ΑΛΕΞΑΝΔΡΕΙΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΘΕΣΣΑΛΟΝΙΚΗΣ

Σχεδιάση Βάσει Μοντέλου και Ελεγχος Ποιότητας Μοντέλου: Βιβλιογραφική Αναφορά

Σχολή Τεχνολογικών Εφαρμογών

Τμήμα Πληροφορικής

ΣΙΜΙΤΣΗ ΚΩΝΣΤΑΝΤΙΝΑ

ΑΜ: 05/2861

Επιβλέπων Καθηγητής : κ. ΑΜΠΑΤΖΟΓΛΟΥ ΑΠΟΣΤΟΛΟΣ

Πίνακας περιεχομένων

Πρόλογος 3

Περίληψη 4

Abstract 5

1. Εισαγωγή 6

1.1 Ανάπτυξη - Σχεδίαση λογισμικού 8

1.2 Οι ιδιαιτερότητες της ανάπτυξης λογισμικού 25

2. Σχεδίαση Βάσει Μοντέλου (MDD) 33

2.1 Μοντελοκεντρική Αρχιτεκτονική 40

2.2 Πρότυπα Σχεδίασης 53

3. Έλεγχος ποιότητας μοντέλου (QDD) 60

3.1 Έλεγχος παραγώμενου κώδικα 68

4. Εργαλεία ποιότητας λογισμικού 75

5. Ποιοτικές, μετρήσεις και μετρικές λογισμικού 93

5.1 Κατηγορίες Μετρικών 96

6. Μοντέλα αξιοπιστίας λογισμικού 113

7. Πρότυπα ανάπτυξης και αξιολόγησης λογισμικού 117

8. Συμπεράσματα 128

9. Βιβλιογραφία 130

10. Παράρτημα 135

10.1 Προσαρμογέας – Παράδειγμα 135

10.2 Σύνθετο – Παράδειγμα 140

10.3 Μοναδιαίο – Παράδειγμα 143

10.4 Γέφυρα – Παράδειγμα 144

10.5 Παρατηρητής – Παράδειγμα 145

10.6 Επισκέπτης – Παράδειγμα 149

10.7 Αφηρημένο εργοστάσιο – Παράδειγμα 151

10.8 Στρατηγική – Παράδειγμα 154

10.9 Μέθοδος υπόδειγμα – Παράδειγμα 155

Πρόλογος

Η Ποιότητα Λογισμικού αποτελεί σήμερα ένα πολύ σημαντικό και ενδιαφέρον κεφάλαιο στην Επιστήμη των Υπολογιστών. Με το πέρασμα του χρόνου, καθώς επίσης και με την εξέλιξη της τεχνολογίας η ανάγκη για την εξασφάλιση της ποιότητας σε πρώτο στάδιο, και ακολούθως η ανάγκη για σωστή αξιολόγηση και επιτυχή διασφάλιση της ποιότητας λογισμικού γίνονται όλο και μεγαλύτερες και αποτελούν βασικότατες επιδιώξεις επιχειρήσεων, οργανισμών και προγραμματιστών. Ο όρος Ποιότητας Λογισμικού μπορεί να αποκτήσει πολλές διαστάσεις και ερμηνείες, αναλόγως τις επιδιώξεις, τους στόχους και τις ανάγκες του κάθε χρήστη. Η πτυχιακή αυτή επικεντρώνεται στην ανάλυση της αξιολόγησης και της διασφάλισης της ποιότητας λογισμικού, παρουσιάζοντας τρόπους και μοντέλα, με τη βοήθεια των οποίων είναι εφικτή η αποτελεσματική αξιολόγηση και διασφάλιση της ποιότητας.

Περίληψη

Η πτυχιακή αυτή επικεντρώνεται στην ανάλυση της αξιολόγησης και της διασφάλισης της ποιότητας λογισμικού, παρουσιάζοντας τρόπους και μοντέλα, με τη βοήθεια των οποίων είναι εφικτή η αποτελεσματική αξιολόγηση και διασφάλιση της ποιότητας.

Στο πρώτο κεφάλαιο, αναλύεται ο όρος «σχέδιο λογισμικού» καθώς και τα κριτήρια σχεδίασης λογισμικού. Επιπροσθέτως παρουσιάζονται οι τεχνοτροπίες σχεδίασης λογισμικού και τέλος αναλύονται και τα τέσσερα μοντέλα διατάξεων λογισμικού. Στο δεύτερο κεφάλαιο, περιγράφεται η γραφική γλώσσα μοντελοποίησης γενικού σκοπού Unified Modeling Language (UML), τα διαγράμματα που χρησιμοποιεί για να μοντελοποιήσει τις απαιτήσεις του λογισμικού καθώς επίσης δίνεται μεγάλη έμφαση και στην ανάπτυξη της Μοντελοκεντρικής Αρχιτεκτονικής. Στο τρίτο κεφάλαιο, αναλύεται η έννοια της ποιότητας λογισμικού, με την περιγραφή των πιο γνωστών μοντέλων ποιότητας. Επίσης, παρουσιάζεται συνοπτικά το σύστημα ποιότητας λογισμικού και η χρήση και συνεισφορά του στην ανάπτυξη λογισμικού. Τέλος, προσπαθήσαμε να δουμε τις διαδικασίες που ακολουθούνται ώστε να υπάρχει ένας έλεγχος τόσο της εγκυρότητας όσο και της ορθότητας του παραγόμενου κώδικα. Στο τέταρτο κεφάλαιο, αναπτύσσονται τα εργαλεία ή αλλιώς στατικοί αναλυτές για την πραγματοποίηση ελέγχων με σκοπό των εντοπισμό σφαλμάτων καθώς και τα επιμέρους στάδια της στατικής ανάλυσης. Σκοπός του πέμπτου κεφαλαίου, είναι η διεξοδική ανάλυση ορισμένων μετρήσεων, που πραγματοποιούνται στην διαδικασία λειτουργίας ενός συστήματος ποιότητας λογισμικού, με τη χρήση μετρικών και η παρουσίαση επιλεγμένων μετρικών καθώς και τρόπων μέτρησης. Αναλύονται οι κατηγορίες μετρικών και τέλος, παρουσιάζονται ορισμένες από τις πιο γνωστές μετρικές προϊόντος που εφαρμόζονται στον τελικό κώδικα του λογισμικού. Στο έκτο κεφάλαιο, παραθέτονται τα μοντέλα αξιοπιστίας λογισμικού και τα χαρακτηριστικα αυτών και τέλος στο έβδομο κεφάλαιο γίνεται μια εκτενής αναφορά στα πρότυπα ανάπτυξης και αξιολόγησης λογισμικου και τέλος, αναλύονται οι απαιτήσεις και τα πλεονεκτήματα του προτύπου ISO 9001.

