτητων διαδρομών ενός προγράμματος λογισμικού και αποδίδει μια αριθμητική τιμή στην πολυπλοκότητα με βάσει τον ακόλουθο τύπο:

$v(G)= e – n + p$

G: το γράφημα

e: οι άκρες, τα βέλη

n: οι κόμβοι

p: τα συνδεδεμένα συστατικά

Βασικό κριτήριο για την εκτίμηση της πολυπλοκότητας μίας μονάδας λογισμικού (π.χ. μίας μεθόδου) είναι η διακλάδωση στη ροή ελέγχου. Προτάσεις if, switch, for και while διακλαδώνουν τη ροή ελέγχου. Το αποτέλεσμα είναι η αύξηση της πολυπλοκότητας του λογισμικού.

Στη γενική περίπτωση ένας χαμηλός αριθμός CC θεωρείται καλύτερος υπό την έννοια ότι συνεπάγεται μειωμένο χρόνο δοκιμών και αυξημένη κατανόηση της μεθόδου ή ότι οι αποφάσεις του προγράμματος λαμβάνονται λόγω της μετάδοσης μηνυμάτων και όχι επειδή η μέθοδος μπορεί να μην είναι πολύπλοκη.

Η μέθοδος του McCabe (1976) δεν μπορεί να χρησιμοποιηθεί προκειμένου να μετρήσει την πολυπλοκότητα μιας κλάσης λόγω του χαρακτηριστικού της κληρονομικότητας. Μπορούν όμως να συνδυασθούν οι CC μεμονωμένων μεθόδων για το σκοπό αυτό.

Ένας τρόπος υπολογισμού για κώδικα σε Java είναι να ξεκινήσουμε από ένα και να προσθέτουμε κάθε φορά που συναντούμε προτάσεις if, for, while, case και catch ή τους τελεστές &&, || και ?.

Ο McCabe προσδιόρισε το πρόβλημα ότι οποιοδήποτε πρόγραμμα με εκτελούμενη προς τα πίσω διακλάδωση έχει άπειρο πλήθος μονοπατιών. Ωστόσο είναι δυνατόν με την βοήθεια αλγεβρικών εκφράσεων να βρούμε το συνολικό αριθμό των πιθανών μονοπατιών ενός προγράμματος.

Εξαιτίας αυτής της πολυπλοκότητας αναπτύχθηκε η εν λόγω μετρική που καθορίζει τα βασικά μονοπάτια που παράγουν το κάθε πιθανό μονοπάτι.

Η τεχνική δοκιμής των βασικών μονοπατιών (basis path testing) προσδιορίζει ένα σύνολο βασικών μονοπατιών εκτέλεσης του προγράμματος και τη σχεδίαση περιπτώσεων ελέγχου για τα μονοπάτια αυτά. Κάθε μονοπάτι εκτέλεσης ξεκινά από τον κόμβο εισόδου του προγράμματος, περιλαμβάνει μια ακολουθία από ενδιάμεσους κόμβους και τελειώνει στον κόμβο εξόδου. Εάν κάθε βασικό μονοπάτι δοκιμαστεί, τότε κάθε εντολή του προγράμματος θα έχει εκτελεστεί δοκιμαστικά τουλάχιστον μια φορά τόσο για την περίπτωση που η συνθήκη είναι αληθής, όσο και για την περίπτωση που αυτή είναι ψευδής.

Το γράφημα ελέγχου ροής

Για τη γραφική αναπαράσταση των μονοπατιών εκτέλεσης χρησιμοποιείται ο συμβολισμός των γραφημάτων ροής (flow graphs). Ένα γράφημα ροής αποτελείται από κορυφές, οι οποίες αντιπροσωπεύουν μία ή περισσότερες εντολές του προγράμματος και πλευρές, οι οποίες αντιπροσωπεύουν την ροή ελέγχου. Κάθε πλευρά πρέπει να τερματίζει πάνω σε μια κορυφή. Ο χώρος που περικλείεται από πλευρές και κορυφές καλείται ‘περιοχή’. Κάθε προγραμματιστική δομή του δομημένου προγραμματισμού μπορεί να αναπαρασταθεί με ένα γράφημα ροής, συνεπώς, για κάθε αναπαράσταση διαδικασιακού προγράμματος μπορεί να κατασκευαστεί ένα αντίστοιχο γράφημα ροής.

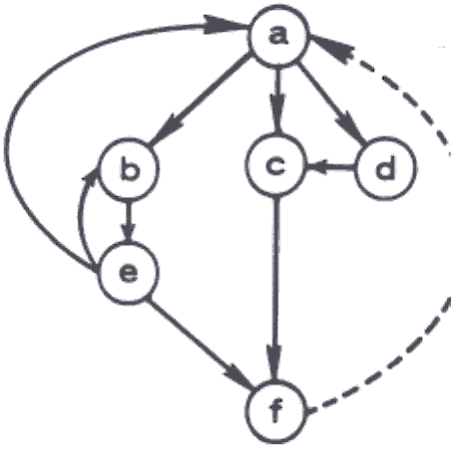
Θεώρημα 1:

Σε ένα άρρηκτα συνδεδεμένο γράφημα, η πολυπλοκότητα είναι ίση με το μέγιστο αριθμό των γραμμικών ανεξάρτητων κυκλωμάτων.

Το γράφημα έχει μία μοναδική είσοδο και εξόδους. Κάθε κόμβος στο γράφημα αναπαριστά ένα τμήμα κώδικα του προγράμματος όπου η ροή είναι ακολουθιακή και τα βέλη σε κλαδιά του προγράμματος. Κάθε κόμβος μπορεί να βρεθεί από την είσοδο και κάθε κόμβος μπορεί να βρει την έξοδο.

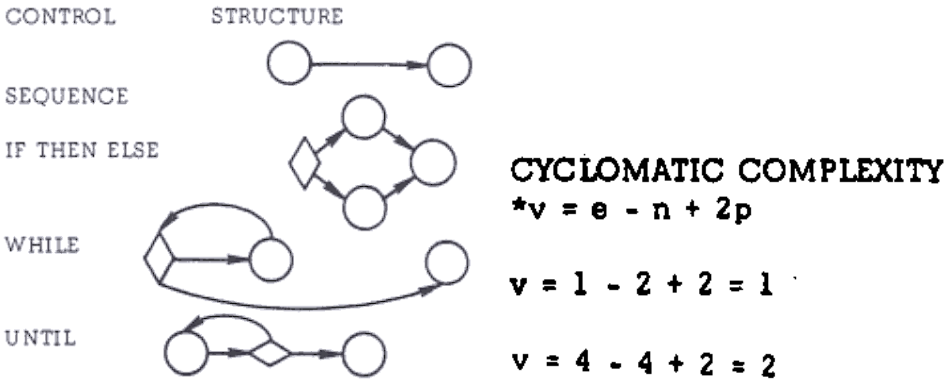
Ένα πρώτο παράδειγμα είναι το γράφημα Σχήμα_ΙΙΙ στο οποίο είσοδος είναι ο κόμβος “a” και έξοδος είναι ο κόμβος “f”.

G:



Σχήμα_III

Μερικά απλά παραδείγματα με γραφήματα των πιο συχνών δομών που χρησιμοποιούνται στον προγραμματισμό με τις πολυπλοκότητες τους:



Σχήμα_IV

Σημειώστε ότι η ακολουθία των κόμβων με αυθαίρετο αριθμό έχει πάντα πολυπλοκότητα 1 και ότι η κυκλική πολυπλοκότητα συμμορφώνεται στην αντίληψη του ελάχιστου αριθμού των διαδρομών.

Ιδιότητες:

v(G)>=1

v(G) είναι ο μέγιστος αριθμός των γραμμικά ανεξάρτητων μονοπατιών στο G, το μέγεθος της βάσης του.

Εισάγοντας ή διαγράφοντας λειτουργικές εκφράσεις στο G δεν επηρεάζεται το v(G).

Το G έχει μόνο ένα μονοπάτι μόνο εάν v(G) =1.

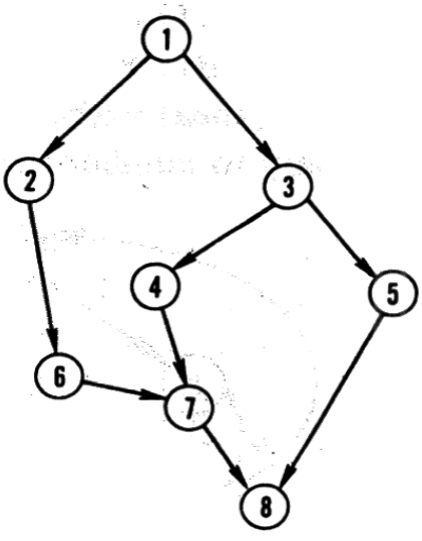
Εισάγοντας ένα νέο κόμβο στο G αυξάνεις το v(G) κατά μία μονάδα

Το v(G) εξαρτάται μόνο από την απόφαση υλοποίησης του G

Σε αυτό το σημείο απεικονίζονται μερικά γραφήματα ελέγχου ροής που αναπαριστούν πραγματικά προγράμματα . Είναι ταξινομημένα σε αύξουσα σειρά πολυπλοκότητας ώστε να αναπαρασταθεί η αντιστοιχία μεταξύ της κυκλικής πολυπλοκότητας και της πολυπλοκότητας όπως την αντιλαμβανόμαστε.

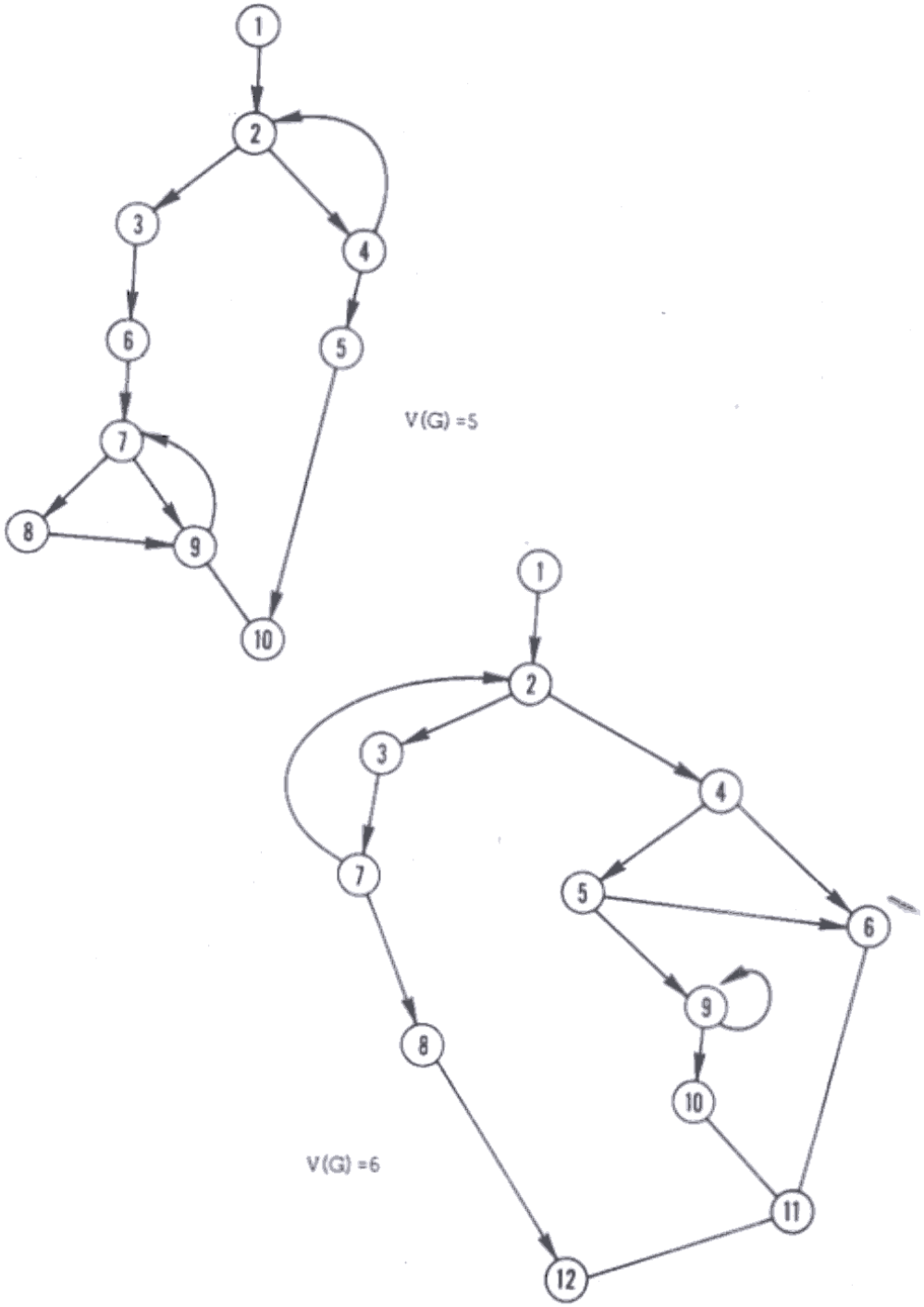
v(G)=2

Σχήμα_V

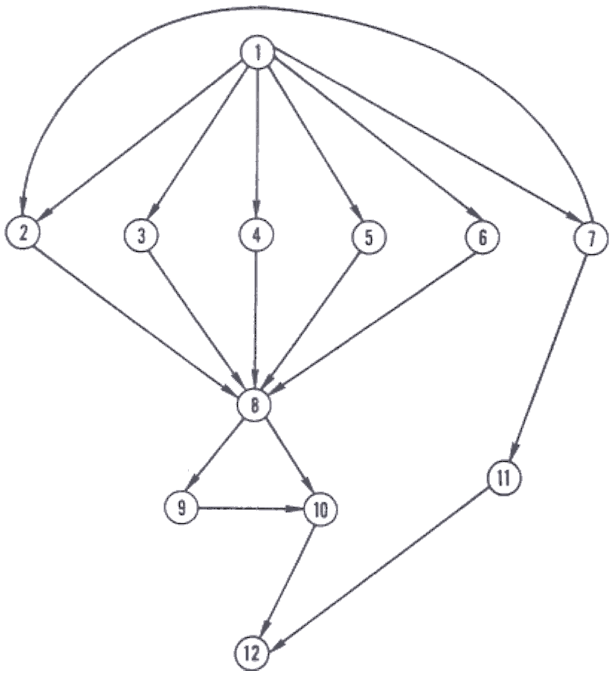


v(G)=3

Σχήμα_VI

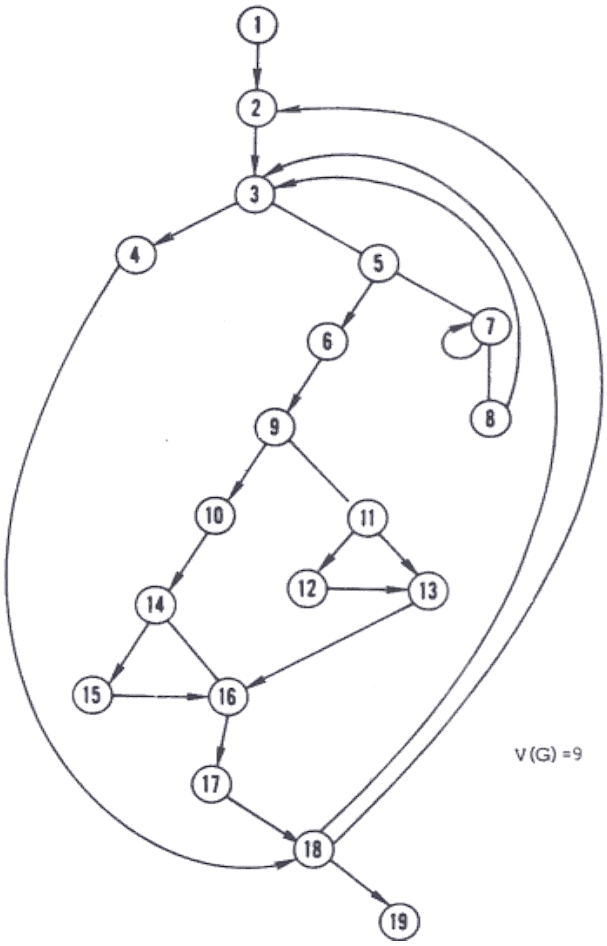


Σχήμα_VII

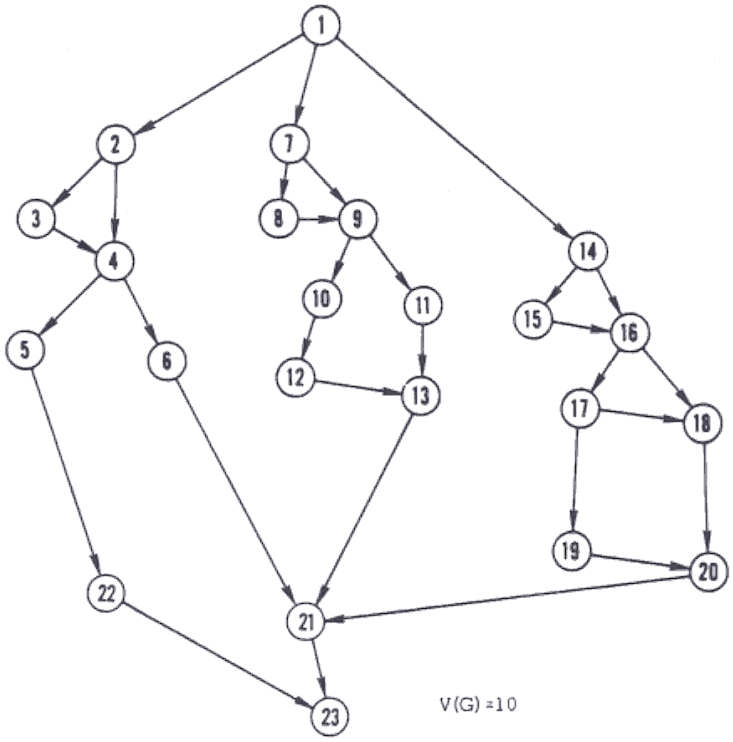


v(G)=8

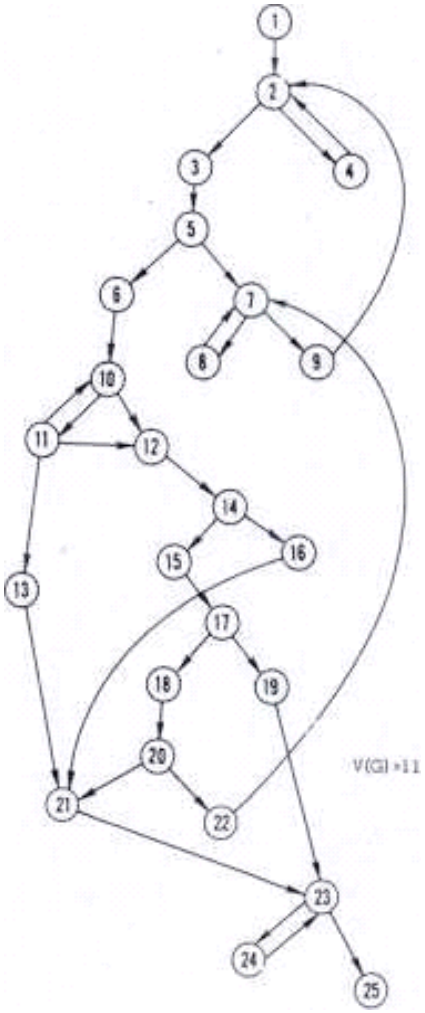
Σχήμα_VIII


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
V(G) =9

Σχήμα_IX

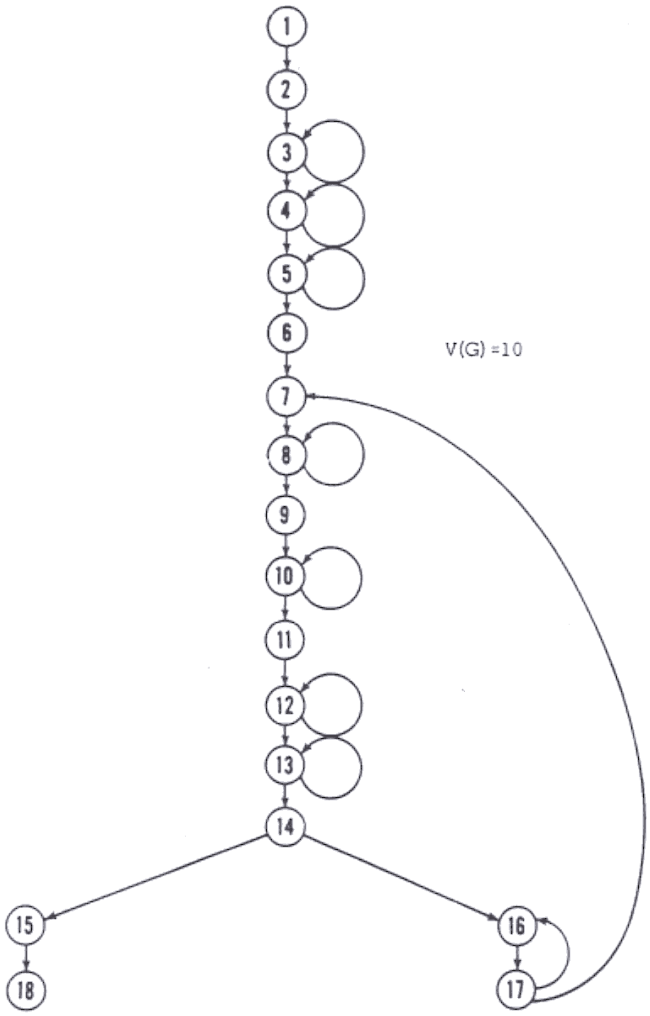

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
V(G) =10

Σχήμα_X



Σχήμα_XI

Μία από τις πιο ενδιαφέρουσες πτυχές της αυτόματης προσέγγισης είναι ότι ο έλεγχος ροής θα μπορούσε να εφαρμοστεί πολύ πιο αποτελεσματικά σε μεταγλωττιστή μιας γλώσσας προγραμματισμού. Είναι εφικτό να μειώσει την εργασία ενός προγραμματιστή Fortran από ένα χρόνο σε μόλις 20 λεπτά. Παρακάτω παρουσιάζονται γραφήματα ενός προγραμματιστή που θα μπορούσε κάποιος να αναγνωρίσει το στυλ του σημειώνοντας παρόμοια πρότυπα. Για παράδειγμα ένας προγραμματιστής είχε συγγενέψει την ακολουθία αριθμητικών απλών βρόγχων ως εξής:



Σχήμα_XII

Τέτοια αποτελέσματα χρησιμοποιήθηκαν σε λειτουργικά περιβάλλοντα συμβουλεύοντας τους προγραμματιστές να περιορίσουν τον κώδικα τους με την συγκεκριμένη πολυπλοκότητα και όχι μειώνοντας το μέγεθος του προγράμματος. Το ιδανικό μέγιστο όριο που έχει οριστεί για την πολυπλοκότητα αυτή είναι 10, το οποίο φαίνεται ένα λογικό αλλά όχι μαγικό ανώτατο όριο. Απαιτούνταν από τους προγραμματιστές να υπολογίσουν την πολυπλοκότητα του προγράμματος τους και όταν αυτή ξεπερνούσε το 10 έπρεπε είτε να αναγνωρίσουν και να τροποποιήσουν τις υπορουτίνες του είτε να επαναλάβουν την συγγραφή όλου του κώδικα. Η πρόθεση ήταν να κρατηθεί το μέγεθος του κώδικα εύχρηστο και να επιτρέπεται ο έλεγχος όλων των ανεξάρτητων μονοπατιών. Η μόνη περίπτωση στην οποία αυτό το όριο φαίνεται παράλογο είναι όταν ένας μεγάλο αριθμός ανεξάρτητων περιπτώσεων ακολουθεί μια επιλεγμένη λειτουργία (μεγάλη περίπτωση χρήσης), το οποίο επιτρέπεται.

Είναι σημαντικό να αναφερθεί πως το μεμονωμένο προγραμματιστικό στυλ σχετίζεται με την μέτρηση της πολυπλοκότητας. Ο McCabe είχε εντοπίσει προγραμματιστές που δεν είχαν εκπαιδευτεί ποτέ τους σε δομημένο προγραμματισμό κι όμως η πολυπλοκότητα των προγραμμάτων τους ήταν μεταξύ 3 και 7 δηλαδή αρκετά καλά δομημένα. Από την άλλη πλευρά

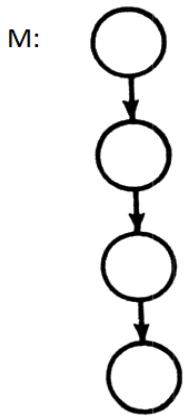
εντοπίστηκαν προγραμματιστές με πολυπλοκότητα στο κώδικα τους 40 με 50 και οι οποίοι υποστήριζαν ότι δεν υπήρχε τρόπος να συνταχθεί το πρόγραμμα τους πιο δομημένα ώστε να μειωθεί η πολυπλοκότητα. Στη μια περίπτωση ο McCabe έδωσε μία κασέτα DEC με 24 Fortran υπορουτίνες που αποτελούσαν μέρος ενός μεγάλου real time (σε πραγματικό χρόνο) συστήματος γραφικών. Θα ήταν μάλλον ανησυχητικό να βρεθούν, σε ένα σύστημα όπου η αξιοπιστία είναι κρίσιμη, υπορουτίνες της πολυπλοκότητας : 16,17,24, 24, 32, 34, 41, 54, 56, και 64. Μετά την παράθεση των αποτελεσμάτων στους προγραμματιστές ο McCabe είπε ότι οι υπορουτίνες στις DEC κασέτες επελέγησαν επειδή ήταν "ενοχλητικές" και πράγματι με υψηλή σχέση μεταξύ του βαθμού πολυπλοκότητας και του βαθμού αξιοπιστίας.