Abstract

The following thesis focuses on the analysis of the evaluation as well as the assurance of software quality, recording modes and models, with the help of which it is possible to effectively evaluate and ensure quality.

The first chapter discusses the term "software design" and the software design criteria. Additionally presents the software design techniques, and finally all four models of software provisions analyzed. The second chapter describes the general-purpose graphical modeling language Unified Modeling Language (UML), the diagrams used to model software requirements placing great emphasis on the development of Model-based Architecture. The third chapter analyzes the concept of software quality, with a description of the most popular quality models. It also outlines the quality system and software use and its contribution to the software development. Finally, we tried to see the procedures followed in order to have a check of both the validity and correctness of the generated code. In the fourth chapter, tools or otherwise static analyzers are developed to perform checks aiming at the debugging and the various stages of static analysis. The purpose of the fifth chapter is a thorough analysis of some measurements made in the process of operating a system software quality, using metrics and presenting selected metrics and measurement methods. Furthermore, categories of metrics are analyzed and finally some of the most famous product metrics applied to the final software code presented. In the sixth chapter, software reliability models and their characteristics are listed and finally in the seventh chapter is an extensive report on development standards and evaluation of software and in conclusion in the analysis of the requirements and benefits of ISO 9001.

1. Εισαγωγή

Στις μέρες μας όλοι μιλούν για την επανάσταση της πληροφορικής και των τηλεπικοινωνιών, για τις σύγχρονες εφαρμογές των υπολογιστών, που θα αλλάξουν τη ζωή μας, για το Internet και τις τεχνικές αλλά και κοινωνικές πλευρές της διάδοσής του. Η επιστήμη των υπολογιστών έχει, αναμφίβολα, προοδεύσει δραματικά. Τίποτα όμως δε θα μπορούσε να είναι ορατό και εφαρμόσιμο σε ευρεία κλίμακα, αν δεν υπήρχε το κατάλληλο λογισμικό.

Το λογισμικό είναι εκείνο το συστατικό που, αν και το ίδιο δεν έχει χειροπιαστή υπόσταση, μπορεί να καταστήσει μια υπολογιστική μηχανή χρήσιμη στον άνθρωπο. Όσο περισσότερο αναπτύσσονται οι ηλεκτρονικοί υπολογιστές, όσο περισσότερες δυνατότητες αποκτούν, τόσο περισσότερες γίνονται οι απαιτήσεις του ανθρώπου από αυτούς, τόσο περισσότερο σύνθετες εργασίες τους αναθέτουμε. Η ικανοποίηση των απαιτήσεων αυτών γίνεται με τη βοήθεια του λογισμικού, η πολυπλοκότητα του οποίου -αναπόφευκτα- συνεχώς και αυξάνεται.

Αν συνδυάσει κανείς το γεγονός ότι το λογισμικό λειτουργεί σε υπολογιστικές μηχανές, οι οποίες συνεχώς εξελίσσονται και ότι ικανοποιεί απαιτήσεις οι οποίες γίνονται ολοένα περισσότερες, πιο πολύπλοκες και μεταβάλλονται ταχύτατα με το χρόνο, με τη μη χειροπιαστή φύση του λογισμικού, τότε μπορεί εύκολα να υποψιαστεί ότι η κατασκευή του είναι από μόνη της μια ιδιαίτερα δύσκολη υπόθεση. Πράγματι, από τα πρώτα χρόνια της διάδοσης των υπολογιστών, όχι ακόμα σε ευρεία κλίμακα, εκδηλώθηκαν σημαντικά προβλήματα στην κατασκευή λογισμικού. Είναι χαρακτηριστικό ότι ο όρος Τεχνολογία Λογισμικού (Software Engineering) εισήχθη για πρώτη φορά μαζί με τον όρο Κρίση Ποιότητας Λογισμικού (Software Quality Crisis), το 1968. Έκτοτε, οι κατασκευαστές λογισμικού και οι ακαδημαϊκοί ερευνητές προσπαθούν να προτείνουν τρόπους ώστε να γίνεται σωστά και αποτελεσματικά η κατασκευή λογισμικού καλής ποιότητας. Σε κάθε εποχή, ο ενθουσιασμός και οι τυμπανοκρουσίες της έλευσης μιας νέας προσέγγισης έδιναν τη θέση τους στην προσγειωμένη πραγματικότητα. Τα προβλήματα στην ανάπτυξη του λογισμικού συμπεριφέρονταν λίγο ως πολύ σαν λερναία ύδρα, όπου στη θέση κάθε κεφαλιού που κοβόταν φύτρωναν περισσότερα. Σήμερα, η αναζήτηση του «καλύτερου» τρόπου κατασκευής λογισμικού θεωρείται ιδεατή επιδίωξη. Έχει καταστεί σαφές

ότι δεν υπάρχει καμία «χρυσή συνταγή» και ότι η ανάπτυξη του ποιοτικού λογισμικού οφείλει να είναι μια ιδιαίτερα ευέλικτη διαδικασία, εύκολα προσαρμόσιμη στις εκάστοτε συνθήκες, αλλά και στη φύση τού εκάστοτε προβλήματος στην επίλυση του οποίου χρησιμοποιείται λογισμικό.