Η έννοια του p

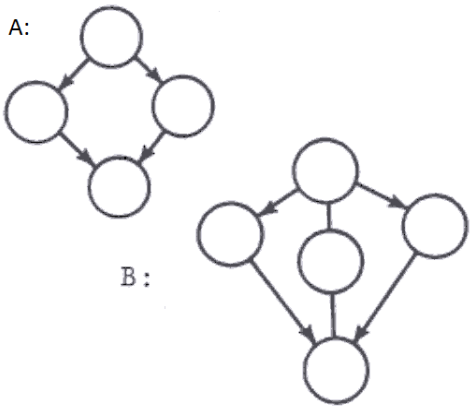
Κρίνεται επιβεβλημένο να εξηγηθεί ο ρόλος του p στον υπολογισμό της πολυπλοκότητας v = e – n + 2p.

Ο αρχικός ορισμός αναφέρει το p ως τον αριθμό των συνδεδεμένων συστατικών. Ο τρόπος με τον οποίο ορίζουμε το γράφημα ελέγχου ροής (μοναδική είσοδος και κόμβοι εξόδου, όπου όλοι οι κόμβοι βρίσκονται από την είσοδο και η έξοδος βρίσκεται από όλες) έχει ως αποτέλεσμα σε όλα τα γραφήματα ένα μόνο συνδεδεμένο συστατικό.

Ωστόσο μπορεί να υπάρχει ένα κυρίως πρόγραμμα M και δύο καλούμενες υπορουτίνες A και B που να έχουν μία δομή ως εξής:



Σχήμα_XIII



Σχήμα_XIV

Το συνολικό γράφημα όπως αναπαρίσταται έχει 3 συνδεδεμένα συστατικά (ένα γράφημα είναι συνδεδεμένο αν για κάθε ζεύγος κορυφών υπάρχει μια αλυσίδα από το μία στην άλλη. Δοθείσας μιας κορυφής, το σύνολο των κορυφών που μπορεί να συνδεθεί με το a, μαζί με το ίδιο το a είναι συνδεδεμένο συστατικό) ως MUAUB. Με p=3 υπολογίζουμε την πολυπλοκότητα ως εξής:

v(MUAUB) = e - n + 2p = 13 - 13 + 2X3 = 6

Αυτή η μέθοδος με p διάφορο του 1 χρησιμοποιείται για να υπολογιστεί η πολυπλοκότητα μια συλλογής προγραμμάτων κυρίως σε μια ιεραρχία φωλιασμένων υπορουτινών όπως παρουσιάζεται παραπάνω.

Σημειωτέον ότι

v(NUAUB) = v(M) + v(A) + v(B) =6

Γενικά η πολυπλοκότητα συλλογής C γραφημάτων με k συνδεδεμένα συστατικά είναι ίση με το άθροισμα της πολυπλοκότητας των. Έστω ότι Ci, 1<=i<=k, όπου k τα συνδεδεμένα συστατικά και ei και ni ο αριθμός των κορυφών και κόμβων στην ιοστή, τότε έχουμε:

$$v(C) = e - n + 2p = \sum_{1}^{k} e_i - \sum_{1}^{k} n_i + 2k$$

$$= \sum_{1}^{k} (e_i - n_i + 2) = \sum_{1}^{k} v(C_i).$$

Από την στιγμή που ο υπολογισμός v = e - n + 2p μπορεί να είναι κουραστικός για τον προγραμματιστή έχει γίνει μία προσπάθεια απλούστευσης των υπολογισμών της πολυπλοκότητας για γραφήματα με ένα συστατικό. Υπάρχουν δύο τρόποι, ο πρώτος επιτρέπει να γίνουν οι υπολογισμοί από την πλευρά της σύνταξης-κατασκευής του προγράμματος και ο δεύτερος, απλούστερα, από το γράφημα.

Θεώρημα:

Εάν ο αριθμός των λειτουργιών, κατηγορημάτων και η συλλογή των κόμβων σε δομημένο πρόγραμμα είναι θ, π, και y αντίστοιχα, με e των αριθμό των κορυφών τότε:

e = 1 + θ + 3π

Αφού για κάθε κατηγόρημα υπάρχει ακριβώς μία συλλογή κόμβων με μοναδική είσοδο και εξόδους ισχύει το ακόλουθο:

n = θ + 2π + 2

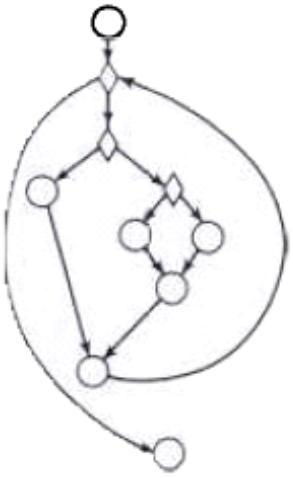
Αντικαθιστώντας το p = 1 σε v = e - n + 2 έχουμε

v = (1 + θ + 3π) - (θ + 2π + 2) + 2 = π + 1

Αυτό αποδεικνύει ότι η κυκλωματική πολυπλοκότητα ενός δομημένου προγράμματος είναι ίση με τον αριθμό των κατηγορημάτων + 1.

Για παράδειγμα στην πολυπλοκότητα v(G) = π + 1 = 3 +1 = 4

G



Σχήμα_XV

Είναι προφανής η ευκολία μέτρησης της πολυπλοκότητας απλά μετρώντας τον αριθμό των κατηγορημάτων του κώδικα χωρίς να εμπλεκόμαστε με τον γράφημα.

Στην πράξη συνθέτοντας κατηγορήματα όπως IF C1 AND C2 then αντιμετωπίζονται ως δύο συμβαλλόμενα στην πολυπλοκότητα καθώς χωρίς το συνδετικό AND θα είχαμε

IF C1 THEN IF C2 THEN

δηλαδή 2 κατηγορήματα. Γι'αυτόν τον λόγο και για δοκιμαστικούς σκοπούς μας διευκολύνει περισσότερο η μέτρηση συνθηκών και όχι κατηγορημάτων για τον υπολογισμό της πολυπλοκότητας. Έχει αποδειχθεί ότι γενικά η πολυπλοκότητα οποιουδήποτε (μη δομημένου) προγράμματος είναι π + 1.

Ο δεύτερος τρόπος υπολογισμού του e - n + 2p μειώνει τον υπολογισμό της οπτικής επιθεώρησης του γραφήματος ελέγχου ροής. Χρησιμοποιώντας το μοντέλο του Euler εάν το G είναι το γράφημα με n κορυφές, e κόμβοι και r περιφέρειες έχουμε:

n - e + r = 2

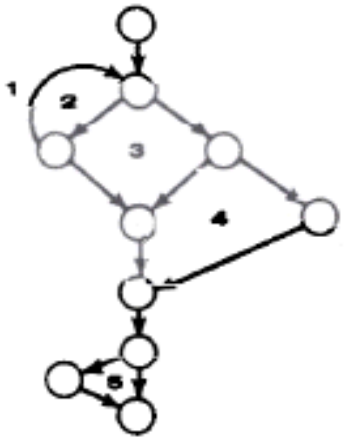
Απλά αλλάζοντας την σειρά των όρων παίρνουμε

r = e - n + 2

άρα ο αριθμός των περιφερειών ισούται με την κυκλωματική πολυπλοκότητα.

Δοθέντος ενός προγράμματος με επίπεδα στο γράφημα ελέγχου ροής μπορούμε να υπολογίσουμε το v απλά μετρώντας τις περιοχές του, όπως σ'αυτό:

G:



v(G) = 5

Σχήμα_XVI

Μεθοδολογία δοκιμών

Η μέτρηση της πολυπλοκότητας v είναι σχεδιασμένη να συμφωνεί με την διαισθητική έννοια της πολυπλοκότητας και δεδομένου ότι κατά μέσο όρο ξοδεύουμε το πενήντα τοις εκατό του χρόνου μας σε δοκιμές και αποσφαλμάτωση, η μέτρηση πρέπει να συνδέεται άρρηκτα με το μέγεθος της εργασίας που απαιτείται για την δοκιμή του προγράμματος. Επομένως αξίζει να επισημανθεί η σχέση μεταξύ της δοκιμής και της κυκλωματικής πολυπλοκότητας καθώς και ο ορισμός της μεθοδολογίας δοκιμών.

Ας υποθέσουμε ότι το πρόγραμμα P έχει πολυπλοκότητα v και ο αριθμός των μονοπατιών του είναι ac (actual complexity). Εάν το ac είναι μικρότερο από το v τότε μία από τις παρακάτω συνθήκες είναι αληθής:

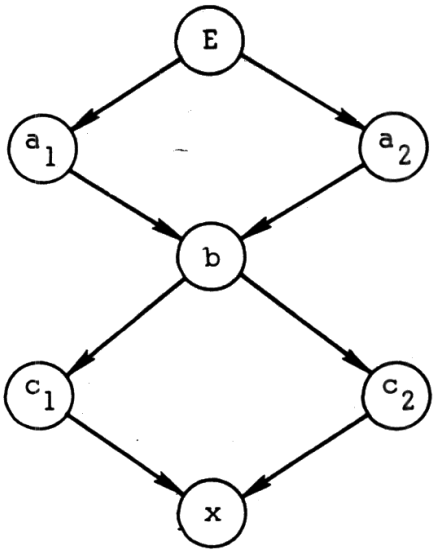
1) Πρέπει να γίνουν περισσότερες δοκιμές (να δοκιμαστούν περισσότερα μονοπάτια)

2) Το γράφημα ελέγχου ροής του προγράμματος πρέπει να μειωθεί ως προς την πολυπλοκότητα του για v - ac (v - ac αποφάσεις που πρέπει να ληφθούν) και

3) Τμήματα του προγράμματος πρέπει να μειωθούν ως προς τις γραμμές του κώδικα (η πολυπλοκότητα έχει αυξήσει το φυλασσόμενο χώρο).

Μέχρι αυτό το σημείο το θέμα της πολυπλοκότητας έχει αναπτυχθεί καθαρά από την πλευρά της κατασκευής του γραφήματος ελέγχου ροής. Το θέμα της δοκιμής, όμως, συνδέεται στενά με τη ροή των δεδομένων, διότι η συμπεριφορά των δεδομένων είναι αυτή που είτε απαγορεύει είτε καθιστά λογική την εκτέλεση οποιασδήποτε συγκεκριμένης διαδρομής ελέγχου. Μερικά απλά παραδείγματα μπορούν να βοηθήσουν στην καλύτερη κατανόηση. Έστω ότι ξεκινάμε με το παρακάτω γράφημα ελέγχου ροής.

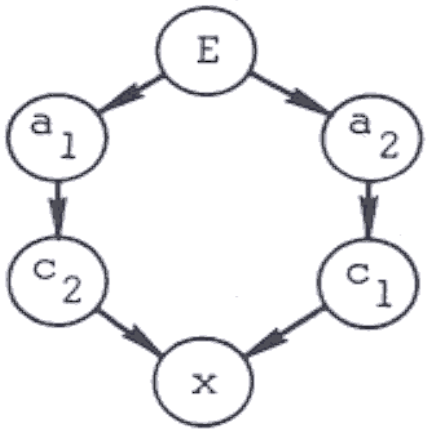
G:



Σχήμα_XVII

με ac = 2 και με δύο δοκιμασμένα μονοπάτια [E, a1 , b, C2 , x] και [E,a2,b,C1,x]. Μετά δίνοντας τα μονοπάτια [E, a1 , b, C1, x] και [E,a2,b,C2,x] δεν μπορούμε να το εκτελέσουμε αν ac <= ν επομένως διατηρείται η περίπτωση 2 και το G μπορεί να μειωθεί αφαιρώντας την απόφαση b με το παρακάτω αποτέλεσμα

G1:



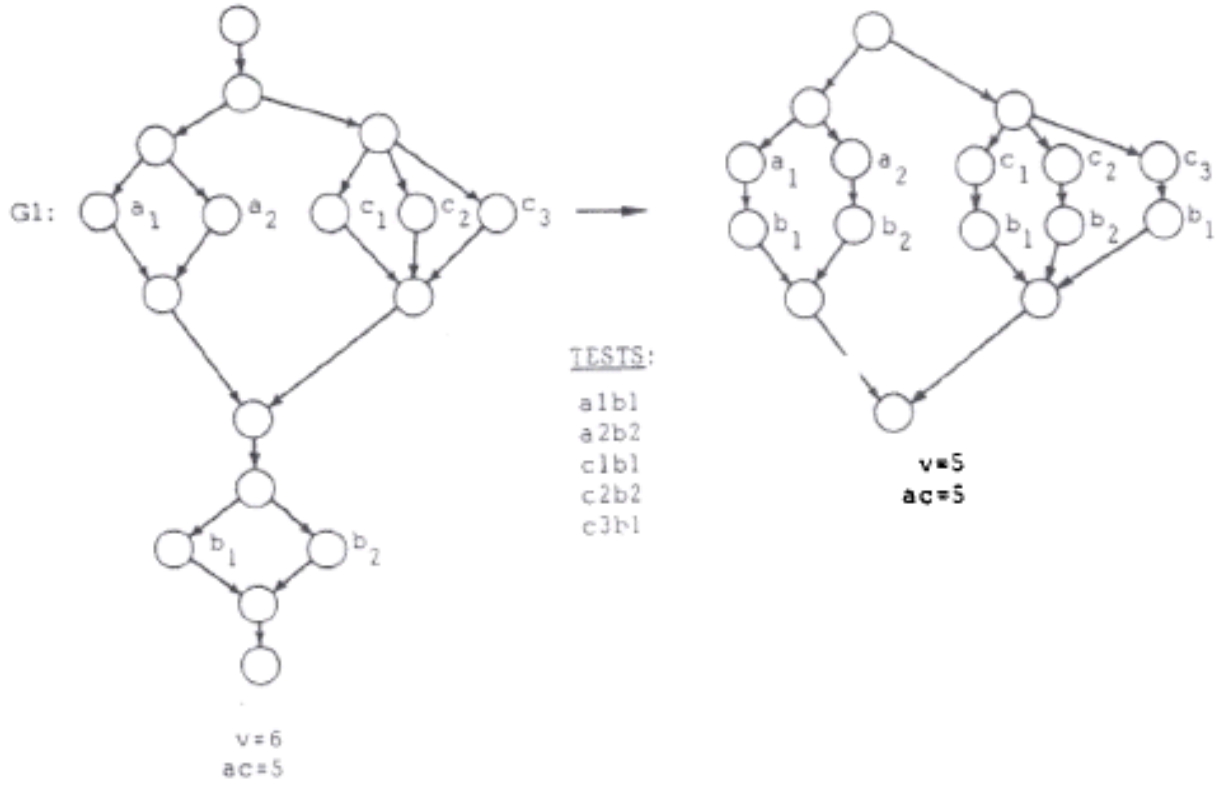
Σχήμα_XVIII

Σημειωτέον πως στο G v = ac και η πολυπλοκότητα του G1 είναι μικρότερη από αυτήν του G.

Εμπειρικά η προσέγγιση αυτή είναι πιο ωφέλιμη όταν απαιτείται από τους προγραμματιστές να τεκμηριώσουν το γράφημα και την πολυπλοκότητα τους και να παρουσιάσουν με σαφήνεια τις διαφορετικές διαδρομές που έχουν δοκιμαστεί. Είναι συχνή η περίπτωση όπου ο πραγματικός αριθμός των δοκιμασμένων διαδρομών είναι συγκρίσιμος με την κυκλωματική πολυπλοκότητα όπου αρκετά εναλλακτικά μονοπάτια ανακαλύπτονται, τα οποία κανονικά θα έπρεπε να αγνοηθούν. Πρέπει να σημειωθεί ότι το v είναι ο ελάχιστος αριθμός των ανεξάρτητων μονοπατιών που πρέπει να δοκιμαστεί. Συχνά υπάρχουν και εναλλακτικά μονοπάτια που πρέπει να δοκιμαστούν.

Πρέπει επιπλέον να σημειωθεί ότι αυτή η διαδικασία (όπως και οποιαδήποτε άλλη διαδικασία δοκιμής) δεν εγγυάται ούτε αποδεικνύει την τελειότητα του λογισμικού, το μόνο που μπορεί να κάνει είναι να εξαλείψει σφάλματα και να βελτιώσει την ποιότητα του.

Στο σχήμα Σχήμα_XIX παρουσιάζονται δύο τελικά παραδείγματα χωρίς σχόλια.



Σχήμα_XIX

Μετρική κληρονομικότητας:
DIT (Depth of Inheritance Tree)

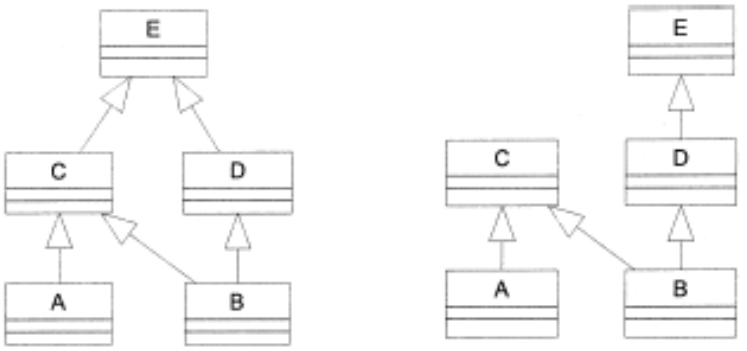
Η κληρονομικότητα είναι ένα είδος σχέσης μεταξύ των κλάσεων που παρακινεί τον προγραμματιστή να χρησιμοποιήσει προηγούμενα ορισμένα αντικείμενα, μεταβλητές και τελεστές. Το βάθος της κληρονομικότητας ιεραρχούμενο μπορεί να μας οδηγήσει σε αστάθεια του κώδικα με αύξηση της πολυπλοκότητας και με απρόβλεπτη συμπεριφορά του προγράμματος.

Δυο βασικές μετρικές που έχουν προταθεί για την μέτρηση κλάσεων σχετικά με την κληρονομικότητα είναι η DIT (βάθος δενδροειδούς κληρονομικότητας) και η NOC (Number Of Children). Χρησιμοποιείται επίσης κι άλλη μία μετρική που είναι η Number Of Methods για την μέτρηση των μεθόδων.

Στην παρούσα εργασία θα αναπτύξω την DIT η οποία είναι η αντιπροσωπευτικότερη των παραπάνω αφού υπολογίζει το πλήθος των κλάσεων γονέα μέχρι την ρίζα. Η μεγαλύτερη DIT είναι η πιο μακρινή κλάση ιεραρχικά, δηλαδή η DIT είναι ίση με το μέγιστο μήκος από την κλάση έως την ρίζα. Μεγαλύτερα δέντρα συνιστούν μεγαλύτερη σχεδιαστική πολυπλοκότητα, αφού εμπλέκονται περισσότερες κλάσεις και μέθοδοι. Στην περίπτωση που εμπλέκονται πολλαπλές κληρονομικότητες η DIT είναι το μεγαλύτερο μήκος από τον κόμβο μέχρι την ρίζα του δέντρου (Chidamber και Kemerer, 1994). Η κληρονομικότητα μπορεί να μειώσει φυσικά την πολυπλοκότητα μειώνοντας τις λειτουργίες αλλά αυτή η αφαίρεση των αντικειμένων μπορεί να καταστήσει την συντήρηση και τον σχεδιασμό δύσκολο.

Στο σχήμα Σχήμα_XX (a) ο υπολογισμός της τιμής της DIT είναι απλός για τις κλάσεις A και B, επειδή το μέγιστο μήκος από τις A και B μέχρι την ρίζα E μπορεί να προσδιοριστεί ακριβώς, DIT(A).DIT(B).2 Στο σχήμα Σχήμα_XX (b) ωστόσο, το μέγιστο μήκος από τον κόμβο B δεν είναι σαφές. Υπάρχουν δύο ρίζες στο σχήμα. Το μέγιστο μήκος από τον κόμβο A έως την ρίζα C είναι ένα και το μέγιστο μήκος από τον κόμβο A έως την ρίζα B είναι δύο. Ο δεύτερος διφορούμενος παράγοντας έγκειται στους αντικρουόμενους στόχους που ορίζονται στην θεωρητική βάση και στις οπτικές γωνίες εξέτασης της μετρικής DIT. Θεωρητικά η DIT «μετράει το πλήθος των κλάσεων προγόνων που μπορούν να επηρεάσουν αυτή την κλάση» (Chidamber και Kemerer, 1994). Οι οπτικές της μετρικής δηλώνουν την μέτρηση του πλήθους των κλάσεων που μπορούν να επηρεάσουν την κλάση.

Και οι δύο, θεωρητική βάση και οπτικές γωνίες της μετρικής, επικεντρώνουν την μέτρηση στον υπολογισμό του πλήθους των προγόνων κλάσεων της κλάσης. Ωστόσο, ο ορισμός της DIT δηλώνει την μέτρηση ως τον υπολογισμό του μήκους του μονοπατιού στο ιεραρχικό δέντρο, το οποίο είναι η απόσταση μεταξύ δύο κόμβων στο γράφημα Σχήμα_XX. Αυτό προκαλεί σύγκρουση.



(a) Inheritance tree with single root. (b) Inheritance tree with multiple roots.

Fig. 1. Inheritance trees with single and multiple roots.

Σχήμα_XX

Αυτή η ασάφεια είναι ορατή μόνο όταν υφίστανται πολλαπλές κληρονομικότητες σε ένα ΟΟ σχεδιασμό στον οποίο η απόσταση μεταξύ της κλάσης και της ρίζας δεν έχει την ίδια τιμή με το πλήθος των κλάσεων προγόνων για την κλάση. Αυτή η σύγκρουση είναι ορατή στο σχήμα D1. Σύμφωνα με τον ορισμό της DIT οι κλάσεις A και B έχουν το ίδιο μέγιστο μήκος από την ρίζα του δέντρου ως τους εαυτούς τους. Η κλάση B όμως κληρονομεί από περισσότερες κλάσεις απ'ότι η κλάση A. Σύμφωνα με την θεωρητική βάση και τις οπτικές γωνίες της μετρικής οι κλάσεις A και B πρέπει να έχουν διαφορετικές τιμές. Δεν υπάρχουν άλλα προτεινόμενα μοντέλα για την DIT μετρική εξαιτίας αυτής της διφορούμενης κατάστασης. Υπάρχει όμως μία άλλη μετρική που μας αποσαφηνίζει την κατάσταση κι αυτή είναι η NAC (Number of Ancestor Classes). Υπολογίζει τον αριθμό των πρωτόγονων κλάσεων και έχει την ίδια θεωρητική βάση και τις ίδιες οπτικές γωνίες με την DIT (Chidamber και Kemerer, 1994).

Όπως φαίνεται στο παρακάτω παράδειγμα κώδικα, η κλάση Location δεν κληρονομεί από καμία κλάση επομένως η DIT είναι μηδέν. Η Point κληρονομεί από την Location επομένως η DIT είναι ένα. Στην περίπτωση της Circle που κληρονομεί από την Location η DIT είναι δύο αφού κληρονομεί σε δεύτερο επίπεδο ιεραρχίας.

```
package Location is

   type Instance is tagged private;

   procedure Initialize(The_Location : in out Instance;
                        Init_X       : in     Integer;
                        Init_Y       : in     Integer);

   function X_Coordinate_Of(The_Location : in Instance) return Integer;

   function Y_Coordinate_Of(The_Location : in Instance) return Integer;

private

   type Instance is tagged
   record
     X : Integer := 0;
     Y : Integer := 0;
   end record;

end Location;

--| Body of Location
package body Location is

   procedure Initialize(The_Location : in out Instance;
                        Init_X       : in     Integer;
                        Init_Y       : in     Integer) is
   begin
     The_Location.X := Init_X;
     The_Location.Y := Init_Y;
   end Initialize;

   function X_Coordinate_Of(The_Location : in Instance) return Integer is
   begin
     return The_Location.X;
   end X_Coordinate_Of;

   function Y_Coordinate_Of(The_Location : in Instance) return Integer is
   begin
     return The_Location.Y;
   end Y_Coordinate_Of;

end Location;
```

```
-- class Point
-----------------------------------------------------------------------
package Location.Point is

   type Instance is new Location.Instance with private;

   procedure Initialize(The_Point : in out Instance;
                        Init_X    : in     Integer;
                        Init_Y    : in     Integer);

   procedure Show (The_Point : in out Instance);

   procedure Hide (The_Point : in out Instance);

   procedure Drag (The_Point : in out Instance;
                   By        : in     Integer);

   function Is_Visible (The_Point : Instance) return Boolean;

   procedure Move (The_Point : in out Instance;
                   New_X     : in     Integer;
                   New_Y     : in     Integer);

private

   type Instance is new Location.Instance with
   record
     Visible : Boolean := False;
   end record;

end Location.Point;

--| Body of Point

with Graphics;
package body Location.Point is

   procedure Initialize(The_Point : in out Instance;
                        Init_X    : in     Integer;
                        Init_Y    : in     Integer) is
   begin
     Initialize(Location.Instance(The_Point), Init_X, Init_Y);
     The_Point.Visible := False;
   end Initialize;

   procedure Show (The_Point : in out Instance) is
   begin
     The_Point.Visible := True;
     Graphics.Put_Pixel(The_Point.X, The_Point.Y, Graphics.Current_Color);
   end Show;

   procedure Hide (The_Point : in out Instance) is
   begin
     The_Point.Visible := False;
     Graphics.Put_Pixel(The_Point.X, The_Point.Y, Graphics.Background_Color);
   end Hide;

   procedure Drag (The_Point : in out Instance;
                   By        : in     Integer) is
     Delta_X, Delta_Y   : Integer;
     Figure_X, Figure_Y : Integer;
   begin
     Show(The_Point);
     Figure_X := X_Coordinate_Of(Location(The_Point));
     Figure_Y := Y_Coordinate_Of(Location(The_Point));
     while Graphics.Delta_Of(Delta_X, Delta_Y) loop
       Figure_X := Figure_X + (Delta_X * By);
       Figure_Y := Figure_Y + (Delta_Y * By);
       Move(The_Point, Figure_X, Figure_Y);
     end loop;
   end Drag;

   function Is_Visible (The_Point : Instance) return Boolean is
   begin
     return The_Point.Visible;
   end Is_Visible;

   procedure Move (The_Point : in out Instance;
                   New_X     : in     Integer;
                   New_Y     : in     Integer) is
   begin
     Hide(The_Point);
     The_Point.X := New_X;
     The_Point.Y := New_Y;
     Show(The_Point);
   end Move;

end Location.Point;
```

```
package Location.Point.Circle is

  type Instance is new Location.Point.Instance with private;

  procedure Initialize(The_Circle : in out Instance;
                       Init_X     : in     Integer;
                       Init_Y     : in     Integer;
                       Init_Radius: in     Integer);

  procedure Show(The_Circle : in out Instance);

  procedure Hide(The_Circle : in out Instance);

  procedure Contract(The_Circle : in out Instance;
                     By         : in     Integer);

  procedure Expand  (The_Circle : in out Instance;
                     By         : in     Integer);

private

  type Instance is new Location.Point.Instance with
  record
    Radius : Integer;
  end record;

end Location.Point.Circle;

--| Body for Circle

with Graphics;
package body Location.Point.Circle is

  procedure Initialize(The_Circle : in out Instance;
                       Init_X     : in     Integer;
                       Init_Y     : in     Integer;
                       Init_Radius: in     Integer) is

  begin
    Initialize(Point.Instance(The_Circle), Init_X, Init_Y);
    The_Circle.Radius := Init_Radius;
  end Initialize;

  procedure Show(The_Circle : in out Instance) is
  begin
    The_Circle.Visible := True;
    Graphics.Draw_Circle(The_Circle.X, The_Circle.Y, The_Circle.Radius);
  end Show;

  procedure Hide(The_Circle : in out Instance) is
    Temp_Color : Integer;
  begin
    Temp_Color := Graphics.Current_Color;
    Graphics.Set_Color(Graphics.Background_Color);
    The_Circle.Visible := False;
    Graphics.Draw_Circle(The_Circle.X, The_Circle.Y, The_Circle.Radius);
    Graphics.Set_Color(Temp_Color);
  end Hide;

  procedure Expand(The_Circle : in out Instance;
                   By         : in     Integer) is
  begin
    Hide(The_Circle);
    The_Circle.Radius := The_Circle.Radius + By;
    if The_Circle.Radius < 0 then
      The_Circle.Radius := 0;
    end if;
    Show(The_Circle);
  end Expand;

  procedure Contract(The_Circle : in out Instance;
                     By         : in     Integer) is
  begin
    Expand(The_Circle, -(By));
  end Contract;

end Location.Point.Circle;
```

Μετρικές συνοχής: Cohesion Metrics

LCOM (Lack of Cohesion Of Methods)
Έλλειψη συνοχής των μεθόδων

Σε ότι αφορά τις σχετιζόμενες με την συνοχή μετρικές έχουν προταθεί διάφοροι ορισμοί. Από ένα πλήθος οκτώ τέτοιων μετρικών προέκυψαν τα εξής ονόματα:

#1 LCOM1

#2 LCOM ή LCOM2 ή PLCOM1

#3 LCOM3 ή PLCOM2

#4 LCOM4

#5 LCOM5

#6 Coh

#7 TCC

#8 LCC

Όσες μετρικές εμφανίζονται να έχουν πολλαπλά ονόματα, οφείλεται στο ότι υλοποιήθηκαν με διαφορετικά εργαλεία κάθε φορά. Οι μετρικές από #1 - #4 υπολογίστηκαν βασιζόμενες στα ζευγάρια των μεθόδων μιας κλάσης που αναφέρονται από κοινού σε μια ιδιότητα (attribute) χρησιμοποιώντας κλήση με αναφορά (call by reference). Οι μετρικές #7 και #8 μετρούν τον αριθμό των ζευγαριών των μεθόδων μιας κλάσης οι οποίες αναφέρονται από κοινού σε μια ιδιότητα μιας κλάσης χρησιμοποιώντας κλήση με τιμή (call by value) είτε κλήση με αναφορά (call by reference). Η μετρική #5 βασίζεται απλά στις μεθόδους μιας κλάσης που αναφέρονται στις ιδιότητες μιας κλάσης. Οι μετρικές #6-#8 κάνουν απευθείας υπολογισμό της συνοχής, ενώ οι #1-#5 υπολογίζουν την συνοχή βασιζόμενες στην έλλειψη αυτής.

Η ανεξαρτησία μετριέται με την χρήση δύο ποιοτικών κριτηρίων: τη συνοχή και την σύζευξη. Η συνοχή είναι μία μέτρηση της σχετικής λειτουργικής αντοχής τμημάτων του κώδικα, με την σύζευξη θα ασχοληθούμε αμέσως μετά.

Η συνοχή είναι φυσική προέκταση της απόκρυψης πληροφοριών (hiding concept) που περιέγραψα στο εισαγωγικό κείμενο. Ένα συνεκτικό τμήμα κώδικα εκτελεί μία και μόνη εργασία στο πλαίσιο μιας διαδικασίας λογισμικού, απαιτώντας μικρή αλληλεπίδραση με τις διαδικασίες που διεξάγονται σε άλλα μέρη του προγράμματος.

Με απλά λόγια, ένα συνεκτικό τμήμα κώδικα (module) θα πρέπει (το ιδανικό είναι) να κάνει μόνο ένα πράγμα. Η συνοχή, μπορεί να αναπαρασταθεί σαν ένα «φάσμα». Εμείς προσπαθούμε πάντα για την υψηλή συνοχή, παρόλο που και η μεσαία είναι συχνά αποδεκτή. Η κλίμακα για τη συνοχή είναι γραμμική. Δηλαδή, η χαμηλή συνοχή είναι πολύ «χειρότερη» από την μεσαία, η οποία είναι σχεδόν το ίδιο "καλή", όπως και η υψηλή συνοχή. Στην πράξη, ο σχεδιαστής δεν χρειάζεται να ασχολείται με την κατηγοριοποίηση της συνοχή σε συγκεκριμένα τμήματα κώδικα. Αντίθετα, η γενική ιδέα πρέπει να γίνει κατανοητή και τα χαμηλά επίπεδα της συνοχής θα πρέπει να αποφεύγονται κατά την σχεδίαση τμημάτων κώδικα.

Κατά τη χαμηλή (ανεπιθύμητη) άκρη του «φάσματος», συναντούμε ένα τμήμα του κώδικα που εκτελεί μια σειρά από εργασίες που δεν σχετίζονται μεταξύ τους στενά ή ακόμα και καθόλου. Αυτά τα τμήματα κώδικα ονομάζονται συμπτωματικά συνεκτικά (coincidentally cohesive). Ένα τμήμα κώδικα που εκτελεί τις εργασίες του, οι οποίες είναι λογικά συσχετιζόμενες (π.χ. ένα τμήμα κώδικα που παράγει όλες τις εξόδους του ανεξάρτητα τύπου) είναι λογικά συνεκτικό (logically cohesive). Όταν ένα τμήμα κώδικα περιλαμβάνει εργασίες που σχετίζονται με το γεγονός ότι όλες πρέπει να εκτελούνται στο ίδιο χρονικό διάστημα, το τμήμα κώδικα θεωρείται πρόσκαιρα συνεκτικό(temporal cohesion).

Μέτρια επίπεδα συνοχής είναι συσχετισμένα στενά το ένα στο άλλο ως προς το βαθμό της ανεξαρτησίας του τμήματος του κώδικα. Όταν επεξεργαζόμαστε στοιχεία ενός τμήματος κώδικα που σχετίζονται και πρέπει να εκτελεστούν σε μια συγκεκριμένη σειρά, τότε υφίσταται διαδικαστική συνοχή (procedural cohesion). Όταν όλα τα στοιχεία που επεξεργαζόμαστε συγκεντρώνονται σε μια περιοχή μιας δομής δεδομένων, τότε υφίσταται επικοινωνιακή συνοχή (communicational cohesion).

Οι μετρικές συνοχής ελέγχουν τις μεθόδους μιας κλάσης κατά πόσο καλά είναι συνδεδεμένες μεταξύ τους. Μία συνεκτική κλάση εκτελεί μία λειτουργία. Μία μη συνεκτική κλάση δύο ή περισσότερες μη συσχετιζόμενες λειτουργίες και ενδεχομένως να χρειάζεται να ανακατασκευαστεί σε δύο ή περισσότερες μικρότερες κλάσεις.

Η ένδεια συνοχής (Lack of cohesion) σημαίνει ότι η κλάση εκτελεί περισσότερες από μία λειτουργίες. Αυτό δεν είναι επιθυμητό. Εάν η κλάση εκτελεί ένα πλήθος μη συσχετιζόμενων λειτουργιών, θα πρέπει να διαχωριστεί. Η υψηλή συνοχή είναι επιθυμητή, από την στιγμή που προάγει την ενθυλάκωση. Μειονέκτημα αποτελεί ότι μία υψηλής συνοχής κλάση έχει υψηλή σύζευξη μεταξύ των μεθόδων της που έχει ως συνέπεια πολλές και μεγάλες δοκιμασίες (testing) της εν λόγω κλάσης. Χαμηλή συνοχή υποδεικνύει ακατάλληλη σχεδίαση και μεγάλη πολυπλοκότητα. Επιπλέον έχει παρατηρηθεί ότι υποδεικνύει υψηλές περιοχές σφαλμάτων.

Ένδεια συνοχής των μεθόδων (LCOM). Κάθε μέθοδο μέσα σε μια κλάση, C, έχε πρόσβαση σε ένα ή περισσότερα χαρακτηριστικά. Η LCOM είναι ο αριθμός των μεθόδων που έχει πρόσβαση σε ένα ή περισσότερα από τα ίδια χαρακτηριστικά. Σε περίπτωση που δεν υπάρχει πρόσβαση σε μεθόδους με τα ίδια χαρακτηριστικά, τότε η LCOM = 0. Για να γίνει κατανοητή η περίπτωση κατά την οποία η LCOM $\neq$ 0, υποθέστε μία κλάση με έξι μεθόδους. Τέσσερις από τις μεθόδους έχουν ένα ή περισσότερα ίδια χαρακτηριστικά (δηλαδή, έχουν πρόσβαση σε κοινά χαρακτηριστικά). Ως εκ τούτου, η LCOM = 4. Αν η LCOM είναι υψηλή, οι μέθοδοι μπορούν να συνδεθούν μεταξύ τους μέσω των ιδιοτήτων τους. Αυτό αυξάνει την

πολυπλοκότητα του σχεδιασμού της κλάσης. Σε γενικές γραμμές, υψηλές τιμές της LCOM σημαίνει ότι η κλάση θα ήταν καλύτερα να διασπαστεί σε δύο ή περισσότερες ξεχωριστές κλάσεις. Although there are cases in which a high value for LCOM is justifiable, it is desirable to keep cohesion high; that is, keep LCOM low.

LCOM4 (Hitz & Montazeri)

Η LCOM4 είναι η μετρική της ένδειας της συνοχής που προτείνεται για προγράμματα σε Visual Basic. Η LCOM4 μετράει το πλήθος των συνδεδεμένων συστατικών σε μια κλάση. Συνδεδεμένο συστατικό είναι ένα σύνολο συσχετιζόμενων μεθόδων (και μεταβλητών σε επίπεδο κλάσης). Πρέπει να υπάρχει μόνο ένα τέτοιο συστατικό σε κάθε μία κλάση. Εάν υπάρχουν δύο ή περισσότερα συστατικά, η κλάση πρέπει να διαχωριστεί σε τόσες μικρότερες κλάσεις.

Ποιές μέθοδοι είναι συσχετιζόμενες; Οι μέθοδοι α και β είναι συσχετιζόμενες εάν:

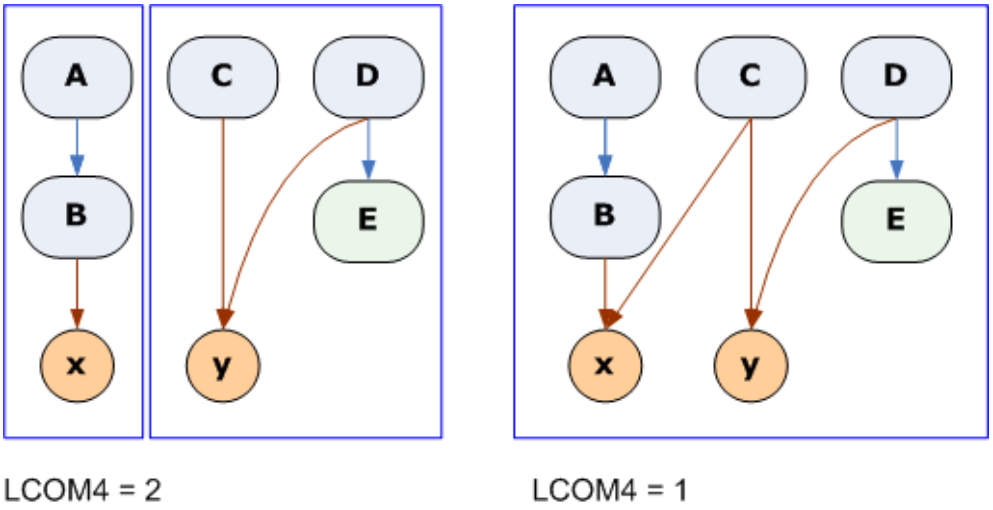
- Και οι δύο έχουν πρόσβαση στην ίδια επιπέδου κλάσης μεταβλητή, ή
- Η α καλεί την β, ή η β καλεί την α.

Μετά τον καθορισμό των συσχετιζόμενων μεθόδων, ενδείκνυται ένα γράφημα Σχήμα_XXI που συνδέει τις συσχετιζόμενες μεθόδους μεταξύ τους. Η LCOM4 ισούται με τον αριθμό των συνδεδεμένων συνόλων των μεθόδων.

LCOM4 =1 υποδεικνύει μία συνεκτική κλάση, δηλαδή μία «καλή» κλάση

LCOM4 >=2 υποδεικνύει πρόβλημα. Η κλάση πρέπει να διαχωριστεί σε τόσες μικρότερες κλάσεις.

LCOM4 = 0 συμβαίνει όταν δεν υπάρχουν μέθοδοι σε μια κλάση. Αυτή είναι επίσης μία «κακή» κλάση.



Σχήμα_XXI

Το παράδειγμα στα αριστερά δείχνει μία κλάση που περιέχει μεθόδους από την A έως την E και μεταβλητές την x και την y. Η A καλεί την B και access στην μεταβλητή x. Οι C και D μαζί access στην y. Η D καλεί την E, αλλά δεν access σε καμία μεταβλητή. Αυτή η κλάση αποτελείται από δύο μη συσχετιζόμενα συστατικά (LCOM4=2) επομένως πρέπει να διαχωριστεί σε {A, B, x} και {C, D, E, y}.	Στο παράδειγμα στα δεξιά έγινε το C έτσι ώστε να access το x για να αυξηθεί η συνοχή. Τώρα η κλάση περιέχει ένα μόνο συστατικό (LCOM4=1) και αποτελεί μία συνεκτική κλάση.

LCOM1, LCOM2 και LCOM3

Οι LCOM1, LCOM2 και LCOM3 δεν είναι τόσο κατάλληλες για την Visual Basic όπως η LCOM4. Είναι λιγότερο ακριβείς, κυρίως δεδομένου ότι δεν υπολογίζουν την επίδραση των accessors και των καλούμενων διαδικασιών, οι οποίες χρησιμοποιούνται συχνά για να προσβάσουν τιμές των μεταβλητών σε ένα συνεκτικό τρόπο. Μπορούν να είναι πιο κατάλληλες σε άλλες αντικειμενοστραφείς γλώσσες όπως η C + +. Αναφέρω αυτές τις μετρήσεις για λόγους πληρότητας. Μπορούν να χρησιμοποιούνται ως συμπληρωματικές μετρήσεις εκτός της LCOM4.

LCOM1 (Chidamber & Kemerer)

Η LCOM1 παρουσιάστηκε στην ομάδα μετρικών των Chidamber & Kemerer Ονομάζεται επίσης LCOM ή LOCOM, και υπολογίζεται ως εξής:

Λάβετε κάθε ζεύγος μεθόδων της κλάσης. Αν έχουν πρόσβαση σε μη συνδεδεμένα σύνολα μεταβλητών, αυξήστε το P κατά ένα. Εάν μοιράζονται τουλάχιστον μία μεταβλητή, όσον αφορά της πρόσβαση, αυξήστε το Q κατά ένα.

LCOM1 = P - Q, αν P> Q LCOM1 = 0 αλλιώς

LCOM1 = 0 δείχνει μια συνεκτική τάξη.

LCOM1> 0 σημαίνει ότι η κλάση χρειάζεται ή μπορεί να χωριστεί σε δύο ή περισσότερες κλάσεις, δεδομένου ότι οι μεταβλητές της, ανήκουν σε μη συνδεδεμένα σύνολα.

Κλάσεις με υψηλή LCOM1 διαπιστώθηκε ότι είναι επιρρεπείς σε σφάλματα.

Μια υψηλή τιμή της LCOM1 δηλώνει ανομοιότητα στη λειτουργία που παρέχεται από την κλάση. Αυτή η μετρική μπορεί να χρησιμοποιηθεί για την αναγνώριση κλάσεων οι οποίες επιχειρούν να επιτύχουν πολλούς διαφορετικούς στόχους και κατά συνέπεια, είναι πιθανό να συμπεριφέρονται με λιγότερο προβλέψιμο τρόπο από τις κλάσεις που έχουν χαμηλότερο LCOM1. Οι εν λόγω κλάσεις θα μπορούσαν να είναι πιο επιρρεπής σε λάθη, πιο δύσκολες για έλεγχο και θα μπορούσαν ενδεχομένως να διαχωριστεί σε δύο ή περισσότερες κλάσεις καθιστά καλύτερα ορισμένη την συμπεριφορά τους. Η LCOM1 μπορεί να χρησιμοποιηθεί από ειδικούς σχεδιαστές και υπευθύνους έργων (project managers) ως ένα σχετικά απλό τρόπο για να παρακολουθούν αν η τιμή της συνοχής είναι επιθυμητή για το σχεδιασμό της εφαρμογής τους και να την συμβουλεύονται για επικείμενες αλλαγές.

Η LCOM1 φαίνεται να έχει πληθώρα ελαττωμάτων επομένως θα πρέπει να χρησιμοποιείται με προσοχή.

Πρώτον η LCOM1 δίνει τιμή μηδέν σε πολλές διαφορετικές κλάσεις. Για να ξεπεραστεί αυτό το πρόβλημα, έχουν προταθεί νέες μετρικές, οι LCOM2 και LCOM3.

Δεύτερον, Gupta είχε προτείνει ότι η LCOM1 δεν είναι έγκυρος τρόπος για να μετρηθεί η συνοχής μιας κλάσης. Αυτό επειδή ο ορισμός της είναι βασισμένος στην αλληλεπίδραση μεθόδων - δεδομένων, κι αυτό δεν αποτελεί τον σωστό τρόπο για να καθοριστεί η συνοχή στον αντικειμενοστραφή κόσμο.

Τρίτον, επειδή η LCOM1 ορίζεται σε προσβάσιμες μεταβλητές, δεν είναι κατάλληλη για τις κλάσεις όπου έχουν εσωτερική πρόσβαση στα δεδομένα τους μέσω των ιδιοτήτων τους. Μία κλάση που παίρνει / θέτει (gets/sets) τα δικά της εσωτερικά δεδομένα μέσω των δικών της ιδιοτήτων, και όχι μέσω των άμεσων μεταβλητών ανάγνωσης / εγγραφής, είναι δυνατόν να παρουσιάζει υψηλή LCOM1. Αυτό δεν είναι ένδειξη μιας προβληματικής κλάσης απλά η LCOM1 δεν είναι κατάλληλη για τη μέτρηση των εν λόγω κλάσεων.

LCOM2 and LCOM3 (Henderson-Sellers, Constantine & Graham)

Για να ξεπεραστούν τα προβλήματα της LCOM1, δύο επιπλέον μετρικές έχουν προταθεί: οι LCOM2 και LCOM3.

Οι χαμηλές τιμές των LCOM2 και LCOM3 δείχνουν υψηλή συνοχή και μία καλά σχεδιασμένη κλάση. Φαίνεται ότι το σύστημα έχει καλά διαχωρισμένες τις κλάσεις του που συνεπάγεται απλότητα και μεγάλη δυνατότητα επαναχρησιμοποίησης. Μια συνεκτική κλάση τείνει να παρέχει υψηλό βαθμό ενθυλάκωσης.

Αποτελεί στιλιστικό θέμα η επιλογή της LCOM2 ή της LCOM3. Οι LCOM2 και LCOM3 εφαρμόζουν τις ίδιες μετρήσεις με διαφορετικές φόρμουλες. Η LCOM3 κυμαίνεται στο διάστημα [0,1], ενώ η LCOM2 είναι στην περιοχή [0,2].

LCOM2> = 1 δείχνει μια πολύ προβληματική κλάση.

Η LCOM3 δεν έχει ενιαία κατώτατη τιμή.

Μια καλή ιδέα είναι να αφαιρέσουμε τυχόν νεκρές μεταβλητές πριν την ερμηνεία των LCOM2 ή LCOM3. Νεκρές μεταβλητές μπορούν να οδηγήσουν σε υψηλές LCOM2 και LCOM3, οδηγώντας έτσι σε λάθος ερμηνείες του τι πρέπει να γίνει.

Πίνακας_V

Ορισμοι που χρησιμοποιουνται στις LCOM2 και LCOM3	
m	Αριθμός των μεθόδων της κλάσης
a	Αριθμός των μεταβλητών της κλάσης
mA	Αριθμός των μεθόδων που έχουν πρόσβαση σε μεταβλητές της κλάσης
sum(mA)	Άθροισμα του mA επί των γνωρισμάτων της κλάσης

Λεπτομέρειες εκτέλεσης

Το m είναι ίσο με το WMC.

Το a περιέχει όλες τις μεταβλητές είτε επιμερίζονται είτε όχι. Όλες οι προσβάσεις σε μια μεταβλητή υπολογίζονται.

LCOM2

LCOM2 = 1 - sum (mA) / (m * a)

Η LCOM2 ισούται με το ποσοστό των μεθόδων που δεν έχουν πρόσβαση σε συγκεκριμένες ιδιότητες, οι οποίες αποτελούν μέσους όρους των ιδιοτήτων της κλάσης. Εάν ο αριθμός των μεθόδων ή των ιδιοτήτων είναι μηδέν, η LCOM2 είναι απροσδιόριστη και εμφανίζεται ως μηδενική.

LCOM3 ή αλλιώς LCOM*

LCOM3 = (m - sum(mA)/a) / (m-1)

Η LCOM3 κυμαίνεται μεταξύ 0 και 2. Τιμές 1 .. 2 θεωρούνται ανησυχητικές.

Σε μια κανονική κλάση της οποίας οι μέθοδοι έχουν πρόσβαση σε δικές της μεταβλητές, η LCOM3 κυμαίνεται από 0 (υψηλή συνοχή) έως 1 (καμία συνοχή). Όταν η LCOM3 = 0, κάθε μέθοδος έχει πρόσβαση σε όλες τις μεταβλητές. Αυτό δείχνει την μεγαλύτερη δυνατή συνοχή. Όταν η LCOM3 = 1 δείχνει μια ακραία έλλειψη συνοχής. Στην περίπτωση αυτή, η κλάση πρέπει να διαχωριστεί.

Όταν υπάρχουν μεταβλητές που δεν είναι προσβάσιμες από μεθόδους της κλάσης, 1 <LCOM3 <= 2. Αυτό συμβαίνει αν οι μεταβλητές είναι νεκρές ή έχουν πρόσβαση μόνο εκτός της κλάσης. Και οι δύο περιπτώσεις αποτελούν ελαττώματα σχεδιασμού. Η κλάση είναι υποψήφια για να ξαναγραφεί ως τμήμα του κώδικα. Εναλλακτικά, οι μεταβλητές της κλάσης θα πρέπει να ενθυλακωθούν με προσβάσιμες μεθόδους ή ιδιότητες. Μπορεί επίσης να υπάρχουν κάποιες νεκρές μεταβλητές προς αφαίρεση.

Αν δεν υπάρχουν περισσότερες από μία μέθοδοι σε μία κλάση, η LCOM3 είναι αδιευκρίνιστη. Εάν δεν υπάρχουν μεταβλητές σε μια κλάση, η LCOM3 είναι και πάλι αδιευκρίνιστη. Μια απροσδιόριστη LCOM3 εμφανίζεται ως μηδέν.

Μελέτη των μετρικών συνοχής, πειράματα – σχολιασμός αποτελεσμάτων

Για την εξαγωγή χρήσιμων συμπερασμάτων έγιναν κάποια πειράματα με χρήση κώδικα C++. Για την υλοποίηση των μετρικών χρησιμοποιήθηκαν τα εξής τρία εργαλεία:

• HYSS

• GEN++ εργαλείο μέτρησης

• Patricia/test metrics

Σύγκριση Μετρικών με Ομάδες Προγραμματιστών

Σε μια πρώτη φάση επιλέχθηκε, τα αποτελέσματα που θα δώσουν οι μετρικές να συγκριθούν με τις εκτιμήσεις που θα έδιναν δύο ομάδες προγραμματιστών καθώς θα προσπαθούσαν να κρίνουν την έννοια της συνοχής κάθε κλάσης από την δική τους σκοπιά. Η μόνη διαφορά μεταξύ των ομάδων αυτών είναι ότι η δεύτερη ομάδα θεωρείται περισσότερο έμπειρη στην χρήση object-oriented συστημάτων. Στην συνέχεια και με την χρήση του συντελεστή συσχέτισης του Pearson (Pearson’s correlation coefficient) έγινε μια αξιολόγηση για το πόσο σχετικά είναι τα αποτελέσματα μεταξύ των μετρικών και των ομάδων.