1.1 Ανάπτυξη - Σχεδίαση λογισμικού

Αυτό που ζητείται από τη σχεδίαση είναι ένας τρόπος περιγραφής της κατασκευής του λογισμικού έτσι ώστε αυτό να ικανοποιεί τις προδιαγραφές που έχουν τεθεί, δηλαδή να μπορεί να εκτελεί τις επιθυμητές λειτουργίες και να έχει τα επιθυμητά χαρακτηριστικά. Η ύπαρξη των προδιαγραφών είναι αναγκαία για να ξεκινήσει η σχεδίαση και επιβάλλεται από τις αρχές της τεχνολογίας λογισμικού. Η περιγραφή αυτή, που παράγεται κατά τη σχεδίαση, ονομάζεται «σχέδιο λογισμικού».

Σε αυτό το σημείο όμως, καλό είναι να ορίσουμε αρχικά την έννοια τού σχεδίου λογισμικού. Ουσιαστικά μιλώντας για «σχέδιο λογισμικού» μιλάμε για την περιγραφή των μονάδων που αποτελούν το λογισμικό, των συσχετίσεων μεταξύ τους, της διάταξής τους, καθώς και της εσωτερικής τους λεπτομέρειας.

Η παραγωγή του σχεδίου του λογισμικού είναι αναγκαία προκειμένου να γίνει δυνατή η κατασκευή του, όπως, άλλωστε, ισχύει για κάθε τεχνικό έργο. Η λύση στο πρόβλημα της σχεδίασης λογισμικού δεν είναι καθόλου εύκολη. Για κάθε προδιαγραφή δεν είναι παράλογο να θεωρούμε ότι μπορούμε να κατασκευάσουμε περισσότερα του ενός σχέδια, δηλαδή να θεωρούμε ότι μπορεί να υλοποιηθεί με περισσότερους του ενός τρόπους. Μερικές από τις σημαντικότερες πλευρές του προβλήματος της σχεδίασης είναι οι ακόλουθες:

1. Με ποια στρατηγική πρέπει να αντιμετωπίσουμε τη μετάβαση από τις προδιαγραφές στη σχεδίαση έτσι ώστε η εργασία μας να είναι αποτελεσματική;
2. Ποιος από τους τρόπους που μπορούμε να σκεφτούμε για την υλοποίηση μιας προδιαγραφής είναι ο καλύτερος και πώς τεκμηριώνεται αυτό;
3. Σε ποιο βαθμό δεσμεύεται η σχεδίαση από το περιβάλλον (γλώσσα προγραμματισμού, εργαλεία, λειτουργικό σύστημα) στο οποίο θα γίνει η κατασκευή του προγράμματος;
4. Ποια είναι η περισσότερο επαρκής, δηλαδή κατάλληλη, χωρίς να είναι ούτε ελλιπής ούτε φλύαρη, περιγραφή του σχεδίου του λογισμικού;
5. Πώς εξασφαλίζεται η ποιότητα του παραγόμενου λογισμικού μέσα από τις εργασίες που λαμβάνουν χώρα κατά τη σχεδίαση;

Σκοπός της σχεδίασης είναι να δώσει την καλύτερη δυνατή -στις εκάστοτε συνθήκες- απάντηση στα παραπάνω ερωτήματα. Ας σημειωθεί ότι δεν υπάρχει η «καλύτερη», κατ'' απόλυτη έννοια, λύση, παρά μόνο η «καλύτερη δυνατή στις εκάστοτε συνθήκες». Η εποχή που επιδίωξη των μηχανικών λογισμικού ήταν η εύρεση της απόλυτα καλύτερης και γενικής λύσης στο πρόβλημα της σχεδίασης έχει δώσει τη θέση της στο ρεαλισμό της επιδίωξης της βέλτιστης λύσης μέσα σε κάθε συγκεκριμένο περιβάλλον κατασκευής λογισμικού. Παρά τον υποκειμενισμό που, γενικά, διέπει το χαρακτηρισμό ενός σχεδίου λογισμικού, είναι χρήσιμο να συμφωνούνται κριτήρια ποιότητας, στα οποία να αποδίδεται η προσήκουσα κατά περίπτωση βαρύτητα. Τέσσερα τέτοια κριτήρια είναι τα ακόλουθα:

-Το σχέδιο πρέπει να ικανοποιεί όλες τις προδιαγραφές των απαιτήσεων από το λογισμικό (λειτουργικές και μη).

- Το σχέδιο πρέπει να περιγράφει πλήρως όλες τις πλευρές του λογισμικού: δεδομένα, λειτουργίες και συμπεριφορά, όπως αυτές θεωρούνται από την πλευρά του κατασκευαστή.

- Το σχέδιο πρέπει να είναι εύκολα κατανοητό από αυτούς που θα κληθούν να συγγράψουν τον πηγαίο κώδικα, δηλαδή τους προγραμματιστές.

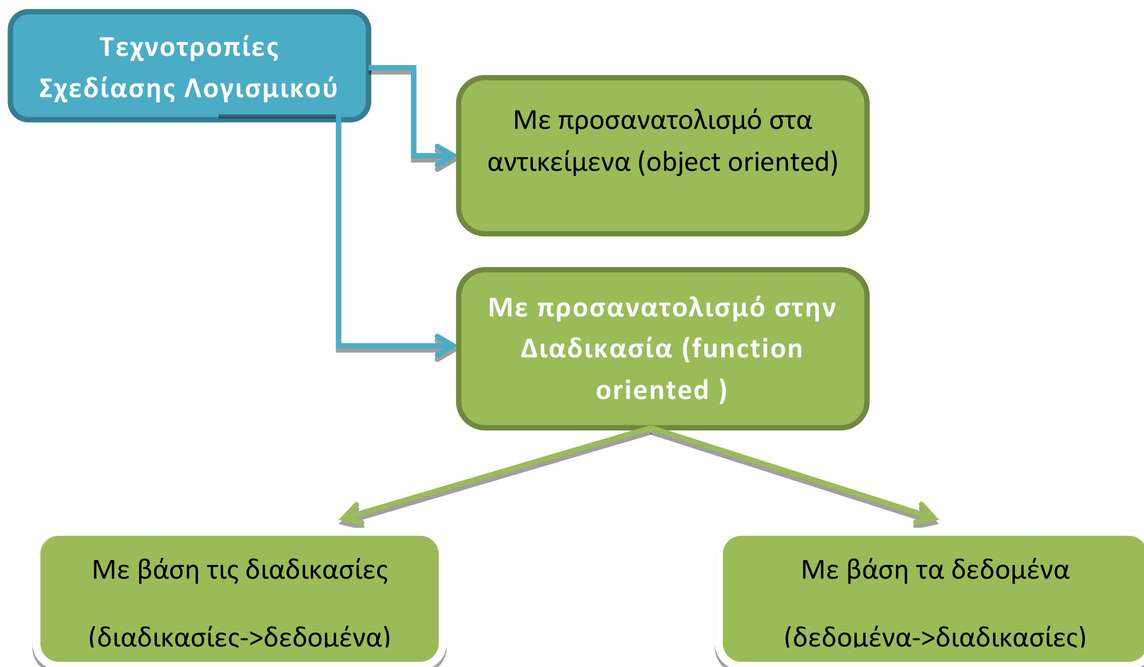
- Το σχέδιο δεν πρέπει να περιέχει σφάλματα.

Τα παραπάνω, εκτός από κριτήρια, μπορούν να ακούγονται εξίσου εύκολα και ως ευχές. Μια πρακτική σχεδίασης πρέπει να στοχεύει στην καθοδήγηση του κατασκευαστή στην παραγωγή σχεδίου λογισμικού το οποίο να πληρεί στο μέγιστο δυνατό βαθμό τα παραπάνω κριτήρια. Αυτό δεν είναι πάντα εύκολο, καθότι ενδέχεται να υπάρχουν και εσωτερικές αντιφάσεις μέσα στα κριτήρια (π.χ. αντιφάσκουσες μεταξύ τους προδιαγραφές), αλλά και άλλοι πρακτικοί λόγοι (π.χ. απαγορευτικό από τον προϋπολογισμό κόστος πλήρους λεπτομερούς περιγραφής του σχεδίου). Η τεχνολογία λογισμικού παρέχει μεθοδολογίες σχεδίασης οι οποίες αποτελούν κατευθυντήριες γραμμές για το σχεδιαστή. Συνδυάζοντας τη γνώση, την εμπειρία και τον αυτοσχεδιασμό, αλλά και με τη χρήση των κατάλληλων εργαλείων ανάπτυξης λογισμικού, ο σχεδιαστής μπορεί να αντιμετωπίσει την πρόκληση -γιατί περί πρόκλησης πρόκειται- ικανοποίησης των κριτηρίων αυτών.

Το πρόβλημα της σχεδίασης λογισμικού μπορεί να αντιμετωπιστεί με διάφορες στρατηγικές προσεγγίσεις. Οι διάφορες μεθοδολογίες που έχουν παρουσιαστεί μπορούν να ενταχθούν σε δύο μεγάλες κατηγορίες: τις προσανατολισμένες στις διαδικασίες (function oriented) και τις προσανατολισμένες

στα αντικείμενα (object oriented). Μια σύντομη αναφορά στα χαρακτηριστικά των δύο κατηγοριών δίνεται στην ενότητα αυτή. Η απάντηση στην ερώτηση «Τι θα κάνει το λογισμικό;» για μεγάλο χρονικό διάστημα δινόταν με τη μορφή μιας ιεραρχίας διαδικασιών, συναρτήσεων και άλλων ενεργών μονάδων λογισμικού, η οποία επιδρούσε σε ένα σύνολο ανεξάρτητων δεδομένων. Αν και από τη δεκαετία του' '60 ακόμα είχαν παρουσιαστεί και άλλες απόψεις, η προσέγγιση αυτή επικράτησε και πάνω της γεννήθηκαν διάφορες μεθοδολογίες σχεδίασης λογισμικού, οι οποίες συγκρότησαν την οικογένεια μεθοδολογιών δομημένης σχεδίασης.

Η «άλλη άποψη» προσπαθούσε να απαντήσει στο ίδιο ερώτημα χωρίς να διακρίνει τα δεδομένα από τις διαδικασίες, προτείνοντας ένα σύνολο συνεργαζόμενων οντοτήτων που περιλαμβάνουν στο ίδιο κέλυφος και δεδομένα και διαδικασίες. Οι οντότητες αυτές ονομάστηκαν αντικείμενα (objects). Οι μεθοδολογίες που γεννήθηκαν πάνω σε αυτή την προσέγγιση ονομάστηκαν προσανατολισμένες στα αντικείμενα (object–oriented). Και στις δύο περιπτώσεις, ο σχεδιαστής καταπιάνεται και με τις διεργασίες και με τα δεδομένα, αποδίδοντάς τους όμως διαφορετική βαρύτητα σε κάθε περίπτωση. Οι δύο οικογένειες φαίνονται στο σχήμα παρακάτω.



Τεχνοτροπίες σχεδίασης λογισμικού

Οι μεθοδολογίες που ακολουθούν αυτή την προσέγγιση προτείνουν τρόπους αποσύνθεσης του συστήματος από πάνω προς τα κάτω (top–down) σε μια ιεραρχία διαδικασιών, συναρτήσεων και άλλων ενεργών μονάδων λογισμικού. Όσο κατεβαίνει κανείς στην ιεραρχία αυτή, τόσο μεγαλύτερη λεπτομέρεια συναντά, μέχρις ότου φτάσει στις απλές δομικές μονάδες, δηλαδή τις εντολές της γλώσσας προγραμματισμού. Η προσέγγιση αυτή παρουσιάζεται στο παρόν βιβλίο. Γνωστές μεθοδολογίες που ανήκουν στην οικογένεια αυτή έχουν προταθεί από πολλούς συγγραφείς και για ενημέρωση μπορείτε να ανατρέξετε στη βιβλιογραφία. Οι περισσότερες των προσεγγίσεων αυτών επικεντρώνουν την προσοχή τους στις διαδικασίες πρώτα και μετά στα δεδομένα. Οι πιο σύγχρονες καθορίζουν τη δομή των διαδικασιών που επιδρούν πάνω στα δεδομένα με βάση τη δομή των δεδομένων αυτών και για το λόγο αυτόχαρακτηρίζονται ως «βασισμένες στα δεδομένα» και συγγενεύουν εν μέρει με τις «προσανατολισμένες στα αντικείμενα» τεχνοτροπίες.