Από τα αποτελέσματα και τις συσχετίσεις που έγιναν φαίνεται να προκύπτει ότι οι μετρικές και οι ομάδες προγραμματιστών μετρούν το ίδιο είδος και ποιότητα συνοχής. Οι ομάδες συμφωνούν μεταξύ τους σε αρκετές μετρικές με τις LCOM και LCC να εμφανίζουν την μεγαλύτερη συσχέτιση. Η μετρική που εμφάνισε την μικρότερη συσχέτιση με τις δύο ομάδες ήταν η TCC. Τέλος το γεγονός ότι η δεύτερη ομάδα ήταν πιο έμπειρη από την πρώτη δεν φαίνεται να παίζει τόσο σημαντικό ρόλο στην σύγκριση και συσχέτιση των μετρήσεων.

Σύγκριση των μετρικών μεταξύ τους

Στην επόμενη μελέτη που πραγματοποιήθηκε, σκοπός ήταν η μεταξύ τους σύγκριση των μετρικών, έτσι ώστε να φανεί αν υπάρχει κάποια συσχέτιση στα αποτελέσματα που αυτές δίνουν. Σε περίπτωση ύπαρξης συσχέτισης θα ήταν αντιληπτό ότι οι μετρικές μετρούν το ίδιο είδος συνοχής με αποτέλεσμα να μην υπάρχει προβληματισμός για τα ποιά θα ήταν η καταλληλότερη και η πιο αξιόπιστη. Από τις συσχετίσεις που έγιναν δυστυχώς προέκυψε ότι δεν υπάρχει κάποια μετρική που να μπορεί να «συμφωνεί» με τα αποτελέσματα των υπολοίπων. Αντιθέτως υπήρχε ένα αρκετά μεγάλο εύρος συσχετίσεων, γεγονός που φανέρωνε ότι υπήρχαν μετρικές των οποίων τα αποτελέσματα συμφωνούσαν απόλυτα, αλλά και κάποιες άλλες με τις οποίες δεν υπήρχε καμία σχέση. Μεγάλη συσχέτιση παρατηρήθηκε κυρίως μεταξύ των LCOM1, LCOM2, LCOM3, LCOM4 μιας και έχουν μικρές διαφορές στους μεταξύ τους ορισμούς. Η LCOM μετρική φάνηκε να έχει πολύ μικρή σχέση μόνο με τις LCOM1-3. Τέλος η TCC και η LCC εμφάνισαν μεγάλη συσχέτιση μεταξύ τους.

Ανάλυση κυριότερων συστατικών

Σε μια τελευταία μελέτη που έγινε για τις μετρικές και για το πόσο μπορεί να συσχετίζονται μεταξύ τους, χρησιμοποιήθηκε η μαθηματική ανάλυση κυριότερων συστατικών (Principal Component) σύμφωνα με την οποία μπορούμε να περιγράψουμε ένα σύνολο από μεταβλητές με την βοήθεια μικρότερων και γραμμικά εξαρτημένων μεταξύ τους συνόλων. Τα μικρότερα σύνολα αποτελούν τις γνωστές κύριες συνιστώσες του αρχικού συνόλου. Από την ανάλυση που έγινε προέκυψαν τέσσερις κύριες συνιστώσες οι οποίες και θεωρήθηκε ότι έχουν την μεγαλύτερη συνεισφορά στην περιγραφή του αρχικού συνόλου. Το γεγονός ότι η κύρια συνιστώσα μιας μετρικής μπορεί να έχει ίδια χαρακτηριστικά με την αντίστοιχη συνιστώσα μιας άλλης μετρικής υποδηλώνει ότι οι μετρικές αυτές εμφανίζουν κάποια συσχέτιση μεταξύ τους. Από την αναζήτηση που έγινε στα αποτελέσματα της ανάλυσης προέκυψε οι εξής ομάδες μετρικών βασισμένες στην μεταξύ τους συσχέτιση:

- LCOM1-LCOM4
- LCOM5, Coh, LCC, TCC
- PLCOM1, PLCOM2

Τέλος η LCOM δεν φάνηκε να συσχετίζεται με καμία μετρική

TCC και LCC — Άρρηκτη και Λιτή συνοχή κλάσεων (Tight and Loose Class Cohesion)

TCC και LCC μετρικές: Άρρηκτη και Λιτή συνοχή κλάσεων. Αυτή η ομάδα των μετρικών στοχεύει στην αποσαφήνιση της διαφοράς της καλής και της κακής συνοχής. Σε αυτές τις μετρικές, οι μεγάλες τιμές είναι επιθυμητές.

Οι μετρικές TCC (Tight Class Cohesion) και LCC (Loose Class Cohesion) παρέχουν έναν άλλο τρόπο για να μετρηθεί η συνοχή μιας κλάσης. Οι TCC και LCC συσχετίζονται με την ιδέα της LCOM4, ακόμα και αν έχουν μερικές διαφορές. Όσο υψηλότερες είναι οι TCC και LCC, τόσο πιο συνεκτικές άρα και καλές είναι οι κλάσεις.

Για τις TCC και LCC υπολογίζουμε μόνο τις ορατές μεθόδους (ενώ οι LCOMx εξετάσουν όλες τις μεθόδους). Μια μέθοδος είναι ορατή αν δεν είναι ιδιωτική (private). Μια μέθοδος είναι ακόμα ορατή και αν εφαρμόζει μια διεπαφή ή χειρίζεται ένα γεγονός. Κατά τα άλλα, έχουμε τον ίδιο ορισμό με την LCOM4.

Οι Μέθοδοι a και b σχετίζονται, εάν:

1. Και οι δύο έχουν πρόσβαση στις ίδιες μεταβλητής της κλάσης, ή
2. Καλούν ιεραρχικά ξεκινώντας από a και b και έχουν πρόσβαση στις ίδιες μεταβλητές της κλάσης. Για τις ιεραρχικές κλήσεις υπολογίζουμε όλες τις διαδικασίες εσωτερικά της κλάσης, συμπεριλαμβάνοντας και τις ιδιωτικές. Εάν η κλήση μεταφέρεται εξωτερικά της κλάσης, σταματάμε να ακολουθούμε την κλήση.

Όταν δύο μέθοδοι σχετίζονται με αυτόν τον τρόπο, ονομάζονται άμεσα συνδεδεμένες (directly connected). Όταν δύο μέθοδοι δεν είναι άμεσα συνδεδεμένες αλλά συνδέονται μέσω άλλων μεθόδων, ονομάζονται έμμεσα συνδεδεμένες (indirectly connected).

Ορισμοί TCC και LCC

NP = μέγιστος αριθμός πιθανών συνδέσεων = N * (N - 1) / 2 όπου N το πλήθος των μεθόδων

NDC = αριθμός άμεσων συνδέσεων (αριθμός ακμών στο γράφημα)

NID = αριθμός έμμεσων συνδέσεων

Άρρηκτη συνοχή κλάσης TCC = NDC/NP

Λιτή συνοχή κλάσης LCC = (NDC+NIC) / NP

Η TCC είναι στο εύρος 0 .. 1.

Η LCC είναι στο εύρος 0 .. 1.

TCC <= LCC.

Όσο υψηλότερές είναι οι TCC και LCC, τόσο πιο συνεκτικές είναι οι κλάσεις

Αποδεκτές και μη αποδεκτές τιμές

Σύμφωνα με τους Bieman & Kang, TCC <0,5 και LCC <0,5 θεωρούνται μη συνεκτικές κλάσεις. LCC = 0,8 θεωρείται αρκετά συνεκτική. TCC = LCC = 1 είναι η μέγιστη συνοχή κλάσης, που ερμηνεύεται ότι όλες οι μέθοδοι είναι συνδεδεμένες.

Η LCC μας δίνει τη συνολική συνοχή. Εξαρτάται από τον αριθμό των μεθόδων και του τρόπου σύνδεσης των.

• Όταν η LCC = 1, όλες οι μέθοδοι στην κλάση είναι συνδεδεμένες, είτε άμεσα είτε έμμεσα. Αυτή είναι η συνεκτική περίπτωση.

• Όταν η LCC <1, υπάρχουν 2 ή περισσότερες ομάδες με μη συνδεδεμένες μέθοδοι. Αυτή η κλάση δεν είναι συνεκτική. Μπορεί να χρειάζεται να επανεξεταστούν αυτές οι κλάσεις για να δούμε γιατί είναι έτσι. Οι μέθοδοι μπορεί να είναι έτσι επειδή έχουν πρόσβαση σε μεταβλητές που δεν είναι σε κλάση ή επειδή έχουν πρόσβαση σε τελείως διαφορετικές μεταβλητές.

• Όταν η LCC = 0, όλες οι μέθοδοι είναι τελείως ασύνδετες. Αυτή είναι η μη συνεκτική περίπτωση.

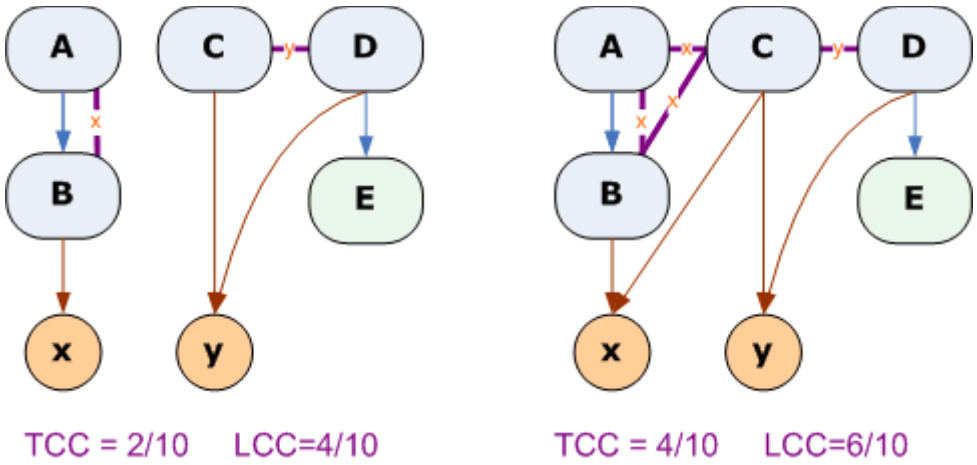
Η TCC μας δίνει την "πυκνότητα της σύνδεσης", (από την στιγμή που η LCC, επηρεάζεται μόνο αν όλες οι μέθοδοι είναι συνδεδεμένες).

• Όταν TCC = LCC = 1 είναι η μέγιστη συνεκτική κλάση όπου όλες οι μέθοδοι συνδέονται άμεσα μεταξύ τους.

• Όταν TCC = LCC <1, όλες οι υφιστάμενες συνδέσεις είναι άμεσες (έστω και αν δεν είναι όλες οι μέθοδοι συνδεδεμένες).

• Όταν TCC <LCC, η «πυκνότητα σύνδεσης" είναι χαμηλότερη από ό, τι θα μπορούσε να είναι θεωρητικά. Δεν είναι όλες οι μέθοδοι άμεσα συνδεδεμένες μεταξύ τους. Για παράδειγμα, η A & B είναι συνδεδεμένες με τη βοήθεια της μεταβλητής x και B και C με τη βοήθεια της μεταβλητής y. Οι A και C δεν μοιράζονται μια μεταβλητή, αλλά είναι έμμεσα συνδεδεμένες μέσω της B.

• Όταν η TCC = 0 (και η LCC = 0), η κλάση είναι τελείως μη συνεκτική και όλες οι μέθοδοι δεν είναι καθόλου συνδεδεμένες.



Σχήμα_XXII

Το παράδειγμα αυτό δείχνει την ίδια κλάση που χρησιμοποιήθηκε και προηγουμένως. Οι συνδέσεις σημειώνονται με έντονες μοβ γραμμές. Οι A και B είναι συνδεδεμένες μέσω της μεταβλητής x. Οι C και D συνδέονται μέσω της μεταβλητή y. Η E δεν είναι συνδεδεμένη, επειδή το δέντρο σταματάει σ'αυτήν και δεν συνεχίζει. Υπάρχουν 2 άμεσες («άρρηκτες») συνδέσεις. Δεν υπάρχουν επιπρόσθετες έμμεσες συνδέσεις σ'αυτήν την περίπτωση.	Στο παράδειγμα, στα δεξιά, κάναμε την C να έχει πρόσβαση στην x για να αυξήσουμε την συνοχή. Τώρα (A, B, C) είναι άμεσα συνδεδεμένες μέσω της x. Οι C και D, εξακολουθούν να είναι συνδεδεμένες μέσω της y και η E παραμένει μη συνδεδεμένη. Υπάρχουν 4 απευθείας συνδέσεις, με αποτέλεσμα η TCC = 4 / 10. Οι έμμεσες συνδέσεις είναι A-D και B-D. Έτσι, η LCC = (4 +2) / 10 = 6 / 10.

Διαφορές LCOM4 και TCC/LCC

Υψηλή LCOM4 σημαίνει μη συνεκτική τάξη. Η LCOM4 = 1 είναι η καλύτερη.

Υψηλότερες τιμές δείχνουν μη συνεκτικό κλάσεις.

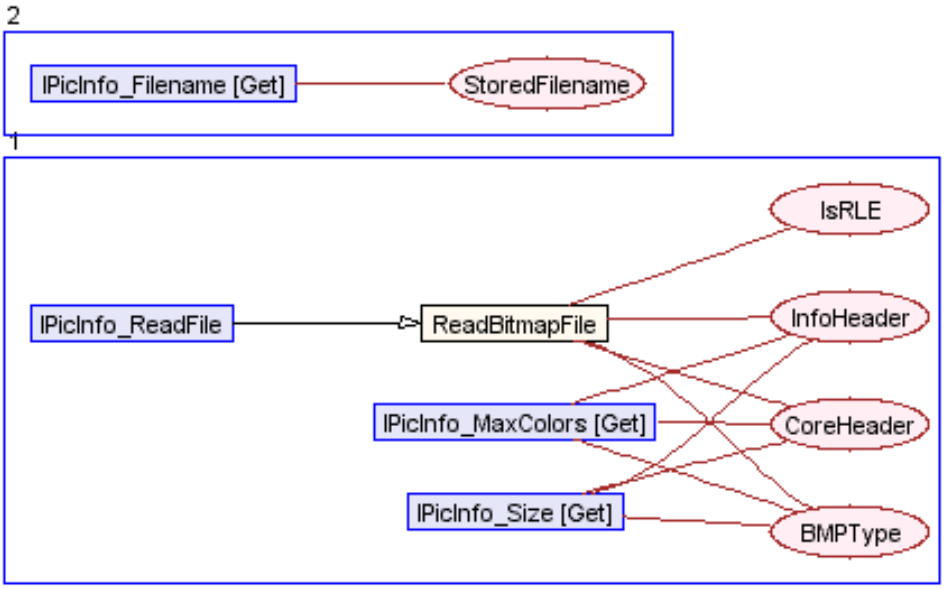
Υψηλή TCC και LCC σημαίνουν συνεκτικές τάξεις. TCC = LCC = 1 είναι η καλύτερη περίπτωση. Χαμηλότερες τιμές μας δείχνουν μικρότερη συνοχή.

Οι βοηθητικές μέθοδοι (μέθοδοι που δεν έχουν πρόσβαση σε μεταβλητές), αντιμετωπίζονται με διαφορετικό τρόπο. Η LCOM4 τις δέχεται ως συνεκτικές μέθοδοι. Οι TCC και LCC τις θεωρούν ως μη συνεκτικές. Μία τέτοια μέθοδος είναι η E στα παραπάνω παραδείγματα.

Διάγραμμα

Το διάγραμμα συνοχής δείχνει τις διαδικασίες τμημάτων του κώδικα καθώς και τις μεταβλητές που χρησιμοποιούνται στα ίδια τμήματα κώδικα. Το διάγραμμα επιτρέπει την κατανόηση των σχέσεων διαδικασία-διαδικασία και διαδικασία-μεταβλητών εντός ενός τμήματος του κώδικα.

Το διάγραμμα συνοχής έχει την ίδια λογική με την μετρική LCOM4. Το εκάστοτε τμήμα κώδικα είναι χωρισμένο σε σχετιζόμενες ομάδες που βασίζονται σε κλήσεις και τις κοινόχρηστες μεταβλητές. Όσες περισσότερες ομάδες υπάρχουν τόσο μικρότερη συνοχή έχουμε. Η τιμές των LCOM4 [1], [2], κλπ. εμφανίζονται δίπλα σε κάθε τμήμα κώδικα. Ακόμα κι αν η LCOM4 είναι μια μετρική κλάσεων και λιγότερο χρήσιμη για τμήματα κώδικα, φόρμες ή δομές, τα διαγράμματα συνοχής είναι διαθέσιμα για όλους τους τύπους των τμημάτων κώδικα.



Σχήμα_XXIII

PSI (Percentage of Shared Ideas)
Ποσοστό κοινών ιδεών

Μια μετρική για τη μέτρηση της σημασιολογικής συνοχής μιας κλάσης σε αντικειμενοστραφή προγράμματα. Η PSI χρησιμοποιεί πληροφορίες σε μια βάση γνώσεων για να αποδώσει την ποσότητα της συνοχής μιας εργασίας κάποιας κλάσης, παρέχοντας μας μια σαφέστερη εικόνα για τη συνοχή από ότι μας επιτρέπει η σύνταξη του κώδικα. Επιπλέον, αυτή η μετρική είναι ανεξάρτητη από τη δομή του κώδικα και μπορεί να υπολογιστεί πριν την υλοποίηση, παρέχοντας ενδείξεις για τις αδυναμίες του σχεδιασμού πιο έγκαιρα στον κύκλο της ανάπτυξης λογισμικού, όπου οι αλλαγές δεν είναι ακόμα τόσο χρονοβόρες και επίπονες.

Το PSI είναι ο αριθμός των εννοιών ή λέξεων-κλειδιών που μοιράζονται από τουλάχιστον δυο λειτουργιών μελών μιας κλάσης, διαιρούμενων με τον αριθμό των εννοιών ή λέξεων-κλειδιών που ανήκουν σε οποιαδήποτε λειτουργία μέλος της κλάσης. Το PSI για την κλάση Ca καθορίζεται ως εξής.

$$\text{PSI}= \frac{\left|\left\{x/x\in I_a \wedge \exists i,j\left(x\in I_{ai} \wedge x\in I_{aj}\right)\right\}\right|}{\left|\left\{y/y\in I_a \wedge \exists k\left(y\in I_{ak}\right)\right\}\right|}$$

Για 1<=i, j, k <= |Fa|, ή 0 αν δεν υπάρχουν ιδέες που συνδέονται με οποιαδήποτε λειτουργία της κλάσης.

Για παράδειγμα, έστω ότι η κλάση Ca να περιλαμβάνει τέσσερις λειτουργίες ως μέλη της και συνολικά δέκα ιδέες (έννοιες ή λέξεις-κλειδιά από τη βάση της) που συνδέονται με λειτουργίες για να δημιουργηθούν τα ακόλουθα σύνολα:

Ia1 = (i3, i4, i5), Ia2 = (i4, i5, i6), Ia3 = (i1, i2), και Ia4 = (i6, i7, i8, i9, i10).

Σε αυτό το παράδειγμα, οι ιδέες i4, i5 και i6 είναι κοινές σε τουλάχιστον δύο λειτουργίες, οι υπόλοιπες όχι. Ως εκ τούτου, το PSI για την κλάση Ca = 3 / 10 = 0,30.

Εμπειρικές μελέτες

Για να εκτελεστεί εμπειρική ανάλυση του PSI και των διάφορων εκδόσεων της LCOM, υπολογίστηκαν οι μετρικές χρησιμοποιώντας τον πηγαίο κώδικα ενός συνόλου κλάσεων από δύο GUI συστήματα γραμμένα σε C++,Gina και wxWindows.

Συγκρίθηκαν οι τιμές των μετρικών με τις εκτιμήσεις ειδικών λογισμικού.

Οι ειδικοί διατίμησαν την συνοχή της κάθε κλάσης στην ακόλουθη συμφωνημένη από κοινού κλίμακα: 0 = κακή, 0,25 = ανεπαρκής, 0,50 = μέτρια, 0,75 = καλή, 1,00 = εξαιρετική. Για να ελεγχθεί ο βαθμός στον οποίο οι ειδικοί έδωσαν σταθερές αξιολογήσεις της συνοχής, υπολογίστηκε η αξιοπιστία της κάθε ομάδας ειδικών από εσωτερικούς εκτιμητές. Χρησιμοποιήθηκε Gen++ για να υπολογιστεί η LCOM, HYSS για τον υπολογισμό LCOM1, LCOM2, LCOM3, LCOM4, LCOM5 και semMet για τον υπολογισμό του PSI .

Διενεργήθηκε στατιστική ανάλυση για να φανεί κατά πόσο οι τιμές των μετρικών μοιάζουν με τις εκτιμήσεις των ειδικών για την κάθε κλάση. Βασιζόμενοι στην ακόλουθη υπόθεση για τα ακόλουθα δύο πειράματα.

Υ1: ρ ίσο με 0 Δεν υπάρχει συσχετισμός μεταξύ της τιμής της μετρικής και της εκτίμησης τη ομάδας.

Υ2: ρ διάφορο του 0 Υπάρχει συσχετισμός μεταξύ της τιμής της μετρικής και της εκτίμησης τη ομάδας.

Όταν δύο μεταβλητές είναι ανεξάρτητες, ο συντελεστής της συσχέτισης είναι 0.

Μια άμεση σχέση μεταξύ δύο μεταβλητών υποδηλώνεται από μια θετική τιμή, η αντίστροφη σχέση υποδεικνύεται από αρνητικό πρόσημο.

Για να γίνει κατανοητό το νόημα αυτών των τιμών, οι Cohen και Hopkins, πρότειναν την ακόλουθη κλίμακα.

<0 .01 –μηδαμινό,

<0 .10 -0 .30 – ασήμαντο,

<0.30 - 0.50 – μέτριο,

<0.50 -0 .70 – μεγάλο,

<0.70 -0 .90 – πολύ μεγάλο,

<0 .90 - 1.0 – σχεδόν τέλειο

Πείραμα 1

Για αυτό το πείραμα, υπολογίστηκαν μετρικές σε ένα σύνολο 13 κλάσεων από το σύστημα wxWindows, ένα σύστημα γραμμένο σε C + + για την ανάπτυξη GUI εφαρμογής η οποία αναπτυσσόταν ή χρησιμοποιούνταν για περισσότερα από δώδεκα χρόνια.

Οι εκτιμήσεις των ειδικών προσμετρήθηκαν με μέσους όρους για να ληφθεί η εκτίμηση της κάθε ομάδας για την κάθε κλάση. Η αξιοπιστία του εσωτερικού εκτιμητή ήταν 0,89 υποδεικνύοντας υψηλό βαθμό συμφωνίας μεταξύ των ειδικών. Από τον πίνακα 1 φαίνεται ότι το PSI έχει στατιστικά πολύ μεγάλο συσχετισμό με την εκτίμηση των ειδικών. Εκτός της LCOM, οι άλλες μετρικές είχαν μέτρια ή μεγάλη συσχέτιση με τις βαθμολογήσεις των ειδικών. Οι LCOM3, LCOM4, και LCOM5 δεν είχαν σημαντική συσχέτιση με τις εκτιμήσεις των ειδικών.

Από τις ακριβείς τιμές που παρατίθενται στον Πίνακα_VII, μπορούμε να δούμε ότι το PSI είναι πιο κοντά στις εκτιμήσεις των ειδικών περισσότερο από κάθε LCOM μετρική.

Πίνακας_VI Συσχετισμοί μετρικών – ομάδων ειδικών για την αξιολόγηση της συνοχής

Metric	Correlation	p-value	Statistically Significant ($\alpha = 0.05$)
PSI	-0.7745	0.0019	✓
LCOM	-0.7257	0.0115	✓
LCOM1	-0.5782	0.0385	✓
LCOM2	-0.5557	0.0486	✓
LCOM3	-0.4873	0.0912	
LCOM4	-0.5371	0.0584	
LCOM5	-0.4462	0.1962	

Πείραμα 2

Σε αυτό το πείραμα υπολογίστηκαν οι μετρικές από ένα σύνολο 277 κλάσεων από Gina και wxWindows συστήματα όπου και ελέγχθηκαν για συσχετισμούς.

Πίνακας_VII **Συσχετισμοί ανά ζεύγος τιμών μεταξύ μετρικών (Πείραμα 2)**

	LCOM	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5
LCOM1	NS					
LCOM2	NS	0.9964				
LCOM3	NS	0.8997	0.9253			
LCOM4	NS	0.8561	0.8849	0.9838		
LCOM5	NS	0.4706	0.4868	0.5217	0.5436	
PSI	0.6333	0.6507	0.5895	NS	NS	NS

Εδώ χρησιμοποιήθηκαν οι ακόλουθες υποθέσεις:

Υ1: ρ ίσο με 0 Δεν υπάρχει συσχετισμός μεταξύ των τιμών των μετρικών

Υ2: ρ διάφορο του 0 Υπάρχει συσχετισμός μεταξύ των τιμών των μετρικών

Βρέθηκε μεγάλος συσχετισμός μεταξύ του PSI και των LCOM, LCOM1, και LCOM2, αλλά καμία σημαντική συσχέτιση μεταξύ του PSI και άλλων. Υπήρξε πολύ μεγάλος συσχετισμός σε κάθε ζεύγος των LCOM1, LCOM2, LCOM3, και LCOM4. Η LCOM5 είχε πολύ μεγάλη έως σχεδόν τέλεια συσχέτιση με κάθε άλλη μετρική εκτός του PSI. Τα συνολικά αποτελέσματα παρουσιάζονται στον πίνακα 2 όπου τα ζεύγη χωρίς στατιστική σημαντική συσχέτιση αναφέρονται με την ένδειξη “NS”.