Η αντικειμενοστρεφής προσέγγιση (object–oriented) ακολουθεί ένα διαφορετικό δρόμο: αντί το σύστημα να θεωρείται ως μια ιεραρχία διαδικασιών, ανεξάρτητων από τα δεδομένα, θεωρείται ως μια συλλογή οντοτήτων, καθεμία εκ των οποίων περικλείει και διαδικασίες και δεδομένα. Η προσέγγιση βασίζεται στην ιδέα ότι στον πραγματικό κόσμο δεδομένα και διαδικασίες μπορούν να ιδωθούν ενιαία με βάση το πεδίο ευθύνης κάποιων οντοτήτων που ονομάζονται «αντικείμενα».

Κάθε αντικείμενο παρέχει στο περιβάλλον του ένα σύνολο υπηρεσιών της ευθύνης του. Η συνεργασία του συνόλου των αντικειμένων του πεδίου μιας εφαρμογής λογισμικού παράγει το επιθυμητό αποτέλεσμα. Μερικές από τις πιο γνωστές προσεγγίσεις που ανήκουν στην κατηγορία αυτή προτάθηκαν από τους Meyer (1988), Booch (1994), Jacobson (1993), Rumbaugh (1992). Για μεγάλο χρονικό διάστημα επικρατούσε μια σύγχυση σε επίπεδο ορολογίας και συμβολισμών στην οικογένεια της αντικειμενοστραφούς ανάλυσης και σχεδίασης. Τα τελευταία χρόνια η σύγχυση αυτή φαίνεται να περιορίζεται με την εμφάνιση «συγχωνευμένων» μεθοδολογιών και ενοποιημένων συμβολισμών.

Στον ορισμό που δόθηκε παραπάνω, το σχέδιο λογισμικού φέρεται ως «η περιγραφή των μονάδων που αποτελούν το λογισμικό, των συσχετίσεων μεταξύ τους, της διάταξής τους, καθώς και της εσωτερικής τους λεπτομέρειας».

Προκειμένου να γίνει δυνατή η περιγραφή αυτή, πρέπει να αντιμετωπίσουμε το πρόβλημα σε τέσσερα επίπεδα ως ακολούθως:

<u>Αρχιτεκτονική σχεδίαση</u>: Αφορά τον καθορισμό τού ποιες είναι οι μονάδες που συγκροτούν το σύστημα λογισμικού και πώς αυτές ανατίθενται για εκτέλεση (διατάσσονται) στις υπολογιστικές μονάδες που είναι διαθέσιμες.

<u>Σχεδίαση διαπροσωπειών (interfaces)</u>: Αφορά τον καθορισμό της επικοινωνίας των μονάδων μεταξύ τους, με άλλα συστήματα λογισμικού, με άλλες συσκευές, καθώς και με τον άνθρωπο. Λέγοντας «καθορισμός επικοινωνίας» εννοούμε την περιγραφή του ποια μονάδα επικοινωνεί με ποια, με ποιες παραμέτρους, καθώς και με όποιο άλλο στοιχείο απαιτείται για να περιγραφεί επαρκώς, κατά περίπτωση, η επικοινωνία.

<u>Λεπτομερής σχεδίαση μονάδων</u>: Αφορά τον καθορισμό της εσωτερικής δομής κάθε μονάδας λογισμικού προκειμένου αυτή να ικανοποιεί, αφενός, τις λειτουργικές απαιτήσεις και, αφετέρου, να συνεργάζεται με τις άλλες μονάδες όπως έχει καθοριστεί κατά την αρχιτεκτονική και τη σχεδίαση δεδομένων.

<u>Σχεδίαση δεδομένων</u>: Πρόκειται για τη λεπτομερή σχεδίαση των δεδομένων, η τήρηση των οποίων αποτελεί απαίτηση από το λογισμικό, η οποία έχει τεθεί στη φάση προδιαγραφής των απαιτήσεων.

Είσοδο στην εργασία της σχεδίασης αποτελούν τα προϊόντα που έχουν παραχθεί κατά τη φάση της προδιαγραφής των απαιτήσεων από το λογισμικό. Σε όλα της τα επίπεδα, η σχεδίαση λογισμικού είναι μια εργασία που προχωρά από τη μικρότερη στη μεγαλύτερη λεπτομέρεια.

<u>Εξέλιξη της σχεδίασης λογισμικού</u>

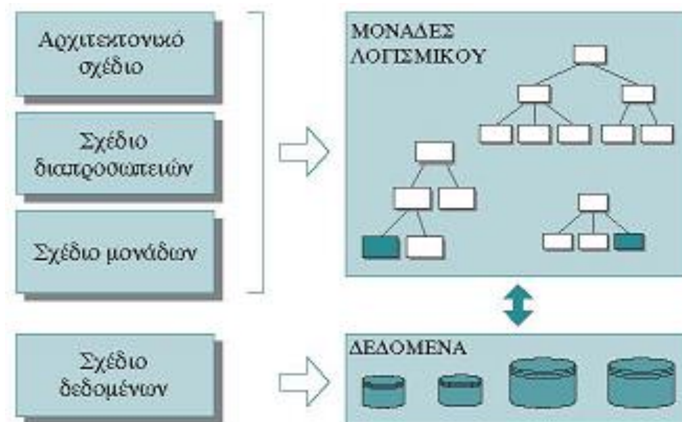
Στο παραπάνω σχήμα φαίνεται μια συσχέτιση μεταξύ των προϊόντων σχεδίασης, που προκύπτουν από την εκτέλεση των εργασιών στα τέσσερα επίπεδα που αναφέρθηκαν, με τα προϊόντα της φάσης προσδιορισμού των προδιαγραφών.