Μετρικές σύζευξης: Coupling

RFC (Response For a Class)
Απόκριση κλάσης

Απόκριση κλάσης (Response For a Class). Η απόκριση μιας κλάσης είναι « το σύνολο των μεθόδων που μπορούν να εκτελεστούν ως απόκριση της κλάσης στην λήψη ενός μηνύματος από ένα αντικείμενο αυτής της κλάσης» (Chidamber και Kemerer, 1994). Η RFC είναι το πλήθος αυτών των μεθόδων ως απάντηση. Όσο αυξάνεται η RFC τόσο αυξάνεται και η καταβολή προσπαθειών για δοκιμές εξαιτίας της αύξησης της ακολουθίας των δοκιμών. Επίσης καθώς αυξάνεται η RFC είναι αναμενόμενο ότι αυξάνεται και η συνολική σχεδιαστική πολυπλοκότητα της κλάσης.

Η RFC μετράει τόσο την εξωτερική όσο και την εσωτερική επικοινωνία, αλλά κυρίως περιλαμβάνει μεθόδους που καλούνται εξωτερικά της κλάσης, επομένως αποτελεί και μέτρηση της δυνητικής επικοινωνίας μεταξύ της εν λόγω κλάσης και άλλων [ChK94, AlC98]. Η RFC είναι πιο ευαίσθητη μετρική σύζευξης από την CBO (που θα μελετηθεί αμέσως μετά) καθώς υπολογίζει μεθόδους αντί για κλάσεις [YSM02].

Εάν εμπλακεί, ως απόκριση σε μηνύματα που λαμβάνει (message passing) , μεγάλο πλήθος μεθόδων τότε η δοκιμές και η αποσφαλμάτωση της κλάσης καθίσταται πιο πολύπλοκη εφόσον απαιτεί μεγαλύτερη κατανόηση από την πλευρά του δοκιμαστή. Η τιμή της χειρότερης περίπτωσης για πιθανές αποκρίσεις θα βοηθήσουν στην κατάλληλη κατανομή της διάρκειας των δοκιμών [ChK94].

RFC = M + R (Πρώτο βήμα μέτρησης)

RFC’ = M + R’ (Πλήρης μέτρηση)

M =ο αριθμός των μεθόδων της κλάσης

R = ο αριθμός των απομακρυσμένων μεθόδων που καλούνται άμεσα από την κλάση

R’ = ο αριθμός των απομακρυσμένων μεθόδων που καλούνται αναδρομικά μέσω της συνολικής ιεραρχίας της κλάσης.

Μία δοθείσα μέθοδος υπολογίζεται μόνο μία φορά ως R (και R’) ακόμα και εάν εκτελείται από διάφορες μεθόδους M.

Όπως φαίνεται και από τον παράδειγμα, που έχει παρουσιαστεί στην μετρική κληρονομικότητας DIT, η RFC για την κλάση Location ισούται με 3, δεδομένου ότι έχει 3 τοπικές μεθόδους (Initialize, Get_X, και Get) και δεν καλούν εξωτερικές κλάσεις. Η Point έχει 6 τοπικές μεθόδους και κάνει κλήσεις στις Location, Jnitialize, Graphic. PutPixel, Graphics. CurrentColor, PointJ.Coordinate_Of και point.Y_Coordinate_Of με RFC= 11. Η Circle έχει 5 τοπικές μεθόδους και κάνει κλήσεις σε Graphics.Draw_Circle, Graphics.Current Color, Graphic. Background_Color, και Graphics.Set_Color με RFC=9.

Μια μελέτη 30 C++ έργων έδειξε ότι η αύξηση της RFC αυξάνει και την πυκνότητα των σφαλμάτων καθώς παράλληλα μειώνει την ποιότητα. Η μελέτη προτείνει την «βέλτιστη» χρήση της RFC, χωρίς όμως να λέει ποια είναι η βέλτιστη. Φαίνεται ασφαλές να υποθέσουμε ότι μία υψηλή RFC είναι επιζήμια σε Visual Basic προγράμματα ωστόσο (Misra & Bhavsar,2003).

Παράδειγμα 1 RFC

```
public class A {
private B aB;
public void methodA1() {
return aB.methodB1();
}
public void methodA2(C aC) {
return aC.methodC1();
     }
}
RS = { methodA1, methodA2, methodB1, methodC1}
```

Παράδειγμα 2 RFC

```
public class A {
private B aB;
public void methodA1() {
return aB.methodB1();
}
public void methodA2() {
return aB.methodB1();
   }
}
RS = { methodA1, methodA2, methodB1}
RS = { methodA1, methodA2, methodB1, methodB1}
```

CBO (Coupling between Object Classes)
Σύζευξη μεταξύ αντικειμενοστραφών κλάσεων

Η CBO είναι ο αριθμός των κλάσεων, όπου οι κλάσεις είναι συνδεδεμένες.

Πολλαπλές προσβάσεις στην ίδια κλάση υπολογίζονται ως μία πρόσβαση. Μόνο καλούμενες μέθοδοι και μεταβλητές με αναφορά υπολογίζονται. Άλλοι τύποι αναφορών, όπως σταθερές, κλήσεις σε API's, χειρισμοί γεγονότων, χρήση τύπων καθορισμένων από τον χρήστη και στιγμιότυπα, αγνοούνται. Εάν η μέθοδος είναι πολυμορφική (είτε λόγω υπέρβασης είτε λόγω υπερφόρτωσης), όλες οι κλάσεις που μπορούν να λάβουν κλήσεις συμπεριλαμβάνονται στην μέτρηση της σύζευξης.

Η υψηλή CBO είναι ανεπιθύμητη. Υπερβολική σύζευξη μεταξύ κλάσεων είναι επιζήμια για τον σχεδιασμό και αποτρέπει την επαναχρησιμοποίηση. Όσο πιο ανεξάρτητη είναι μια κλάση, τόσο πιο εύκολη είναι η επαναχρησιμοποίηση της σε κάποια άλλη εφαρμογή. Προκειμένου προωθηθεί η ενθυλάκωση, η σύζευξη μεταξύ των αντικειμένων πρέπει να κρατηθεί στο ελάχιστο. Όσο μεγαλύτερος είναι ο αριθμός των ζευγαριών, τόσο υψηλότερη είναι η ευαισθησία στις αλλαγές σε άλλα σημεία του σχεδιασμού, και ως εκ τούτου η συντήρηση καθίσταται πιο δύσκολη. Η υψηλή σύζευξη έχει αποδειχθεί ότι προκαλεί επιρρέπεια σε σφάλματα. Αυτό που απαιτείται είναι αυστηρός και παρατεταμένος έλεγχος. Οι Sahraoui, Godin και Miceli σε ένα άρθρο τους είχαν ορίσει ως πολύ υψηλή τιμή την CBO>14.

Λεπτομέρειες εφαρμογής.

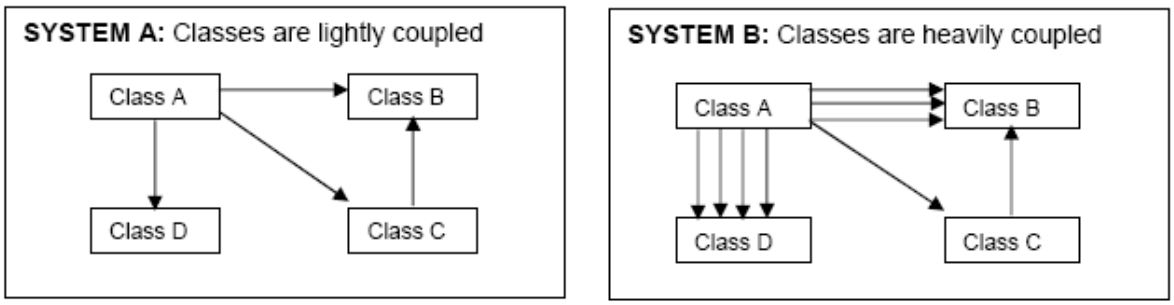
Ο αρχικός ορισμός της CBO αν και δεν είναι απόλυτα σαφής οφείλουμε να τον αναφέρουμε. Ο ορισμός της CBO ασχολείται με τα ίχνη των μεταβλητών (instance variables) και όλες τις μεθόδους της κλάσης. Στο VB.NET, αυτό σημαίνει μη κοινές μεταβλητές, κατανεμημένες ή μη μέθοδοι. Αυτές οι κοινές μεταβλητές δεν λαμβάνονται υπόψη της μετρικής. Αντιθέτως, όλες οι καλούμενες μέθοδοι υπολογίζονται είτε είναι κατανεμημένες είτε όχι.

Εάν η κλήση είναι πολυμορφική είναι στην διεπαφή της μεθόδου στο .NET και δεν προσμετράτε στην σύζευξη, ούτε της διεπαφής ούτε στις κλάσεις που υλοποιούν την διεπαφή. Εάν η κλήση είναι πολυμορφική είναι στο ορισμό της μεθόδου στη Visual Basic διεπαφή της κλάσης (βασική κλάση), υπολογίζεται στην διεπαφή της κλάσης αλλά όχι σε όποια κλάση υλοποιεί την διεπαφή. Αυτός είναι περιορισμός της υλοποίησης και όχι της CBO.

Στην υλοποίηση της CBO, όταν η παράγωγη κλάση καλεί τις βασικές της μεθόδους που κληρονομεί, συνδέεται με την κλάση γονέα στην οποία έχουν οριστεί οι μέθοδοι. Ο αρχικός ορισμός δεν ορίζει αν η κληρονομικότητα πρέπει να χειριστεί με κάποιο συγκεκριμένο τρόπο.

Όπως όρισα και στην αρχή CBO είναι το πλήθος των συνδεδεμένων κλάσεων. Αυτός ο ορισμός είναι απλός και δεν αποσαφηνίζει την πολυπλοκότητα της σύνδεσης μεταξύ των κλάσεων. Το παράδειγμα του σχήματος Σχήμα_XXIV μας διευκολύνει στην κατανόηση.

Σχήμα_XXIV



Κάθε γραμμή στο διάγραμμα αναφέρεται σε μήνυμα από μια κλάση σε μια άλλη.

Τα δύο διαγράμματα παρουσιάζουν τον σχεδιασμό δύο διαφορετικών συστημάτων. Χρησιμοποιώντας την μετρική CBO μετράμε την κλάση A η οποία είναι 3. Ωστόσο, είναι ευδιάκριτο ότι η κλάση A του συστήματος B έχει μεγαλύτερη πολυπλοκότητα συγκριτικά με την κλάση A του συστήματος A.

Ο ορισμός της CBO εδώ θα πρέπει να επαναπροσδιοριστεί με πιο συγκεκριμένο τρόπο. Η CBO θα πρέπει να συνυπολογίζει τον αριθμό των μηνυμάτων που πέρασαν μεταξύ των κλάσεων. Στο παραπάνω παράδειγμα αυτός ο επαναπροσδιορισμός θα μετέτρεπε την τιμή της CBO από 3 σε 8 για το σύστημα B.

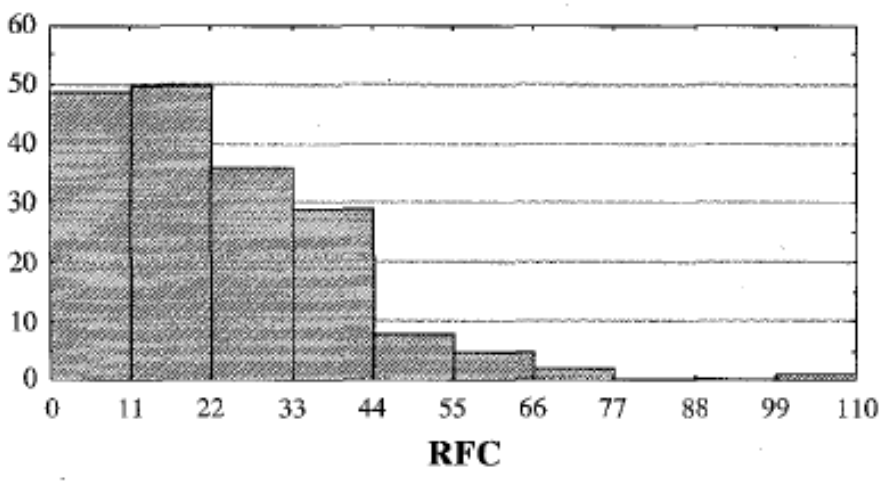
Παράδειγμα CBO

```
import java.util.Calendar;
public class AdultIssuePolicy implements IssuePolicy {
public Calendar computeDueDate(BiblioType type, Calendar from) {
Calendar result = (Calendar)from.clone();
result.add(Calendar.DATE, 14);
return result;
}
}
```

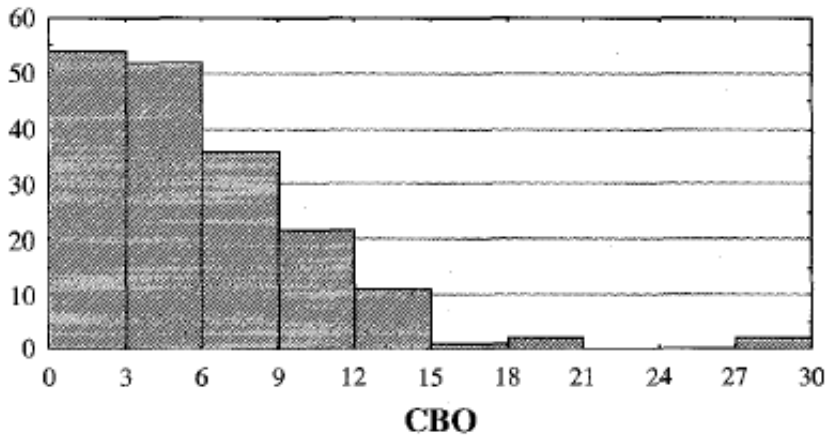
Στατιστικά οι μετρικές RFC και CBO είναι οι πιο ανεξάρτητες μετρικές (όσον αφορά τα αντικειμενοστραφή λογισμικά) συγκριτικά με όλες τις άλλες. Σε μελέτες όπου εφαρμόστηκαν μετρικές μέθοδοι για την αξιολόγηση κλάσεων, μεθόδων και γενικότερα λογισμικού, μετρικές των ίδιων κατηγοριών έδειξαν κάποιο συσχετισμό στα αποτελέσματα τους ενώ οι RFC και CBO ήταν τελείως ανεξάρτητες ως μετρικές που δεν λαμβάνουν κατά τον υπολογισμό των τιμών τους περιττές πληροφορίες.

Μία τέτοια μελέτη διενεργήθηκε από τους Chidamber και Kemerer σε 180 κλάσεις 8 συστημάτων που είχαν συνταχθεί σε C++. Ακολουθούν οι πίνακες Πίνακας_VIII και Πίνακας_XI από την έρευνα τους.

Πίνακας_VIII



Πίνακας_XI



Κατανομή των μετρικών.

Ο x άξονας αντιπροσωπεύει τις τιμές των μετρικών και ο y τον αριθμό των κλάσεων.

ΕΠΙΛΟΓΟΣ ΜΕΤΡΙΚΩΝ

Το αντικειμενοστραφές λογισμικό είναι θεμελιωδώς διαφορετικό από το λογισμικό που αναπτύχθηκε με τη χρήση συμβατικών μεθόδων. Ως εκ τούτου, οι μετρικές για ΟΟ συστήματα επικεντρώνονται στη μέτρηση που μπορούν να εφαρμοστούν στην κλάση και στα σχεδιαστικά χαρακτηριστικά, στην τοπικότητα (localization), στην ενθυλάκωση, στην απόκρυψη πληροφοριών, στην κληρονομικότητα, και στις τεχνικές αφαίρεσης αντικειμένων.

Οι μετρικές μέθοδοι αποτελούν ένα ποσοτικό τρόπο για να εκτιμηθεί η ποιότητα των εσωτερικών ιδιοτήτων του προϊόντος, επιτρέποντας έτσι τον μηχανικό λογισμικού να εκτιμήσει την ποιότητα του προϊόντος του πριν την υλοποίηση του. Οι μετρικές παρέχουν την γνώση ώστε να δημιουργηθεί αρχικά μία αποδοτική σχεδίαση του συστήματος, έπειτα ένας στέρεος κώδικας και τέλος εξονυχιστικοί έλεγχοι και αξιολογήσεις του εκάστοτε προγράμματος. Για να είναι χρήσιμες σε πραγματικές εφαρμογές, οι μετρικές λογισμικού πρέπει να είναι απλές και υπολογίσιμες, πειστικές, συνεπής και αντικειμενικές. Θα πρέπει να είναι ανεξάρτητες από γλώσσες προγραμματισμού και να παρέχουν αποτελεσματική ανατροφοδότηση στον μηχανικό λογισμικού.

Οι μετρικές έχουν νόημα μόνο εάν έχουν εξεταστεί για την στατιστική τους εγκυρότητα. Το διάγραμμα ελέγχου είναι μια απλή μέθοδος για να επιτευχθεί η διαδικασία αυτή, την ίδια στιγμή που εξετάζει τις μεταβολές των αποτελεσμάτων από τις μετρικές μεθόδους. Η συλλογή στοιχείων, οι υπολογισμοί των μετρικών, και η ανάλυση τους είναι τα τρία βήματα που πρέπει να γίνουν για μια μετρική μέθοδος.. Με τη δημιουργία ενός συνόλου μετρικών, ουσιαστικά δημιουργούμε μία βάση δεδομένων που περιλαμβάνει διαδικασίες μετρήσεις καθώς και μετρήσιμα προϊόντα. Οι μηχανικοί λογισμικού και οι επικεφαλής τους μπορούν να αποκτήσουν καλύτερη εικόνα για το έργο που επιτελούν και το προϊόν που παράγουν.

Η ομάδα μετρικών CK καθορίζει έξι μετρικές με βάση τις κλάσεις. Η συγκεκριμένη ομάδα μετρικών περιλαμβάνει μετρικές για την αξιολόγηση των συνεργασιών μεταξύ των κλάσεων και της συνοχής των μεθόδων που υπάρχουν σε μια κλάση. Σε επίπεδο κλάσεων η CK μετρικές μπορούν να ενισχυθούν με τις μετρικές MOOD των Lorenz και Kidd. Αυτές περιλαμβάνουν μετρικές κλάσεων για το μέγεθος τους και μετρικές που παρέχουν πληροφορίες σχετικά με τον βαθμό εξειδίκευσης των υποκλάσεων.

Μετρικές που στρέφονται στην λειτουργικότητα, εστιάζονται στο μέγεθος και την πολυπλοκότητα των μεμονωμένων ενεργειών. Είναι σημαντικό να σημειωθεί, ωστόσο, ότι το πρωταρχικό εφαλτήριο για ΟΟ σχεδιαστικές μετρικές είναι σε επίπεδο κλάσεων.

Μια ευρεία ποικιλία των ΟΟ μετρικών έχουν προταθεί για την αξιολόγηση δοκιμών των ΟΟ συστημάτων. Οι μετρικές αυτές επικεντρώνονται στην ενθυλάκωση, την κληρονομικότητα, την πολυπλοκότητα των κλάσεων, και τον πολυμορφισμό. Πολλές από αυτές τις μετρικές έχουν προσαρμοστεί από τις CK μετρικές και τις μετρικές που προτάθηκαν από τους Lorenz και Kidd. Άλλες έχουν προταθεί και από τον Binder. Μετρήσιμα χαρακτηριστικά της ανάλυσης και του σχεδιασμού μπορούν να βοηθήσουν τον επικεφαλής του έργου για ένα ΟΟ σύστημα στον

σχεδιασμό και την παρακολούθηση των δραστηριοτήτων του συστήματος. Ο αριθμός των περιπτώσεων χρήσης (scenario scripts –use cases), των κλάσεων, και των υποσυστημάτων, παρέχουν πληροφορίες σχετικά με το επίπεδο της προσπάθειας που απαιτείται για την υλοποίηση του συστήματος.

Οι μετρικές για την ανάλυση εστιάζουν στην λειτουργία, τα δεδομένα και την συμπεριφορά, οι τρεις συνιστώσες του μοντέλου ανάλυσης. Οι μετρικές για τον σχεδιασμό υπολογίζουν την αρχιτεκτονική του συστήματος, τα σχεδιαστικά συστατικά και τα σχεδιαστικά θέματα της διεπαφής. Οι σχεδιαστικές μετρικές σε επίπεδο συστατικών παρέχουν ένα δείκτη της ποιότητας των τμημάτων του κώδικα, με την βοήθεια μετρικών συνοχής, σύζευξης και πολυπλοκότητας. Οι μετρικές σχεδιασμού διεπαφών παρέχουν ενδείξεις καταλληλότητας σχεδιασμών για GUI.

Η επιστήμη των υπολογιστών παρέχει ένα ενδιαφέρων σύνολο μετρικών σε επίπεδο πηγαίου κώδικα. Χρησιμοποιώντας πλήθος τελεστών που υπάρχουν στον κώδικα, έχουμε την δυνατότητα με την χρήση των μετρικών να αξιολογήσουμε την ποιότητα των προγραμμάτων μας. Ωστόσο οι μετρικές που έχουν προταθεί για δοκιμές καθώς και για την συντήρηση των προγραμμάτων δεν είναι πολλές.

Η διαδικασία των μετρήσεων επιτρέπει μια οργάνωση ή μια εταιρεία να υιοθετήσει μία στρατηγική πολιτική, παρέχοντας της πλήρη εικόνα για την αποτελεσματικότητα της ανάπτυξης λογισμικού. Οι μετρικές έργων κυρίως στις Η.Π.Α. είναι πολύ συχνές. Δίνουν τη δυνατότητα σε ένα διευθυντή να προσαρμόζει την ροή των έργων της επιχείρησης του και τις τεχνικές προσέγγισης σε πραγματικό χρόνο.

Τόσο οι μετρικές μεγέθους όσο και οι μετρικές σε σχέση με την λειτουργικότητα χρησιμοποιούνται σε ολόκληρη την βιομηχανία. Οι μετρικές μεγέθους χρησιμοποιώντας τις γραμμές του κώδικα παρέχουν έναν τρόπο μέτρησης της αποδοτικότητας των εργαζομένων όπως τις ώρες εργασίας τους ανά μήνα ή την παραγωγικότητα του καθενός. Οι μετρικές ποιότητας λογισμικού, όπως οι μετρικές παραγωγικότητας, εστιάζουν στην εξέλιξη έργων και προϊόντων. Κατά την φάση της ανάλυσης και της ανάπτυξης, οι μετρικές καταγράφουν την ποιότητα με αποτέλεσμα οι επιχειρήσεις να έχουν την δυνατότητα να αναθεωρήσουν και να διορθώσουν τμήματα και πολιτικές τους που παρουσιάζουν προβλήματα ή καθυστερήσεις.

Δυστυχώς όμως η πραγματικότητα δεν είναι τόσο πιστή στα μοντέλα που πρέπει να ακολουθούνται. Η σχεδίαση των εφαρμογών γίνεται σχεδόν ενστικτωδώς και η αξιολόγηση είναι εκτός της κουλτούρας των μηχανικών λογισμικού. Οι προγραμματιστές, στην Ευρώπη μα κυρίως στην Ελλάδα, έχουν δώσει όλο το βάρος τους στην υλοποίηση των εφαρμογών, αδιαφορώντας για την εξέταση αυτών. Η αναγκαιότητα όμως των μετρικών μεθόδων είναι παρούσα και παραφράζοντας την φράση του Σωκράτη κατά την απολογία του "ζωή που δεν την εξετάζεις δεν αξίζει να την ζεις", πρόγραμμα που δεν το εξετάζεις δεν αξίζει να το γράψεις.

Συμπεράσματα - Προτάσεις

Δυστυχώς τα ευρέως χρησιμοποιούμενα προγράμματα NetBeans, Eclipse, Together Borland, δεν παρέχουν την πληθώρα των μετρικών που έχουν προταθεί, δυσκολεύοντας την επιλογή και εφαρμογή αυτών, για την πλειοψηφία των προγραμματιστών. Η παρουσία δε των μετρικών μεθόδων είναι και αυτή πολύ περιορισμένη. Σε μια θεωρητική όμως βάση ή σε προνομιακούς προγραμματιστές των οποίων οι δυνατότητες υπερβαίνουν τον μέσο όρο των προγραμματιστών και κυρίως των φοιτητών, η πρόταση της παρούσας πτυχιακής εργασίας για τις μετρικές μεθόδους είναι η ακόλουθη.

Οι Chidamber και Kemerer έχουν προτείνει τις εξής μετρικές:

- Weighted methods per class (WMC)
- Depth of the inheritance tree (DIT)
- Number of children (NOC)
- Coupling between object classes (CBO)
- Response for a class (RFC)
- Lack of cohesion in methods (LCOM)

Ο Binder το 1994, όρισε μετρικές για την πολυπλοκότητα των κλάσεων και τον πολυμορφισμό, οι οποίες ανήκουν στην κατηγορία της πολυπλοκότητας και περιλαμβάνουν τρεις CK μετρικές: την

- WMC
- CBO
- RFC.

Οι Lorenz και Kidd έχουν προτείνει τις ακόλουθες μετρικές:

- Average operation size (OSavg)
- Operation complexity (OC)
- Average number of parameters per operation (NPavg)
- Class size (CS)
- Number of operations overridden by a subclass (NOO)
- Number of operations added by a subclass (NOA)
- Specialization index (SI)

Ορισμένες εξ'αυτών των μετρικών μεθόδων υπερέχουν άλλων όπως οι TCC και LCC που υπολογίζουν μόνο τις ορατές μεθόδους σε αντίθεση με τις LCOMx που εξετάζουν όλες τις μεθόδους. Μια μέθοδος είναι ορατή αν δεν είναι ιδιωτική (private), αν εφαρμόζει μια διεπαφή ή χειρίζεται ένα γεγονός.

Επιπλέον διεξήχθη το συμπέρασμα με την εκτέλεση θεωρητικής και εμπειρικής ανάλυσης για το PSI και έξι παραλλαγές της LCOM, ότι το PSI είναι εμπειρικά και θεωρητικά πιο έγκυρο, και ότι συμφωνεί περισσότερο με τις εκτιμήσεις των ειδικών για την συνοχή, συγκριτικά με οποιαδήποτε έκδοση της LCOM.

Επίσης, δεδομένου ότι το PSI δεν στηρίζεται στη δομή του κώδικα, μπορεί να υπολογιστεί στο στάδιο του σχεδιασμού, παρέχοντας αποτελέσματα πριν την εφαρμογή. Έτσι οι μηχανικοί λογισμικού θα πρέπει να εξετάζουν το PSI για να αξιολογούν την συνοχή αντικειμενοστραφούς λογισμικού και να το προτιμούν από τις μετρικές LCOMx, TCC και LCC.

Αναφορικά με τις RFC και RFC', η RFC είναι ο αρχικός ορισμός της μέτρησης. Μετράει μόνο το πρώτο επίπεδο των κλήσεων εκτός της κλάσης. Η RFC' μετράει το σύνολο των αποκρίσεων, συμπεριλαμβανομένων και των μεθόδων που καλούνται εξωτερικά, αναδρομικά, μέχρις ότου να μην υπάρχουν νέες μέθοδοι. Η χρήση της RFC', θα πρέπει να προτιμάται έναντι της RFC. Η RFC, δεν είναι ικανή να υπολογίσει όλη την ιεραρχία της κλάσης με χειροκίνητο τρόπο. Με αυτοματοποιημένη ανάλυση του κώδικα από κάποιο εργαλείο, λαμβάνοντας τις τιμές τις RFC' δεν αντιμετωπίζουμε καμία δυσκολία, έχοντας μια πιο εμπεριστατωμένη μέτρηση του εκτελέσιμου κώδικα

Δυο βασικές μετρικές που έχουν προταθεί για την μέτρηση κλάσεων σχετικά με την κληρονομικότητα είναι η DIT (βάθος δενδροειδούς κληρονομικότητας) και η NOC (πλήθος των παιδιών). Χρησιμοποιείται επίσης κι άλλη μία μετρική που είναι η NOM (πλήθος των μεθόδων) για την μέτρηση των μεθόδων. Στην παρούσα εργασία ανέπτυξα την DIT η οποία είναι η αντιπροσωπευτικότερη των παραπάνω αφού υπολογίζει το πλήθος των κλάσεων γονέα μέχρι την ρίζα. Αυτή η μετρική μπορεί επιπλέον να υπολογιστεί για μία κλάση λαμβάνοντας την ένωση όλων των διαγραμμάτων κλάσεων σε ένα UML μοντέλο, που διαπερνούν την ιεραρχία της κληρονομικότητας της εν λόγω κλάσης.

Η εφαρμογή της LOC (πλήθος των γραμμών του κώδικα) εξαιτίας της λειτουργίας της, παρουσιάζει μία εικόνα για την πολυπλοκότητα του κώδικα, όπως έχει αναλυθεί και στο αντίστοιχο υποκεφάλαιο, αλλά η χρήση της CC (κυκλωματική πολυπλοκότητα) κρίνεται αναγκαία, παρά το γεγονός ότι δημιουργήθηκε από τον McCabe το 1976.

Εν τέλει εκ των συμπερασμάτων προκύπτει η εξής πρόταση σχετικά με τις καταλληλότερες μετρικές μεθόδους που πρέπει να εφαρμοστούν για την αξιολόγηση ενός προγράμματος:

- ✓ PSI (Ποσοστό κοινών ιδεών)
- ✓ CBO (Σύζευξη μεταξύ αντικειμενοστραφών κλάσεων)
- ✓ DIT (Βάθος δενδροειδούς κληρονομικότητας)
- ✓ CC (Κυκλωματική πολυπλοκότητα)
- ✓ RFC' (Απόκριση κλάσης)

Η εφαρμογή των παραπάνω μετρικών, ως μία ομάδα, θα εξασφάλιζε τους στόχους και θα αποκομίζαμε τα οφέλη των μετρικών μεθόδων.

Η εφαρμογή μετρικών είναι περισσότερο από ποτέ μία αναγκαιότητα, ταυτόχρονα και η χρήση των προγραμματιστικών διεπαφών των εφαρμογών. Εξαιτίας του ανταγωνισμού των εφαρμογών, των προγραμμάτων και των βιβλιοθηκών είναι αναμενόμενο ότι οι απαιτήσεις του κάθε συμμετέχοντα έχουν αυξηθεί. Η υπεροχή κρίνεται στην αξιοπιστία, στην σταθερότητα, στην συμβατότητα και στην ευκολία και την χαμηλού κόστους συντήρηση. Αυτά επιτυγχάνονται πέρα από την χρήση σωστών διεπαφών, μέσω των σχεδιαστικών κανόνων και επιβεβαιώνονται μέσω των μετρικών.

Οι προγραμματιστές δυστυχώς δεν έχουν ενσωματώσει ακόμα αυτές τις διαδικασίες στην ανάπτυξη λογισμικού. Πολλοί θεωρούν ότι η επιτυχία τους περιορίζεται στην επιτυχή εκτέλεση του προγράμματος και ότι αυτό είναι αρκετό και για την εμπορική επιτυχία του λογισμικού αλλά και για την δική τους ικανοποίηση ως προγραμματιστές. Θεωρούν ότι υπάρχουν κάποιοι που έχουν την δυνατότητα να γράφουν κώδικα και να προγραμματίζουν και κάποιοι άλλοι απλά για να κρίνουν, αυθαίρετα κατά την άποψη τους, τα δικά τους προγράμματα.

Η πραγματικότητα όμως είναι ότι υπάρχουν προγραμματιστές που αναπτύσσουν λογισμικά ενστικτωδώς, χωρίς κανόνες και προγραμματιστές που πριν ξεκινήσουν την συγγραφή του κώδικα, αφιερώνουν ένα μέρος του χρόνου τους για να σχεδιάσουν, κατά την υλοποίηση έχουν διαρκώς κατά νου τους σχεδιαστικούς κανόνες και στο τέλος, ενδεχομένως και ευχής έργο ανά μικρά διαστήματα, τρέχουν τις μετρικές για να ελέγχουν την ορθότητα της ανάπτυξης του λογισμικού τους.

Οι φάσεις αυτές της σχεδίασης και του ελέγχου μέσω των μετρικών δεν είναι απαραίτητα δουλειά ενός άλλου επαγγέλματος, κατώτερου του προγραμματιστή. Είναι ο τρόπος για να γίνει κάποιος επιτυχημένος προγραμματιστής και να υπερέχει μεταξύ των άλλων!

ΒΙΒΛΙΟΓΡΑΦΙΑ

- Χατζηγεωργίου Αλέξανδρος. (2005), Αντικειμενοστρεφής σχεδίαση UML, αρχές, πρότυπα και ευρετικοί κανόνες
- Καμέας Α. (2003), Ειδικά θέματα τεχνολογίας λογισμικού
- Σπινέλης Δ. (2008), Ποιότητα κώδικα: Η προοπτική του ανοικτού λογισμικού
- Μπαρούνης Κωνσταντίνος, A comparison of cohesion metrics for object-oriented systems
- Diomidis Spinellis, Panagiotis Louridas, A Framework for the Static Verification of API Calls
- Xenos M., Tsalidis C., Christodoulakis D., Measuring Software Complexity Using Software Metrics
- Stavrinoudis D, Xenos M., Christodoulakis D., (1999) Relation between software metrics and maintainability
- K. Zikouli, Stavrinoudis D, Xenos M., Christodoulakis D. (2000), Object-oriented metrics – a survey
- Xenos Michalis (2006), Software Metrics and Measurements
- A. Stefani, Xenos M. (2009), Meta-metric Evaluation of E-Commerce-related Metrics
- Tulach Jaroslav (2008), Practical API Design: Confessions of a Java Framework Architect
- Roger S. Pressman, Software engineering: a practitioner's approach, 5th ed.
- Stephen H. Kan (2002), Metrics and Models in Software Quality Engineering, Second Edition
- McCabe J. Thomas (1976), A Complexity Measure
- Berard, Prentice-Hall (1993), Essays on Object-Oriented Software Engineering
- Laurie Williams, Dright Ho, and Sarah Heckman, Software Metrics in Eclipse
- Steven Clarke, Measuring API Usability
- Jeffrey Stylos and Brad Myers (2007), Mapping the Space of API Design Decisions
- Joshua Bloch, How to Design a Good API and Why it Matters
- Michi Henning (2007), API Design Matters
- Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, Mohamed Khalil (2006), Lessons from applying the systematic literature review process within the software engineering domain
- William W. Pritchett IV (1996), Applying Object-Oriented Metrics to Ada 95
- Wei Li (1998), Another metric suite for object-oriented programming
- Edward Allen, Measuring coupling and cohesion: An information – Theory Approach
- James M. Bieman (1998), Measuring Design-Level Cohesion
- Sami Mäkelä, Ville Leppänen (1996), Observations on Lack of Cohesion Metrics
- Cara Stein, Letha Etzkorn, Sampson Gholston, Phillip Farrington, Julie Fortune, A Knowledge-Based Cohesion Metric for Object-Oriented Software
- Lionel C. Briand, Wurgen Wust , John W. Daly, D. Victor Porter (1999), Exploring the relationships between design measures and software quality in object-oriented systems
- L.H. Etzkorn, W.E. Hughes Jr., C.G. Davis (2000), Automated reusability quality analysis of OO legacy software
- Wei Li (1993), Object-Oriented Metrics that Predict Maintainability
- Paul. D. Scott, Gui Gui (2009), Measuring Software Component Reusability by Coupling and Cohesion Metrics

- Aine Mitchel, James F. Power, An Empirical Investigation into the Dimensions of Run-Time Coupling in Java Programs
- Jefferson Offutt, Mary Jean Harrold, Priyadarshan Kolte, A Software Metric System for Module Coupling
- Capers Jones (2008), A short history of lines of code (LOC) metrics
- Everald E. Mills (1988), Software Metrics
- Lanza M., Marinescu R. (2006), Evaluating the Design
- Rodrigo Vivanco, Nicolino Pizzi (2004), Finding Effective Software Metrics to Classify Maintainability Using a Parallel Genetic Algorithm
- Robert E. Park, Wolfhart B. Goethert, William A. Florac (1996), Goal-Driven Software Measurement - A Guidebook
- WilliamW. Cohen, Pradeep Ravikumar, Stephen E. Fienberg (2003), A Comparison of String Distance Metrics for Name-Matching Tasks
- N. Fenton, S.L. Pfleeger (1997), Software Metrics: A Rigorous & Practical Approach
- M.J. Shepperd, D. Ince (1993), Derivation and Validation of Software Metrics
- S. Henry, D. Kafura, Software structure metrics based on information flow
- S.D. Conte, H.E. Dunsmore, V.Y. Shen (1986), Software engineering metrics and models
- H.L. Hausen (1989), Yet another model of software quality and productivity
- K.L. Morris (1988),Metrics for Object-Oriented Software Development Environments
- T.J. McCabe, L. A. Dreyer (1994), Testing an object-oriented application
- M. Lorenz (1991), Real world reuse
- E. Weyuker (1988), Evaluating software complexity measures
- http://www.aivosto.com/project/help/pm-loc.html
- http://www.ndepend.com/Metrics.aspx
- http://eclipse-metrics.sourceforge.net/
- http://www.laynetworks.com/software%20engineering_3.htm
- http://c2.com/cgi/wiki?CouplingAndCohesion
- http://www.sciencedirect.com/
- http://www.theiet.org/
- http://citeseerx.ist.psu.edu/
- http://opac.keele.ac.uk/

Οδηγός Χρήσης του Eclipse Galileo

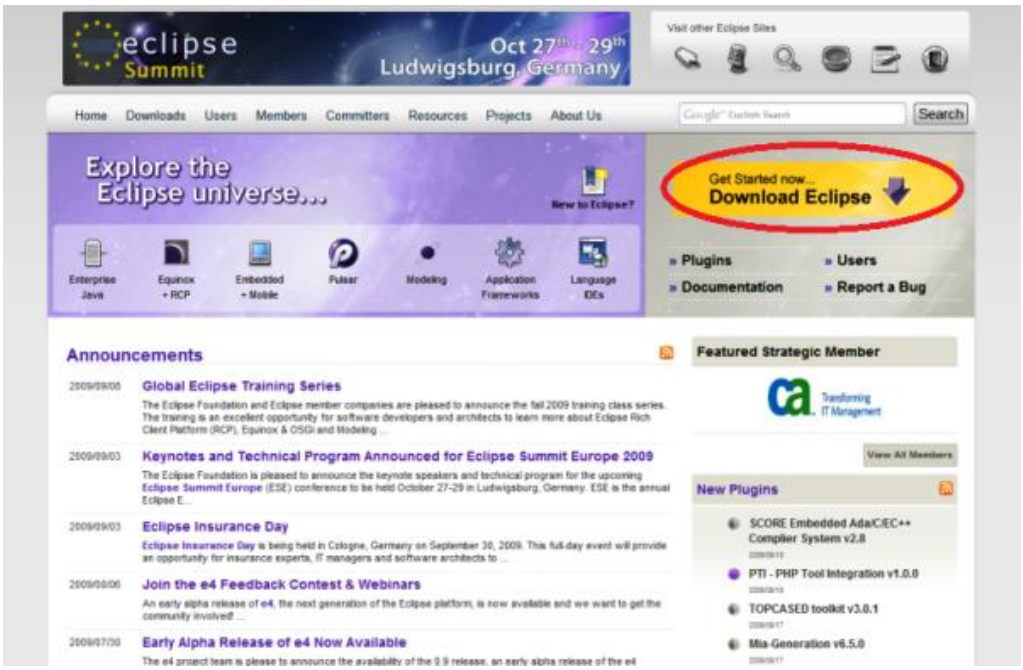
Στο κεφάλαιο αυτό εφαρμόζονται οι μετρικές μέθοδοι σε διάφορα προγράμματα java με την χρήση του Eclipse Galileo. Θα παρουσιαστεί ένας οδηγός χρήσης και έπειτα η εφαρμογή μια σειράς μετρικών μεθόδων. Για το συγκεκριμένο εγχειρίδιο θεωρείται δεδομένη η ύπαρξη ενός Java Developement Kit – JDK.

Eclipse Galileo

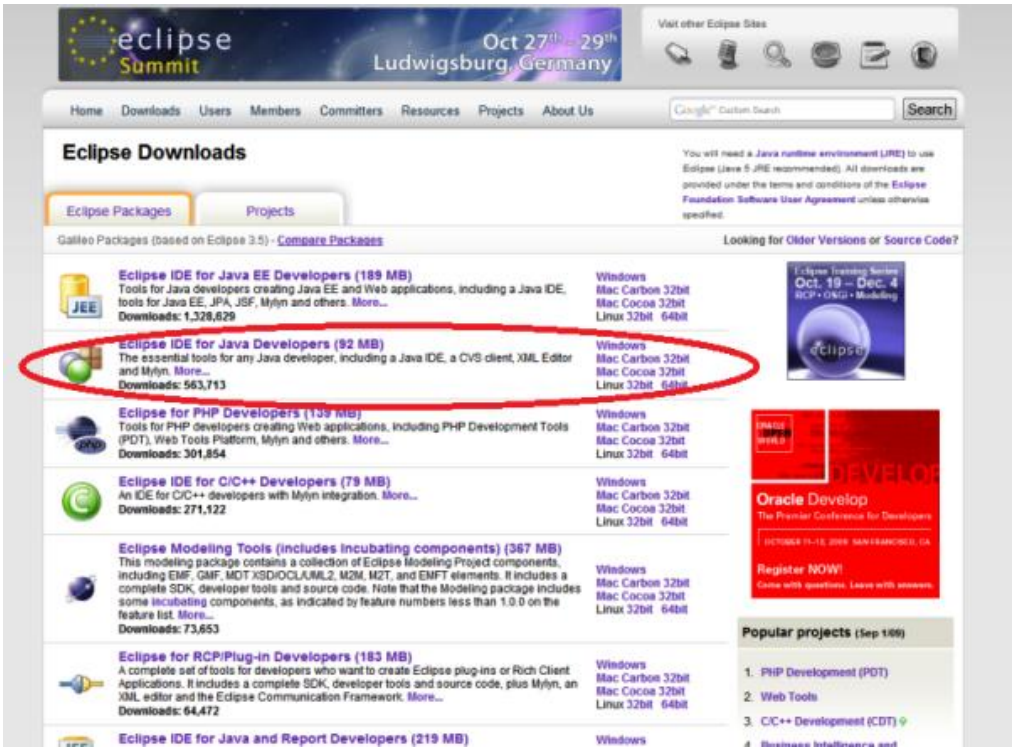
Υπάρχουν πολλά εργαλεία με τα οποία έχουμε την δυνατότητα να τρέξουμε μετρικές, προτίμησα το Eclipse επειδή αποτελεί μια από τις γνωστότερες και καλύτερες σουίτες προγραμματισμού, παρέχει πληθώρα μετρικών μεθόδων (με την προσθήκη του σχετικού plugin) και διατίθεται δωρεάν.

Εγκατάσταση του Eclipse Galileo

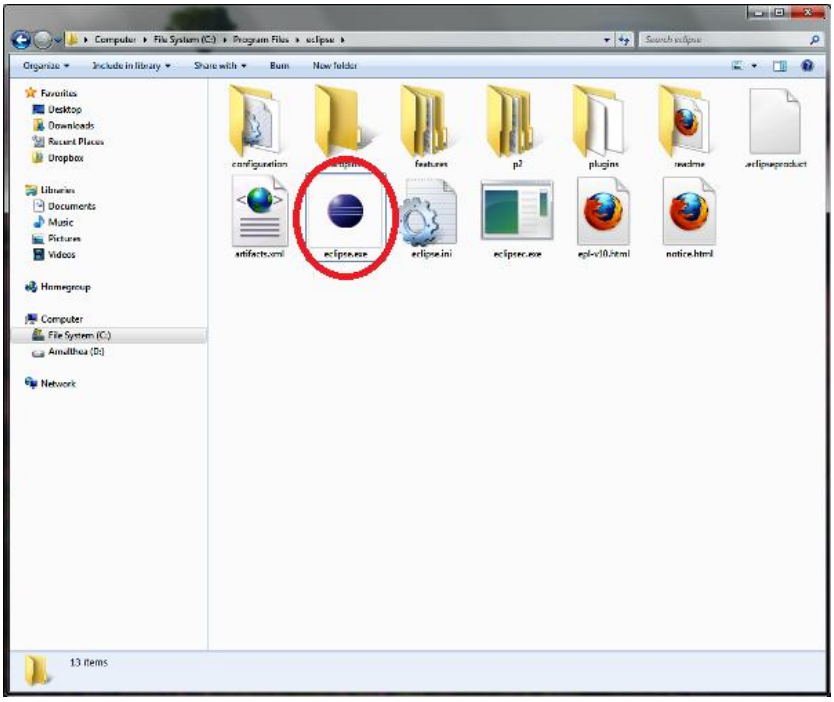
Επισκεπτόμαστε το www.eclipse.org και ακολουθούμε τον σύνδεσμο «Download Eclipse» όπως φαίνεται παρακάτω.



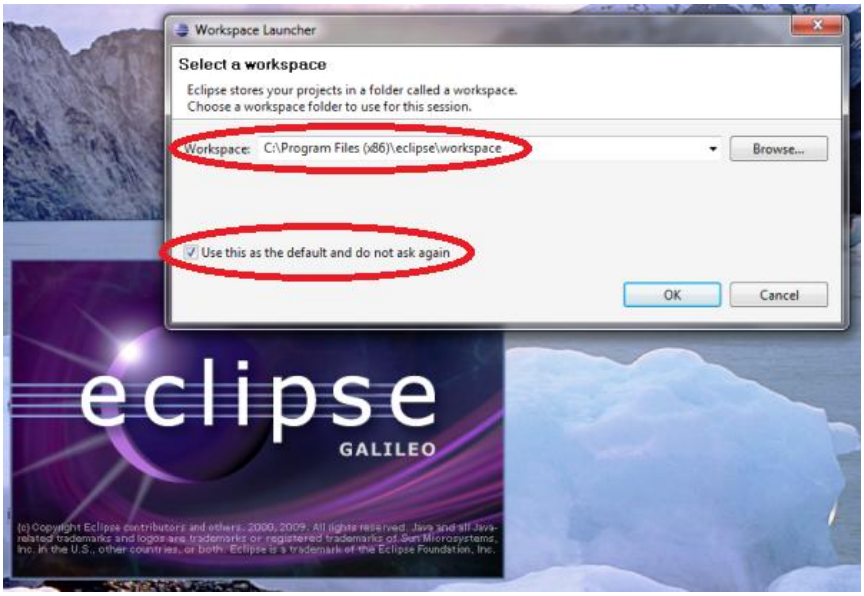
Στη σελίδα που εμφανίζεται επιλέγουμε την νεότερη έκδοση Eclipse IDE for Java Developers και ανάλογα με το λειτουργικό σύστημα που χρησιμοποιούμε κάνουμε κλικ στον κατάλληλο σύνδεσμο στα δεξιά.



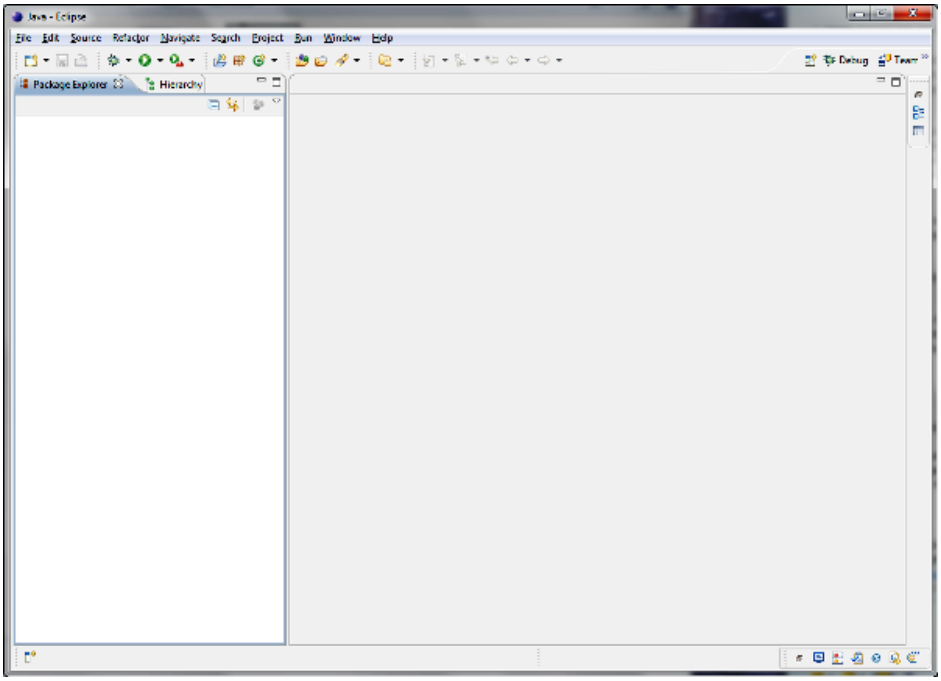
Αφού ολοκληρωθεί η λήψη του πακέτου πρέπει να έχει φθάσει στον υπολογιστή μας ένα αρχείο με όνομα «eclipse-java-galileo-win32.rar». Αποσυμπιέζοντάς το προκύπτει ένας φάκελος με όνομα «eclipse» που περιέχει την εφαρμογή σε εκτελέσιμη μορφή. Μετακινούμε τον φάκελο όπου μας βολεύει (π.χ. C:\Program Files\) και στη συνέχεια τον ανοίγουμε και κάνουμε διπλό κλικ στο «eclipse.exe».



Πριν ανοίξει η κυρίως εφαρμογή καλούμαστε να επιλέξουμε τον φάκελο που θα χρησιμοποιεί το eclipse ως χώρο εργασίας (για να αποθηκεύει δηλαδή τα διάφορα projects μας). Δεν υπάρχει κάποιος περιορισμός ωστόσο πολλοί επιλέγουν να το δημιουργήσουν μέσα στον φάκελο του eclipse (δηλαδή C:\Program Files\eclipse\workspace). Αν δεν θέλετε να ερωτάσθε κάθε φορά που ανοίγετε το Eclipse θα πρέπει να επιλέξετε το checkbox «Use this as the default and do not ask again».