Όπως φαίνεται και στο σχήμα που ακολουθεί, στη δομημένη ανάλυση και σχεδίαση μπορούμε να διακρίνουμε τη σχεδίαση των δεδομένων από τη σχεδίαση που αφορά τις μονάδες προγράμματος. Η σχεδίαση των δεδομένων φαίνεται να σχετίζεται σχεδόν αποκλειστικά με το διάγραμμα οντοτήτων –συσχετίσεων και με το λεξικό δεδομένων, ενώ η σχεδίαση μονάδων (αρχιτεκτονική, επικοινωνία, δομή) σχετίζεται με το σύνολο των υπόλοιπων αποτελεσμάτων των προδιαγραφών. Αποφεύγουμε τον ορισμό της έννοιας «μονάδα λογισμικού», διότι αυτός εξαρτάται από τη γλώσσα προγραμματισμού που χρησιμοποιείται. Οι συναρτήσεις (functions), οι υπορουτίνες (subroutines), οι μονάδες (units), οι διαδικασίες (procedures) είναι οι συνηθέστεροι όροι που χρησιμοποιούνται από τις γλώσσες προγραμματισμού για να περιγράψουν μονάδες λογισμικού.

Από τα προϊόντα προδιαγραφών των απαιτήσεων στα προϊόντα σχεδίασης λογισμικού

Στο σχήμα που ακολουθεί πιο κάτω φαίνεται η ανεξαρτησία των δεδομένων από τις μονάδες λογισμικού, η οποία, όπως τονίστηκε, είναι ουσιώδες χαρακτηριστικό της δομημένης ανάλυσης και σχεδίασης. Από το αρχιτεκτονικό σχέδιο, το σχέδιο διαπροσωπειών και το λεπτομερές σχέδιο μονάδων, τελικά θα

κατασκευαστεί μια δομή ενεργών συστατικών (μονάδων) λογισμικού, τα οποία θα επιδρούν πάνω σε κάποια ανεξάρτητα δεδομένα.



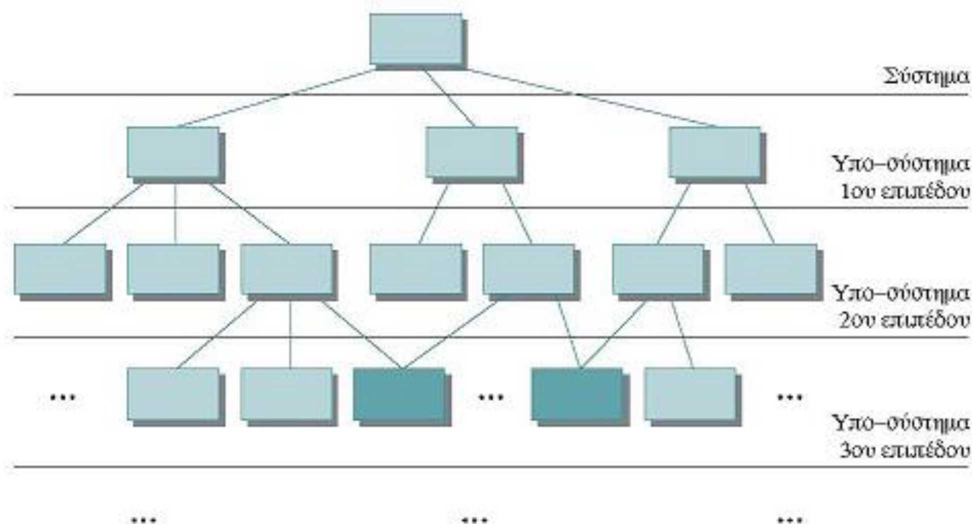
Δεδομένα και μονάδες λογισμικού στη δομημένη ανάλυση και σχεδίαση

Όπως αναφέρθηκε, κατά την αρχιτεκτονική σχεδίαση προσδιορίζεται το ποιες είναι οι μονάδες που αποτελούν το λογισμικό και πώς σχετίζονται μεταξύ τους. Ο προσδιορισμός αυτός αναφέρεται και ως σχεδίαση συστήματος. Στη δομημένη σχεδίαση η εργασία αυτή γίνεται λαμβάνοντας ως είσοδο τα διαγράμματα ροής δεδομένων. Σε αυτά εφαρμόζεται ένα σύνολο κανόνων και καθορίζονται οι μονάδες λογισμικού που είναι αναγκαίες για την υλοποίηση των μετασχηματισμών και των ροών δεδομένων.

Η αρχιτεκτονική δομή του λογισμικού μπορεί να ιδωθεί σε πολλά επίπεδα λεπτομέρειας. Στο σχήμα παρακάτω φαίνεται μια άποψη όπου το λογισμικό θεωρείται στο πρώτο επίπεδο ως ένα σύνθετο σύστημα, το οποίο αποτελείται από υποσυστήματα (δεύτερο επίπεδο), καθένα εκ των οποίων αποτελείται από υποσυστήματα 2ας τάξης (τρίτο επίπεδο) κ.ο.κ., μέχρις ότου φτάσουμε στις ατομικές μονάδες λογισμικού (συναρτήσεις, υπορουτίνες, κτλ.).

Η αρχιτεκτονική λογισμικού καθορίζει ποιες θα είναι αυτές και στη συνέχεια το λεπτομερές σχέδιο μονάδων θα καθορίσει πώς ακριβώς θα υλοποιηθούν. Σε όσο χαμηλότερο επίπεδο βρισκόμαστε, τόσο πιθανότερο είναι να αναγνωρίζονται «κοινοί παράγοντες», δηλαδή μονάδες λογισμικού που ανήκουν σε περισσότερα του ενός υποσυστήματα του αμέσως παραπάνω επιπέδου. Το φαινόμενο εμφανίζεται κυρίως μετά από την εκτέλεση της συναρτησιακής

αποσύνθεσης (functional decomposition), στην οποία θα αναφερθούμε κατά την περιγραφή της λεπτομερούς σχεδίασης μονάδων.