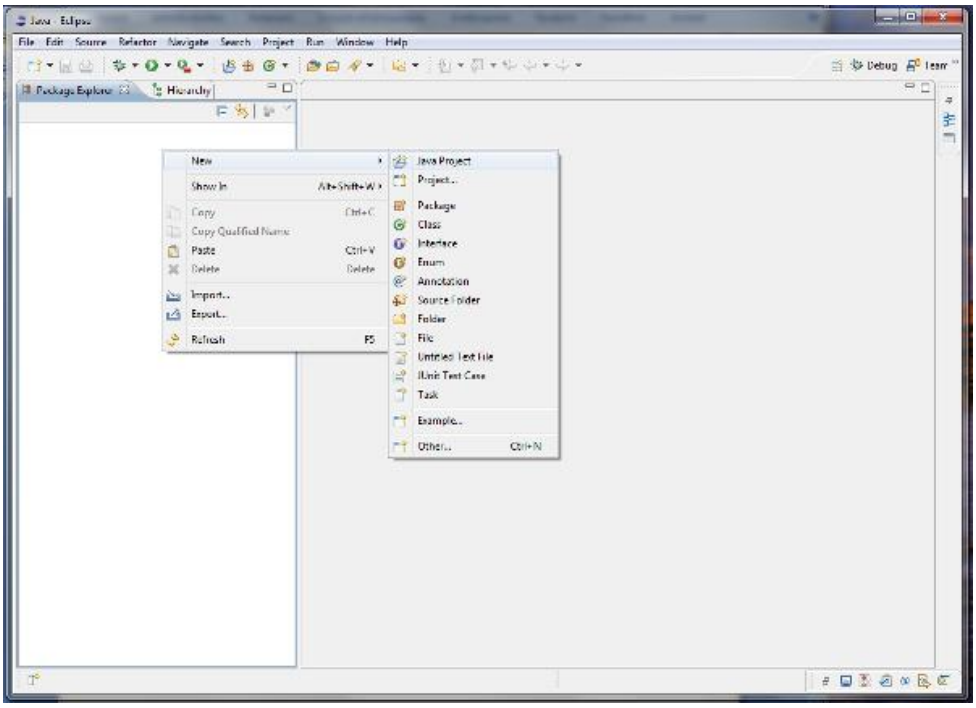
Μετά από αυτό η κύρια οθόνη του eclipse εμφανίζεται.



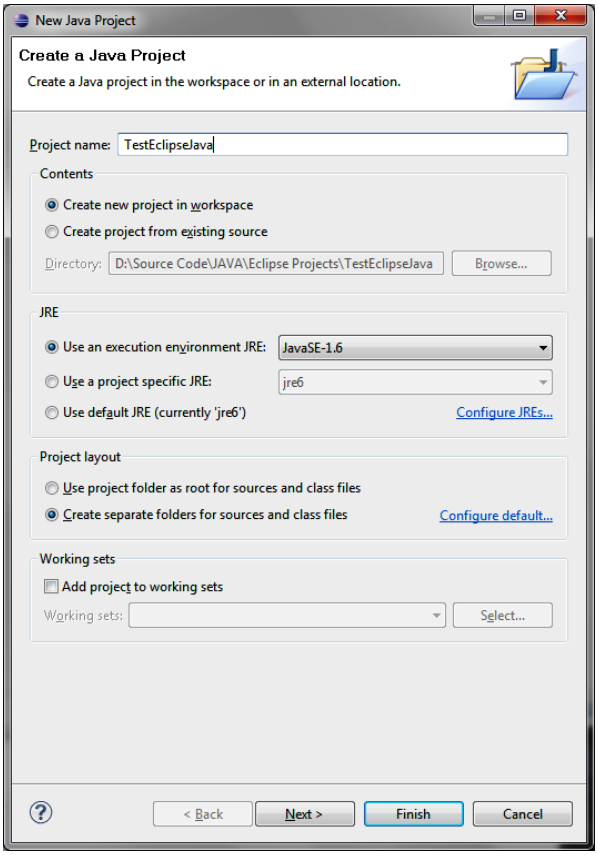
Σε αυτό το σημείο η εγκατάσταση του Eclipse Galileo έχει ολοκληρωθεί.

Δημιουργώντας το πρώτο μας Java πρόγραμμα με το Eclipse

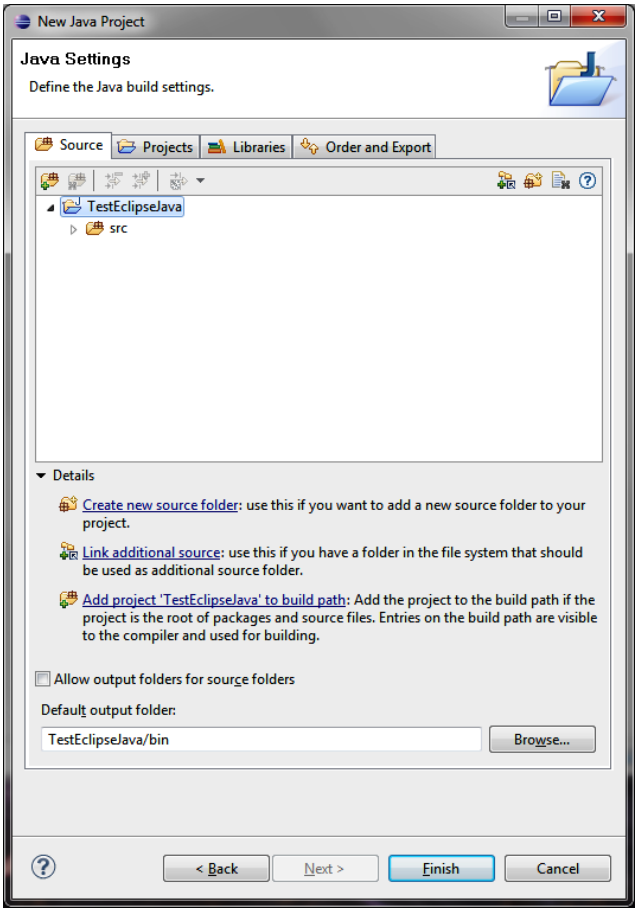
Ξεκινάμε δημιουργώντας ένα καινούριο Java Project (π.χ. TestEclipseJava).



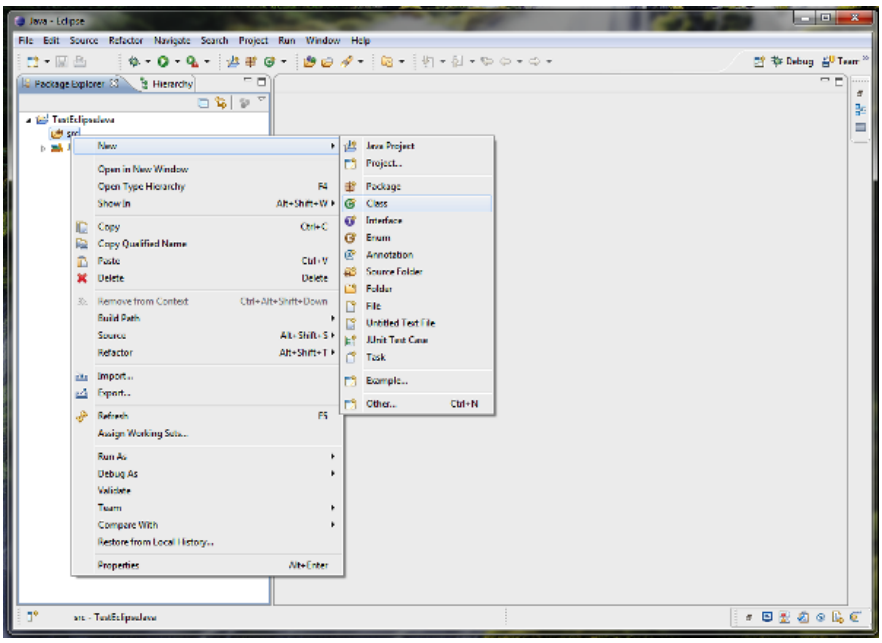
File -> New -> Java Project
Δίνουμε ένα όνομα στο Project και επιλέγουμε «Next»



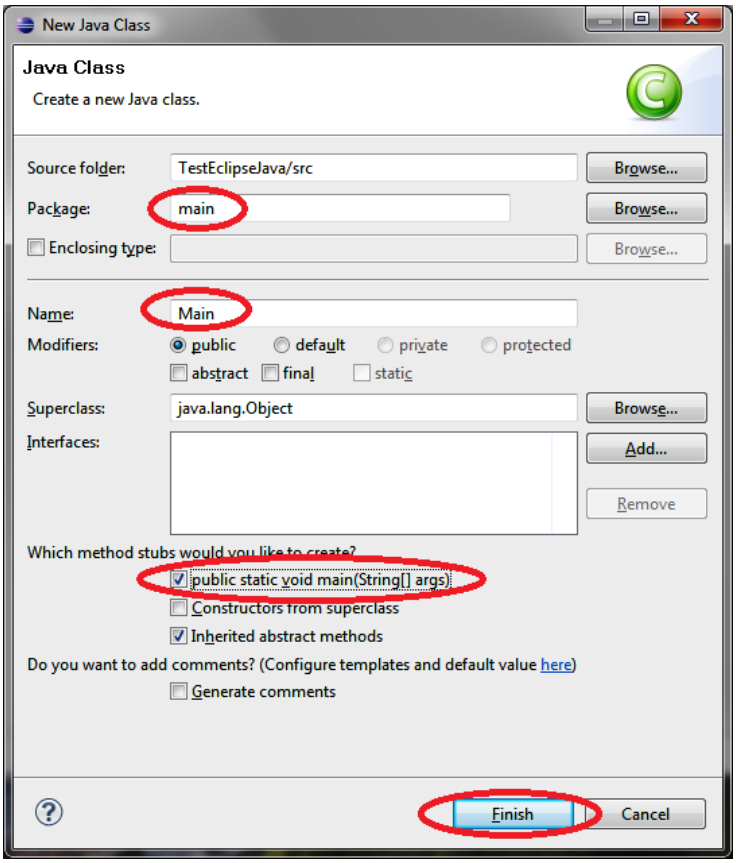
Στην επόμενη οθόνη επιλέγουμε «Finish»



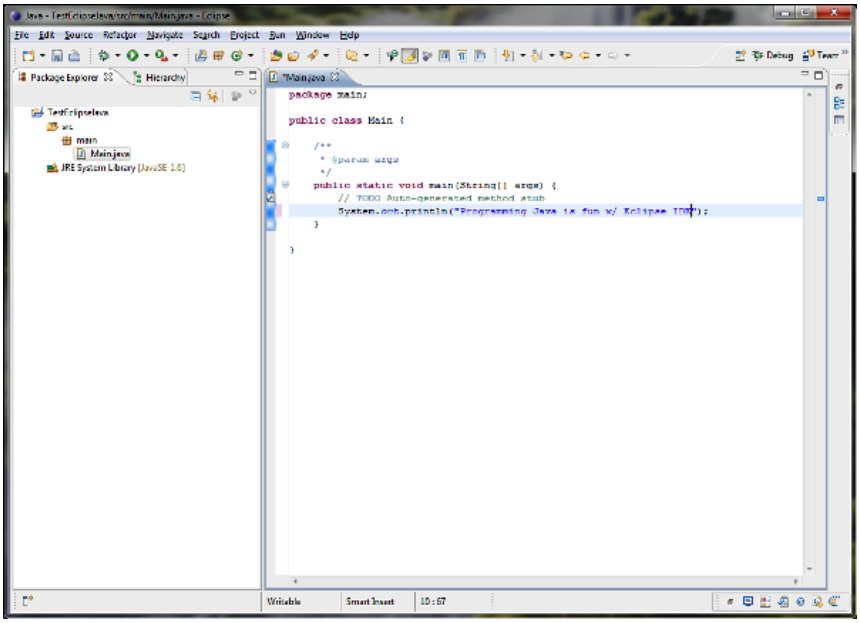
Όπως μπορούμε να δούμε έχουμε επιστρέψει στο κεντρικό παράθυρο στο οποίο έχει δημιουργηθεί το Project με όνομα TestEclipseJava. Κάνοντας διπλό κλικ επάνω του εμφανίζεται το περιεχόμενό του. Κάνουμε δεξί κλικ στον φάκελο «src» και στη συνέχεια επιλέγουμε New -> Class.



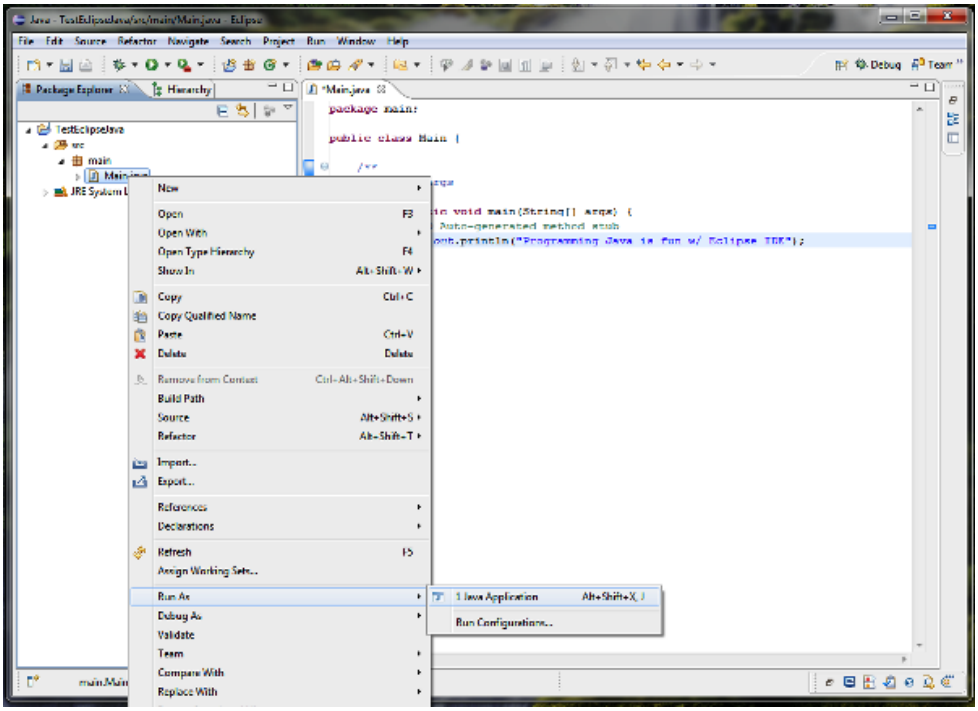
Στο επόμενο παράθυρο ονομάζουμε το πακέτο (main) και την κλάση μας (Main) και αφού ενεργοποιήσουμε το checkbox «public static void main(String [] args)» κάνουμε κλικ στο «Finish».



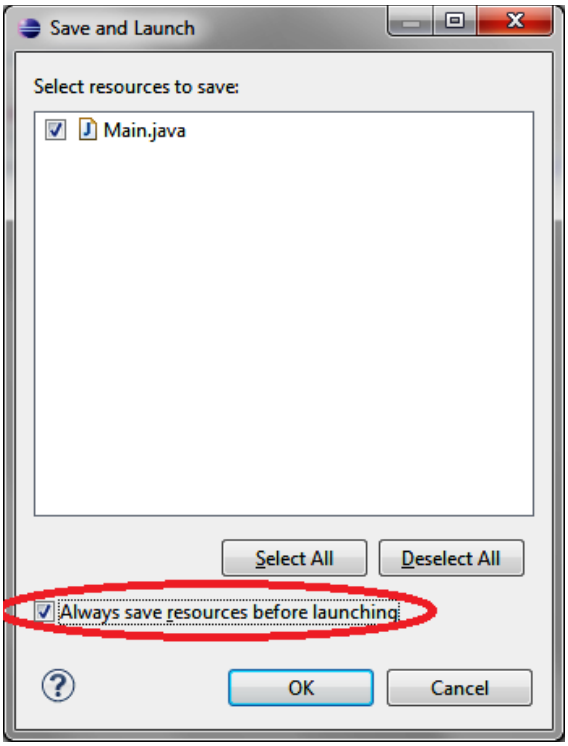
Παρατηρούμε ότι έχει δημιουργηθεί η κλάση Main. Το checkbox που επιλέξαμε μας δημιούργησε αυτόματα και την main μέθοδο. Τώρα το μόνο που μένει είναι να γράψουμε ένα μικρό κομμάτι κώδικα για να τεστάρουμε ότι μεταγλωττίζεται σωστά. Ας προσθέσουμε μέσα στην main μέθοδο την ακόλουθη εντολή: System.out.println("Programming Java is fun w/ Eclipse IDE");



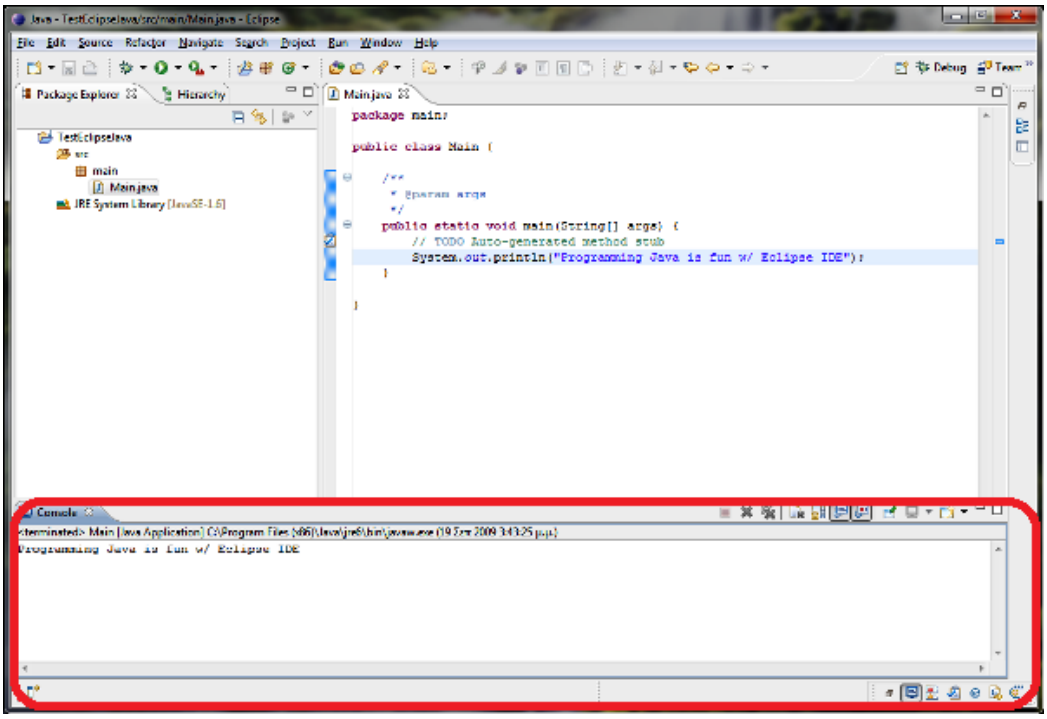
Η παραπάνω εντολή θα τυπώσει, μετά τη μεταγλώττιση, το μήνυμα που υπάρχει μέσα στην παρένθεση. Για να μεταγλωττίσουμε το πρόγραμμά κάνουμε δεξί κλικ στην κλάση Main -> Run As -> Java Application.



Αν τρέχουμε το Eclipse για πρώτη φορά θα δούμε να εμφανίζεται η παρακάτω οθόνη.

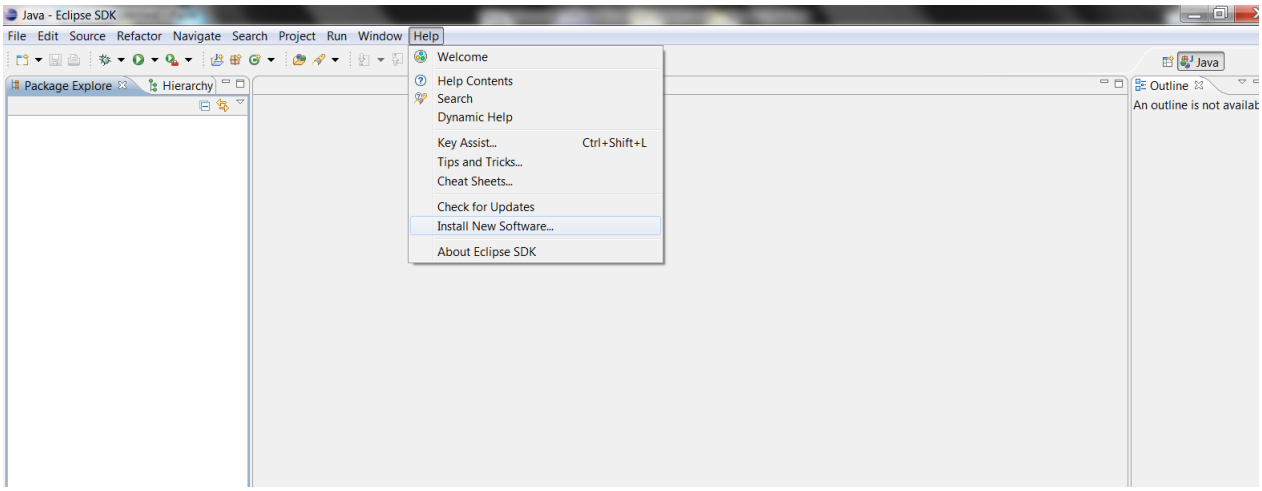


Επιλέγουμε την κλάση Main.java αλλά και το «Always save sources before launching» και κατόπιν επιλέγουμε OK

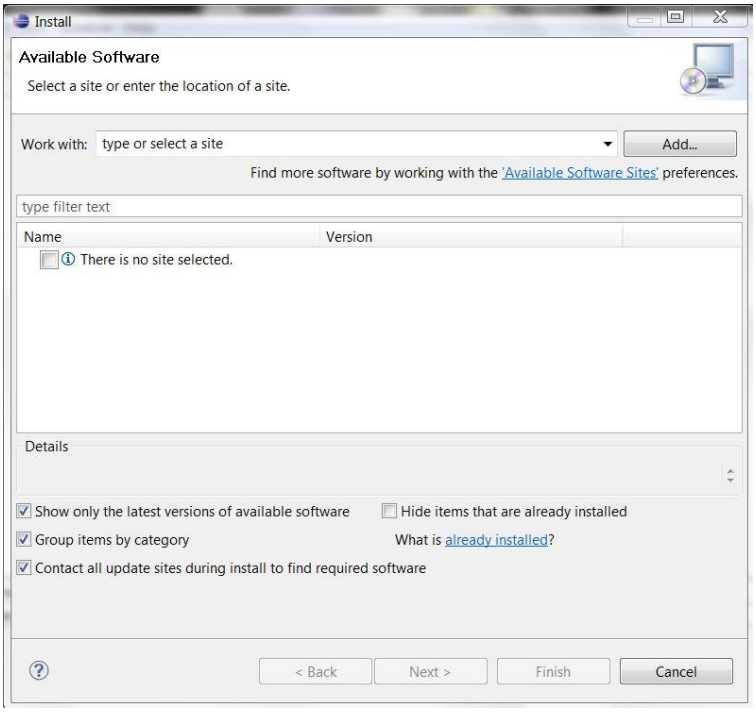


Τέλος εμφανίζεται στην κονσόλα μας ένα μήνυμα αντίστοιχο με αυτό της παραπάνω εικόνας.

Μέτα την περιγραφή της εγκατάστασης και συνοπτικής χρήσης του Eclipse, παρουσιάζεται η διαδικασία εφαρμογής μετρικών μεθόδων. Το Eclipse δεν διαθέτει από μόνο του την δυνατότητα αυτή αλλά την υποστηρίζει με την προσθήκη plugin.



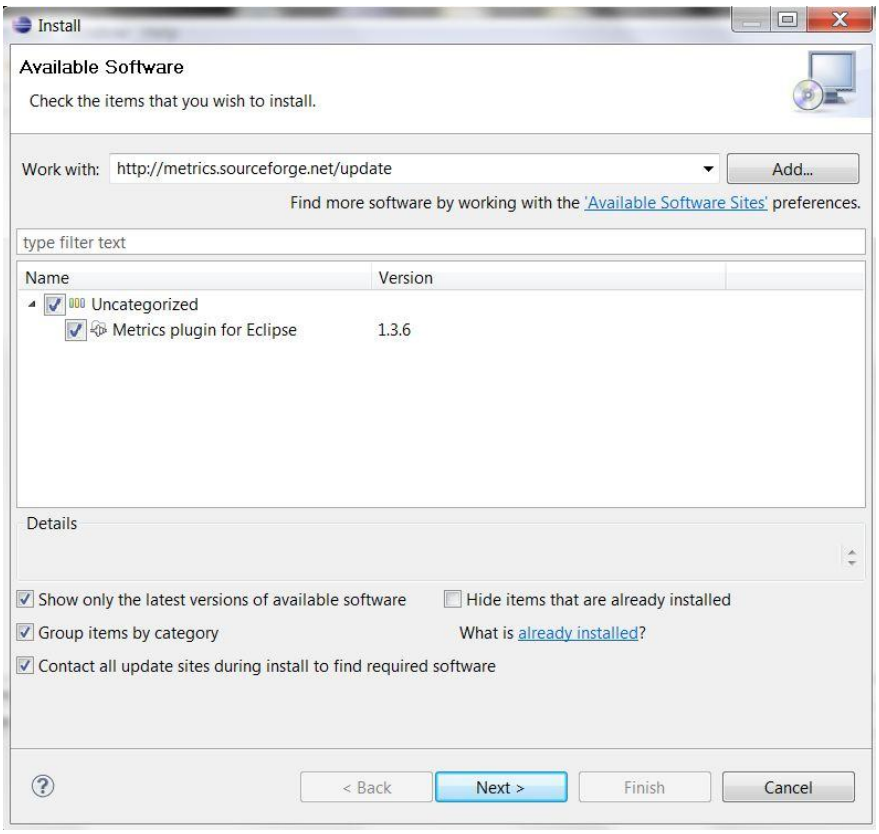
Help -> Install New Software



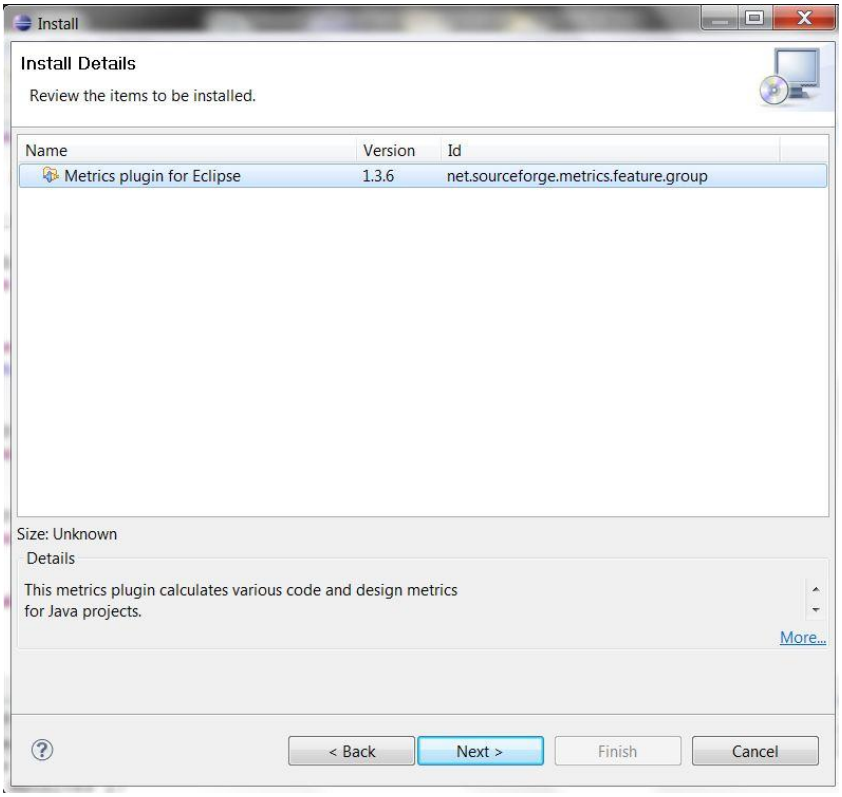
Πηγαίνουμε στο παράθυρο Install και γράφουμε την ακόλουθη διεύθυνση

http://metrics.sourceforge.net/update

Αυτόματα μας εμφανίζει τα σχετιζόμενα με την σελίδα plugin, επιλέγουμε το Metrics 1.3.6 και Next.