<u>Επίπεδα αρχιτεκτονικού σχεδίου λογισμικού</u>

Οι μονάδες που αποτελούν το λογισμικό μπορεί να ανατίθενται προς εκτέλεση σε πολλούς διατιθέμενους υπολογιστικούς πόρους (υπολογιστικά συστήματα, μονάδες επεξεργασίας). Στην απλούστερη περίπτωση, το λογισμικό είναι κατασκευασμένο ώστε να τρέχει εξ ολοκλήρου σε ένα μόνο υπολογιστικό σύστημα. Στη γενικότερη περίπτωση, τα υποσυστήματα του λογισμικού μπορούν να ανατίθενται σε διαφορετικούς υπολογιστικούς πόρους.

Καμία μονάδα λογισμικού δεν είναι ανεξάρτητη. Είτε χρησιμοποιεί άλλες μονάδες προκειμένου να επιτελέσει τη λειτουργία για την οποία προορίζεται είτε χρησιμοποιείται από άλλες μονάδες για τον ίδιο σκοπό. Η διατύπωση ότι μια μονάδα λογισμικού χρησιμοποιεί άλλες μονάδες δηλώνει την κλήση προγραμμάτων, υπορουτινών, συναρτήσεων και άλλων δομικών στοιχείων λογισμικού. Η κλήση μιας μονάδας Β από μια μονάδα Α σημαίνει:

-ενεργοποίηση της μονάδας Β, δηλαδή η ροή του προγράμματος μεταφέρεται στη μονάδα Β και εκτελούνται οι εντολές που περιέχονται σε αυτή, καθώς και

-επικοινωνία της μονάδας Α με τη Β μέσω παραμέτρων που περνά η Α στη Β και ενδεχομένως και αντίστροφα.

Κατά τη σχεδίαση διαπροσωπειών (interface design) καθορίζονται οι λεπτομέρειες του περάσματος των παραμέτρων αυτών σε όλα τα επίπεδα

επικοινωνίας. Οι μονάδες λογισμικού δεν επικοινωνούν μόνο μεταξύ τους, αλλά και με εξωτερικά συστήματα ή συσκευές, καθώς και με τον άνθρωπο. Συγκεντρώνοντας το σύνολο των απαιτήσεων από τη σχεδίαση διαπροσωπειών, καταλήγουμε ότι κατά την εργασία αυτή θα πρέπει να καθοριστούν τα ακόλουθα:

- Ο αριθμός και ο τύπος των παραμέτρων κατά την κλήση μονάδων λογισμικού.
- Το είδος και οι λεπτομέρειες της επικοινωνίας με άλλα συστήματα λογισμικού.
- Το είδος και οι λεπτομέρειες της επικοινωνίας με εξωτερικές συσκευές.
- Η επικοινωνία του λογισμικού με τον άνθρωπο.

Οι παράμετροι κατά την επικοινωνία με άλλες μονάδες λογισμικού (στην ίδια εφαρμογή) καθορίζονται κατά την αρχιτεκτονική και τη λεπτομερή σχεδίαση και περιγράφονται στα διαγράμματα δομής προγράμματος και το λεπτομερές σχέδιο μονάδων, στα οποία θα αναφερθούμε σε επόμενες παραγράφους. Οι εξωτερικές διαπροσωπείες περιγράφονται αυτοτελώς με τον προσφορότερο κατά περίπτωση τρόπο, ανάλογα με τα εξωτερικά συστήματα λογισμικού ή τις εξωτερικές συσκευές. Η σχεδίαση της διαπροσωπείας με τον άνθρωπο είναι ένα ιδιαίτερα πολύπλοκο πρόβλημα, στο οποίο εκτός από μηχανικούς λογισμικού μπορεί να έχουν λόγο και άλλες ειδικότητες, όπως κοινωνιολόγοι και ψυχολόγοι, καθώς και κοινοί χρήστες.

Το περισσότερο λεπτομερές επίπεδο της σχεδίασης είναι η σχεδίαση μονάδων. Πρόκειται για το σημείο όπου πρέπει να καθοριστούν όλες οι κατασκευαστικές λεπτομέρειες που απαιτούνται για τη δημιουργία του πηγαίου κώδικα για όλες τις μονάδες λογισμικού. Η γνώση της ύπαρξης μιας μονάδας και των χαρακτηριστικών επικοινωνίας αυτής με άλλες μονάδες δεν επαρκεί, στη γενική περίπτωση, για την κατασκευή του πηγαίου κώδικα της ίδιας της μονάδας. Σε τετριμμένες περιπτώσεις, η λεπτομερής περιγραφή της εσωτερικής δομής δεν είναι αναγκαία για την κατασκευή της μονάδας. Στις περισσότερες όμως των περιπτώσεων αυτό δεν ισχύει και η λεπτομερής σχεδίαση μονάδων είναι αναγκαία. Το υλικό που χρησιμοποιείται ως είσοδος στη φάση της λεπτομερούς σχεδίασης μονάδων είναι το «αρχιτεκτονικό σχέδιο», το «σχέδιο διαπροσωπειών», το «έγγραφο προδιαγραφών των απαιτήσεων από το λογισμικό», καθώς και το «διάγραμμα μετάβασης καταστάσεων». Τα δύο πρώτα έχουν κατασκευαστεί κατά τη σχεδίαση, ενώ τα δύο τελευταία κατασκευάστηκαν κατά την προδιαγραφή των

απαιτήσεων και περιέχουν περιγραφή της λειτουργικής συμπεριφοράς του λογισμικού με τη μορφή απαιτήσεων. Κατά τη λεπτομερή σχεδίαση, για κάθε μονάδα θα δοθεί ένα περίγραμμα –ομοίωμα της δομής της, χρήσιμο για την κατασκευή της. Αποτέλεσμα της λεπτομερούς σχεδίασης είναι το λεπτομερές σχέδιο μονάδων, το οποίο στη συνέχεια θα δοθεί στους προγραμματιστές που θα συγγράψουν τον πηγαίο κώδικα.