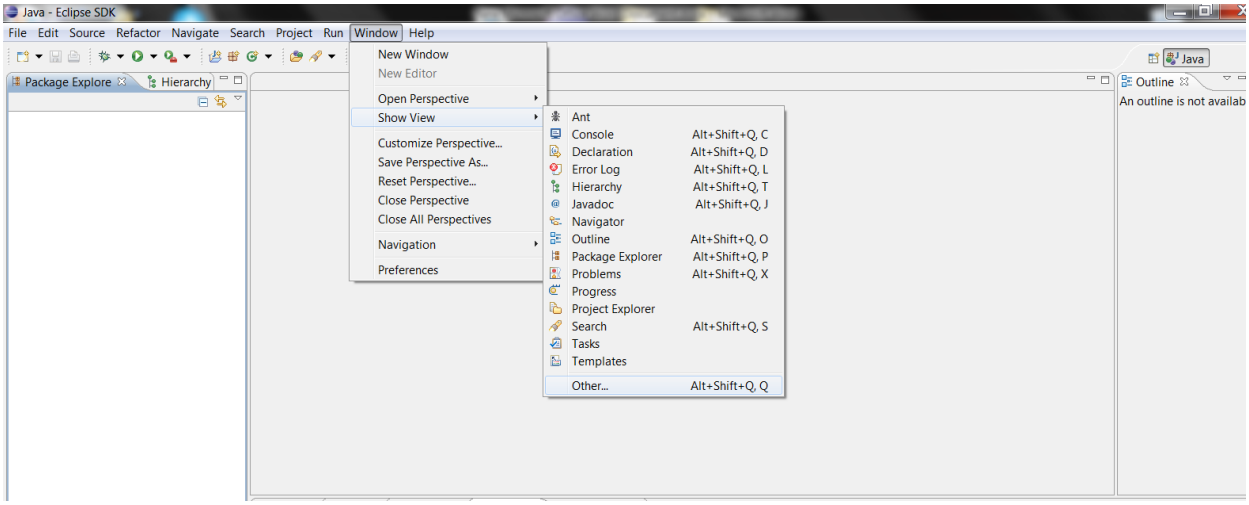
Αμέσως μετά μας εμφανίζει την ακόλουθη εικόνα



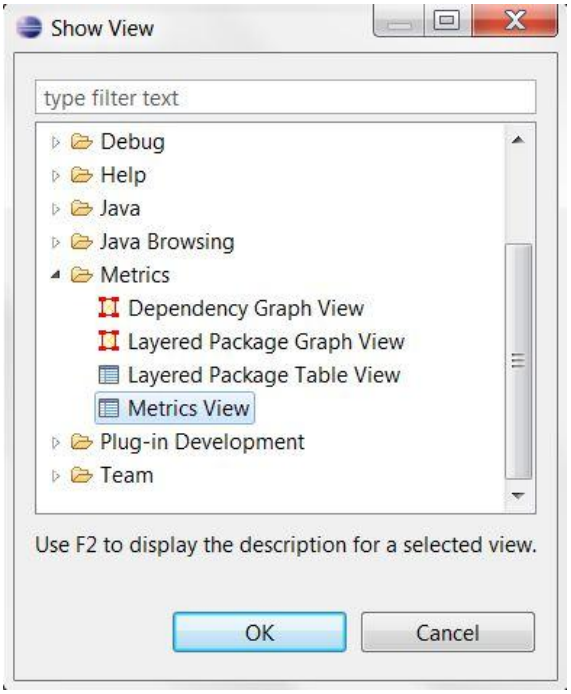
Επιλέγουμε Next αποδεχόμαστε τους όρους χρήσης στο επόμενο παράθυρο και επιλέγουμε Finish. Η εγκατάσταση ολοκληρώθηκε.

Για την εμφάνιση των αποτελεσμάτων των μετρικών είναι απαραίτητο να επιλέξουμε

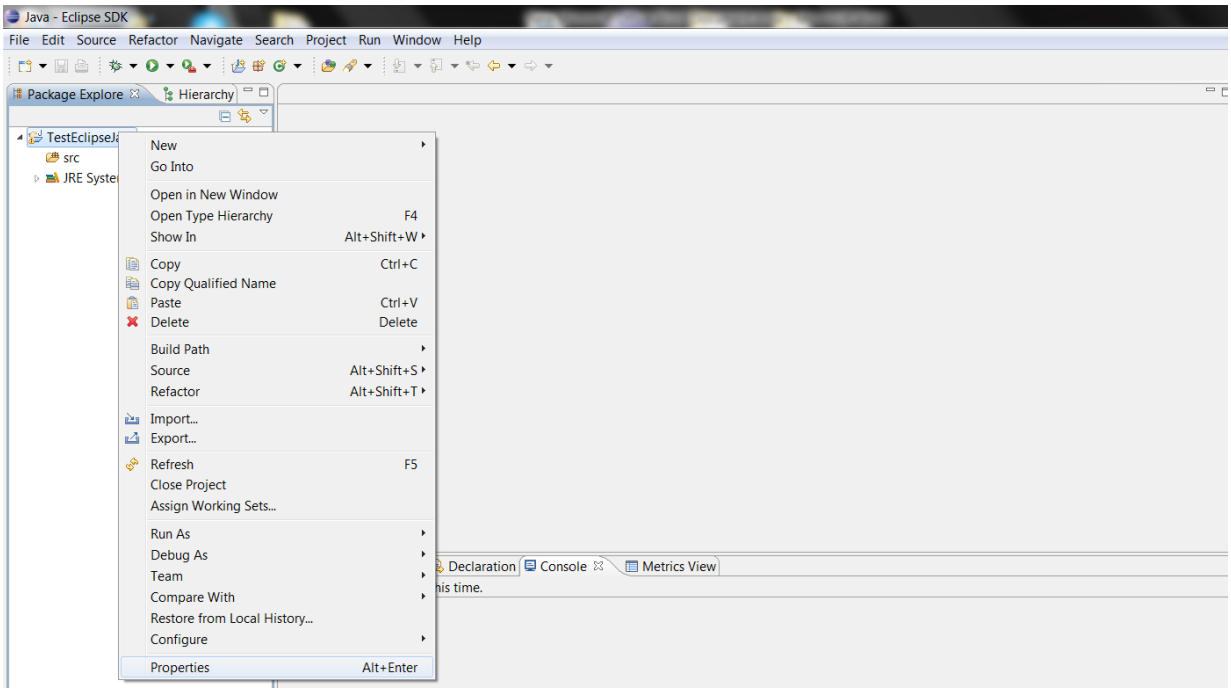
Window -> Show View -> Other



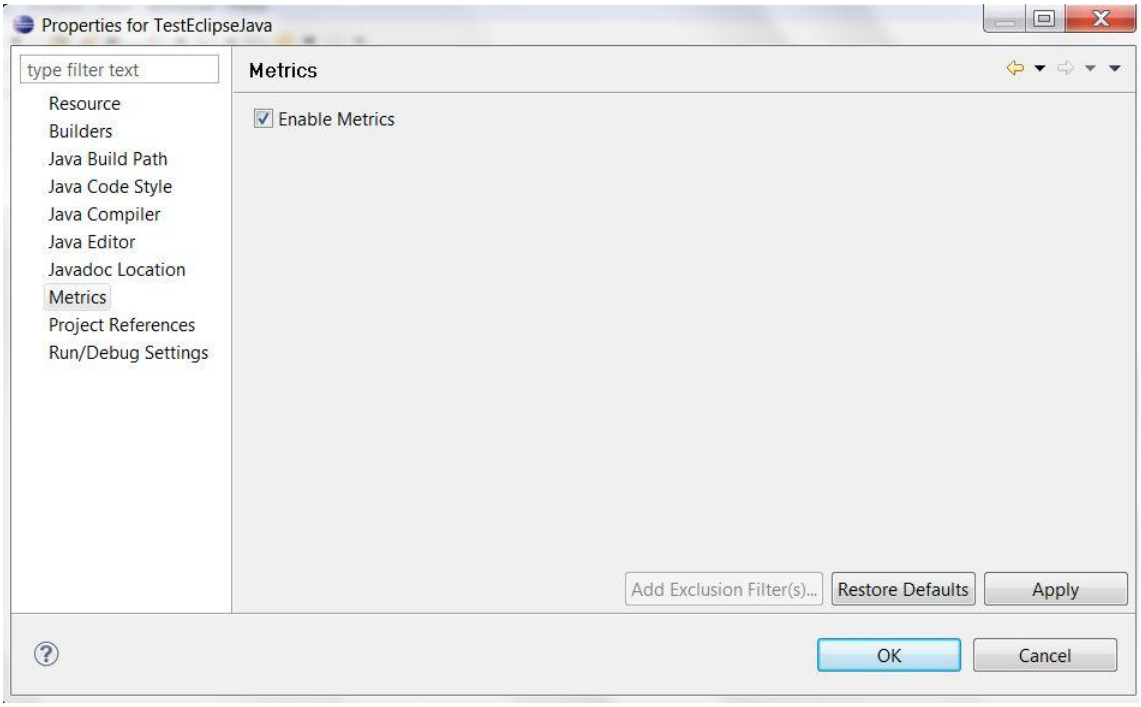
Και από τον φάκελο Metrics -> Metrics view και ok, ώστε να εμφανίζονται.



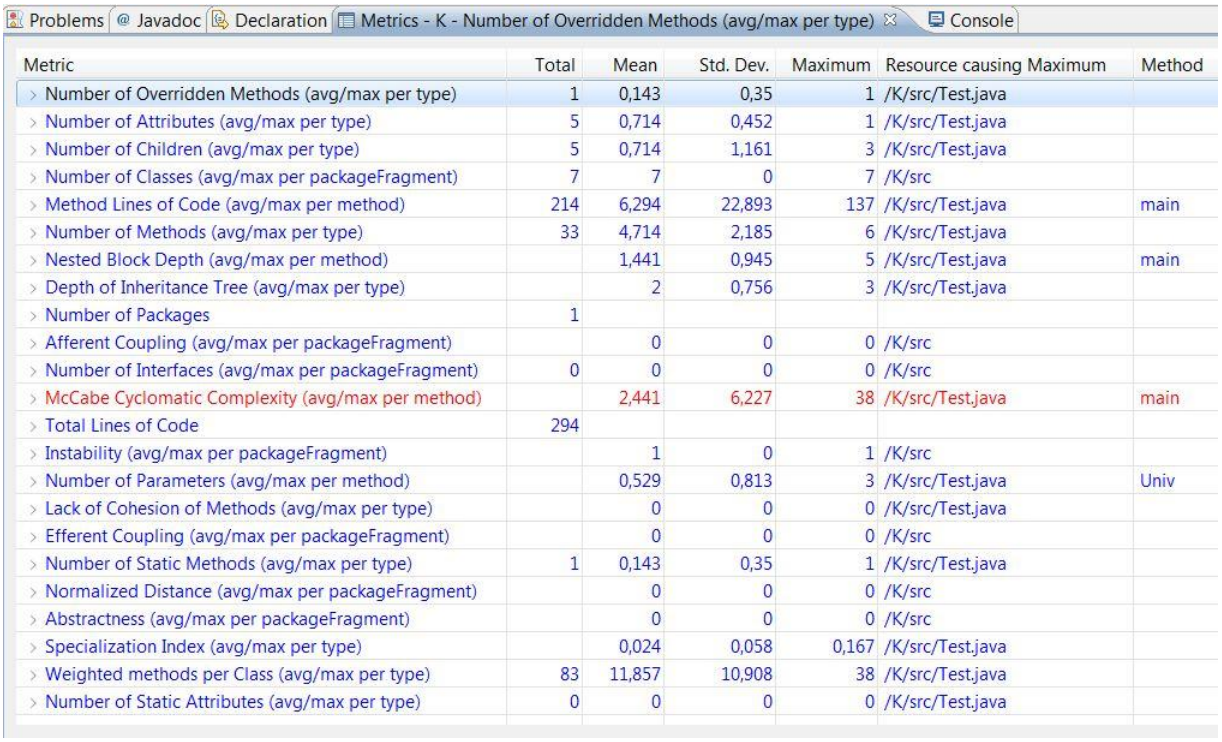
Το τελευταίο που απομένει να κάνουμε είναι δεξί κλίκ στο πακέτο που έχουμε δημιουργήσει (έστω ότι είναι το TestEclipseJava) και από το μενού που θα εμφανίσει να επιλέξουμε Properties.



Από το νέο παράθυρο που θα εμφανιστεί να επιλέξουμε από την αριστερή στήλη Metrics. Θα μας εμφανίσει μία επιλογή Metrics Enabled την οποία και θα επιλέξουμε, ok και έχουμε πλέον την δυνατότητα να τρέξουμε τις μετρικές μεθόδους.



Η εκτέλεση των μετρικών, αυτών που διαθέτει το plugin του Eclipse, οι οποίες είναι αρκετές, γίνεται ταυτόχρονα με την εκτέλεση του προγράμματος. Αφού επιλέξουμε Run και εκτελεστεί το πρόγραμμα, στο tab Console εμφανίζει τα αποτελέσματα του. Δίπλα στο Console υπάρχει το tab Metrics, το οποίο αφού το επιλέξουμε, μας εμφανίζει τις τιμές των μετρικών.

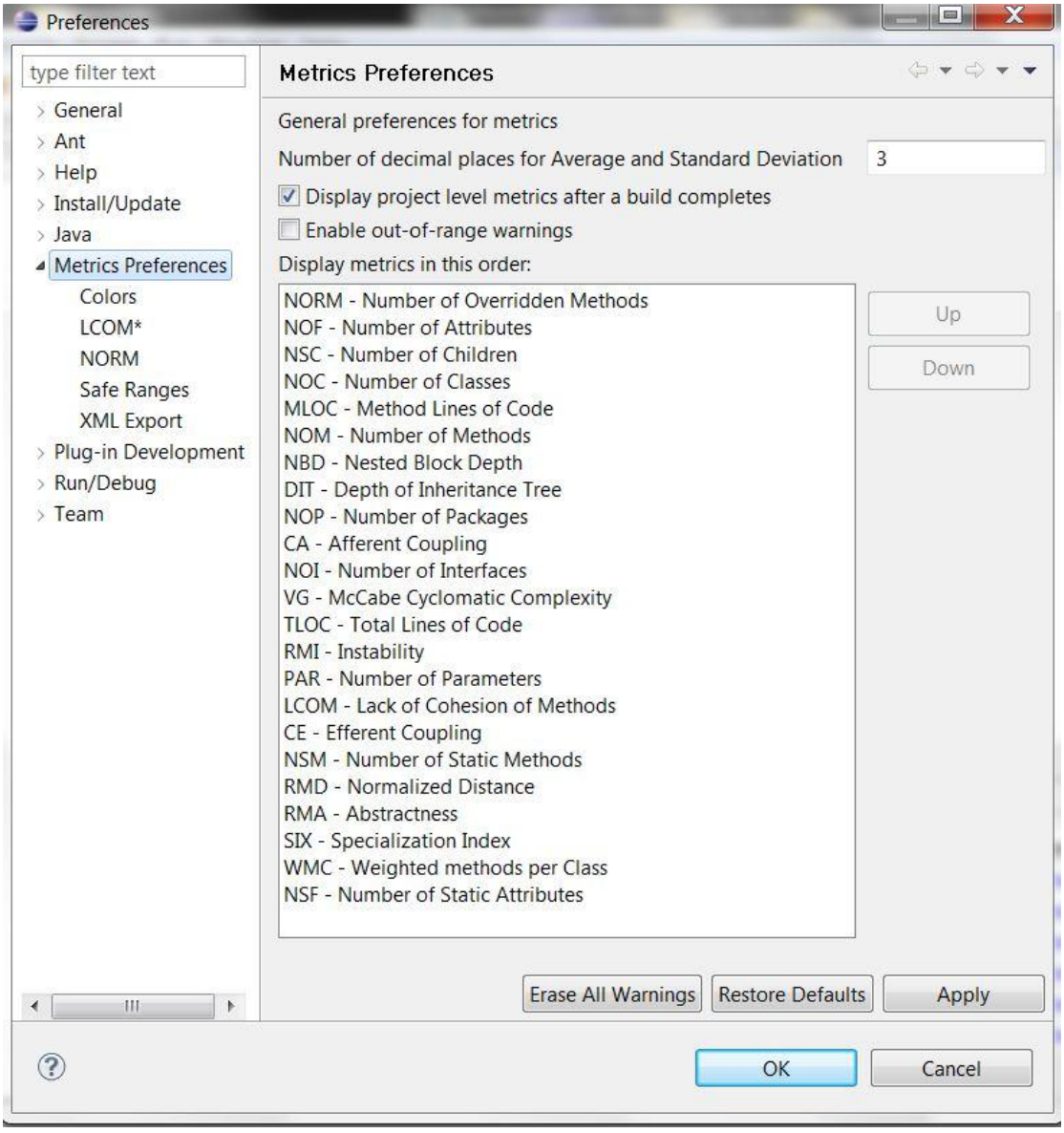


Problems | Javadoc | Declaration | Metrics - K - Number of Overridden Methods (avg/max per type) | Console

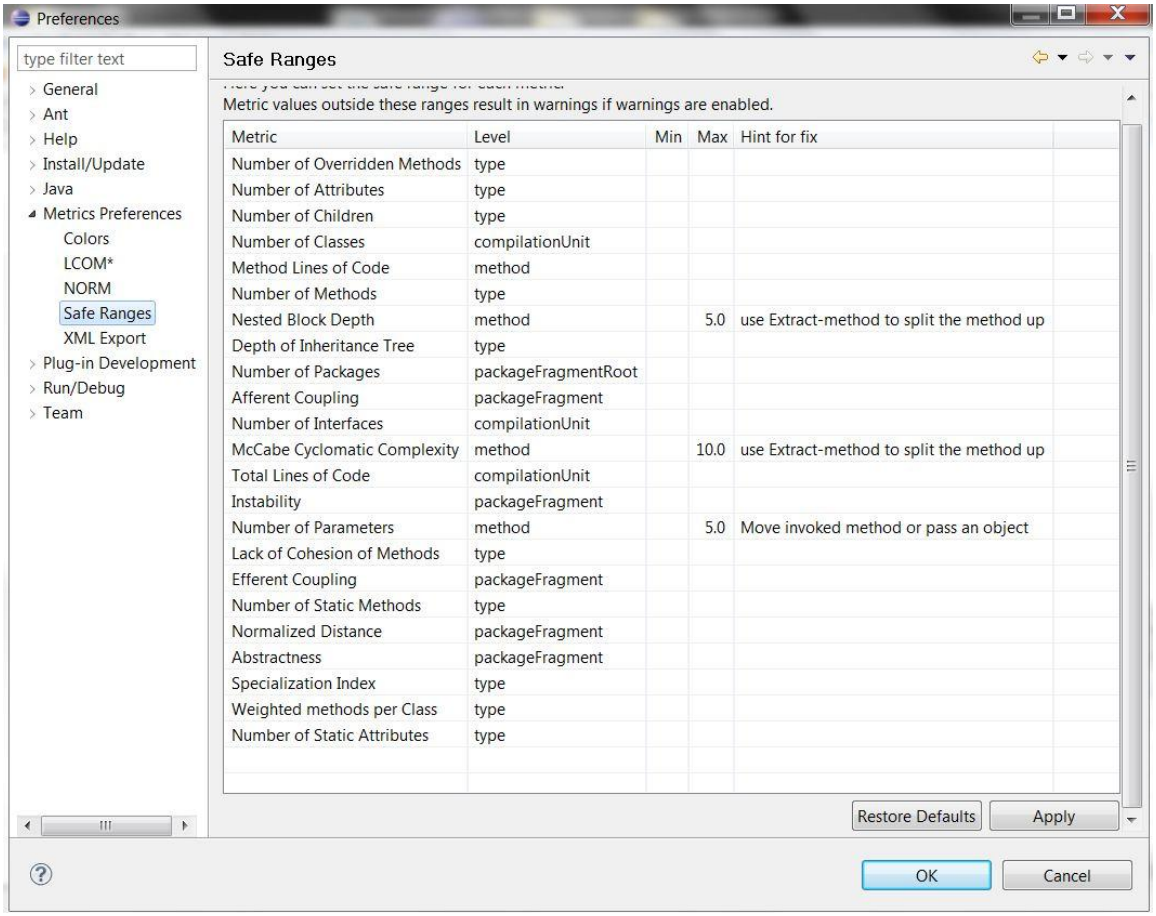
Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per type)	1	0,143	0,35	1	/K/src/Test.java	
Number of Attributes (avg/max per type)	5	0,714	0,452	1	/K/src/Test.java	
Number of Children (avg/max per type)	5	0,714	1,161	3	/K/src/Test.java	
Number of Classes (avg/max per packageFragment)	7	7	0	7	/K/src	
Method Lines of Code (avg/max per method)	214	6,294	22,893	137	/K/src/Test.java	main
Number of Methods (avg/max per type)	33	4,714	2,185	6	/K/src/Test.java	
Nested Block Depth (avg/max per method)		1,441	0,945	5	/K/src/Test.java	main
Depth of Inheritance Tree (avg/max per type)		2	0,756	3	/K/src/Test.java	
Number of Packages	1					
Afferent Coupling (avg/max per packageFragment)		0	0	0	/K/src	
Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/K/src	
McCabe Cyclomatic Complexity (avg/max per method)		2,441	6,227	38	/K/src/Test.java	main
Total Lines of Code	294					
Instability (avg/max per packageFragment)		1	0	1	/K/src	
Number of Parameters (avg/max per method)		0,529	0,813	3	/K/src/Test.java	Univ
Lack of Cohesion of Methods (avg/max per type)		0	0	0	/K/src/Test.java	
Efferent Coupling (avg/max per packageFragment)		0	0	0	/K/src	
Number of Static Methods (avg/max per type)	1	0,143	0,35	1	/K/src/Test.java	
Normalized Distance (avg/max per packageFragment)		0	0	0	/K/src	
Abstractness (avg/max per packageFragment)		0	0	0	/K/src	
Specialization Index (avg/max per type)		0,024	0,058	0,167	/K/src/Test.java	
Weighted methods per Class (avg/max per type)	83	11,857	10,908	38	/K/src/Test.java	
Number of Static Attributes (avg/max per type)	0	0	0	0	/K/src/Test.java	

Όποια μετρική είναι μπλε, με διπλό κλικ μας εμφανίζει αναλυτικά τα αποτελέσματα της για κάθε κλάση ξεχωριστά. Φυσικά εάν επιθυμούμε απλά να τρέξουμε τις μετρικές για ένα πρόγραμμα που ήδη έχουμε μπορούμε την δυνατότητα απλά κάνοντας drug and drop τα αρχεία που το αποτελούν στο source του πακέτου μας.

Επιπλέον έχουμε την δυνατότητα να προσδιορίσουμε τις προτιμήσεις μας από το Window -> Preferences και έπειτα επιλέγοντας Metrics Preferences



Έτσι έχουμε την δυνατότητα να καθορίσουμε την σειρά εμφάνισης των μετρικών και να καθαρίσουμε την βάση δεδομένων της. Ορισμένες μετρικές, μέσω της επιλογής Safe Range, εμφανίζουν τα ασφαλή όρια στα οποία πρέπει να βρίσκονται οι τιμές των μετρικών μεθόδων.



Η πληρότητα του οδηγού χρήσης είναι εκτός των στόχων της παρούσας πτυχιακής, στην οποία καλύπτονται οι απαραίτητες ενέργειες για την επίτευξη της εφαρμογής μετρικών λογισμικού.

Πλήρης οδηγός χρήσης του Eclipse Galileo διατίθεται στις ακόλουθες σελίδες:

- http://wiki.eclipse.org/Galileo
- http://help.eclipse.org/galileo/index.jsp