Η σχεδίαση δεδομένων θεωρείται ως μια από τις σημαντικότερες διαδικασίες στην ανάπτυξη λογισμικού. Η σύλληψη της δομής και των συσχετίσεων των δεδομένων μπορεί να καθορίζει πολλά από τα χαρακτηριστικά των λειτουργικών μονάδων λογισμικού. Στη δομημένη ανάλυση και σχεδίαση, ο προσδιορισμός των απαιτήσεων και η λεπτομερής σχεδίαση των δεδομένων, από τη μία, και η σχεδίαση των μονάδων λογισμικού, από την άλλη, είναι δύο διαφορετικές διακριτές εργασίες. Προκειμένου να μπορεί να κατασκευαστεί με πληρότητα το λεπτομερές σχέδιο μονάδων λογισμικού, είναι απαραίτητο να διατίθεται το λεπτομερές σχέδιο δεδομένων. Αυτό ισχύει για τις περιπτώσεις λειτουργικών μονάδων λογισμικού που επιδρούν, με οποιονδήποτε τρόπο , πάνω σε δεδομένα. Έχοντας, λοιπόν, υπόψη το λεξικό δεδομένων και το διάγραμμα οντοτήτων συσχετίσεων, ο σχεδιαστής λογισμικού:

- Πραγματοποιεί βελτιστοποιήσεις στο μοντέλο οντοτήτων συσχετίσεων, με σκοπό τη βελτιστοποίηση των επιδόσεων, την πληροφοριακή πληρότητα και την εξάλειψη πλεονάζουσας (redundant) πληροφορίας.
- Καθορίζει τη δομή κάθε πίνακα (πεδία και τύποι αυτών) στο φυσικό επίπεδο.
- Καθορίζει τις δυνατές απόψεις των δεδομένων στο λογικό επίπεδο (views).

Με το αντικείμενο της λεπτομερούς σχεδίασης δεδομένων ασχολείται η γνωστική περιοχή των βάσεων δεδομένων. Το αποτέλεσμα της σχεδίασης δεδομένων είναι η λεπτομερής περιγραφή των οντοτήτων και των συσχετίσεων αυτών, σε επίπεδο που να είναι εφικτή η υλοποίηση της Βάσης Δεδομένων.

Σε αναλογία με το έγγραφο προδιαγραφών των απαιτήσεων από το λογισμικό, είναι χρήσιμο το σχέδιο του λογισμικού να αποτυπώνεται με χρήση ενός δομημένου εγγράφου. Το έγγραφο αυτό αποτελεί ένα ενιαίο δομικό κέλυφος στο οποίο ενσωματώνονται όλα τα επιμέρους προϊόντα της σχεδίασης με τη

μορφή διαγραμμάτων, αλλά και με τη μορφή κειμένων ή άλλων τρόπων περιγραφής σχεδίου, τους οποίους θα αναφέρουμε στη συνέχεια.

Πριν προχωρήσουμε στην αναφορά της προσέγγισης της «δομημένης σχεδίασης», είναι χρήσιμο να αναφερθούμε στις υπάρχουσες επιλογές διατάξεων λογισμικού. Θα παρουσιαστούν τέσσερα μοντέλα διατάξεων: το μονολιθικό μοντέλο, το μοντέλο πελάτη –εξυπηρετητή (client– server), το τριμερές μοντέλο (3–tier), καθώς και ένα γενικευμένο μοντέλο πολλαπλής διάταξης (multi–tier). Προκειμένου να αποφύγουμε σύγχυση και διφορούμενα, θα εισαγάγουμε την έννοια της «διάταξης λογισμικού».

Διάταξη λογισμικού (software deployment) είναι η κατάτμηση μιας εφαρμογής σε ανεξάρτητα λειτουργικά τμήματα και η ανάθεση αυτών σε διατιθέμενους υπολογιστικούς πόρους (συστήματα, επεξεργαστές).

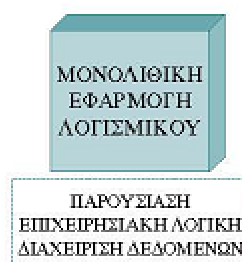
Με βάση τον παραπάνω ορισμό, με τον όρο «διάταξη» αναφερόμαστε στη γενική αρχιτεκτονική του λογισμικού. Σε αρκετές περιπτώσεις χρησιμοποιείται σκέτος ο όρος «αρχιτεκτονική», ιδιαίτερα εκεί όπου το λογισμικό θεωρείται από εξωτερική σκοπιά και όχι από αυτή του κατασκευαστή. Ακολούθως θα αναφερόμαστε στη γενική αρχιτεκτονική με τον όρο «διάταξη» και στην εσωτερική αρχιτεκτονική, στην οποία ήδη αναφερθήκαμε, με τον όρο «αρχιτεκτονική».

Ο ορισμός των διατάξεων που θα ακολουθήσει θα βασιστεί στην κατηγοριοποίηση των εργασιών που μπορεί να κάνει μια εφαρμογή λογισμικού, όπως φαίνεται στο σχήμα παρακάτω. Η κατηγοριοποίηση αυτή έχει γίνει γενικά αποδεκτή από κατασκευαστές και ερευνητές. Διακρίνονται τρία είδη εργασιών: οι εργασίες παρουσίασης, οι εργασίες διαχείρισης δεδομένων και οι εργασίες επιχειρησιακής λογικής.

Μια διάκριση των εργασιών που κάνει μια εφαρμογή λογισμικού

Ως εργασίες παρουσίασης ορίζονται όλες οι εργασίες που σχετίζονται με την επικοινωνία του συστήματος με το χρήστη και με εξωτερικές συσκευές και συστήματα, δηλαδή εργασίες που υλοποιούν τις διαπροσωπείες του λογισμικού. Οι εργασίες διαχείρισης δεδομένων είναι εκείνες που ασχολούνται με την αποθήκευση και την ανάκτηση των δεδομένων. Τέλος, οι εργασίες επιχειρησιακής λογικής (business logic) είναι όλες οι υπόλοιπες εργασίες, δηλαδή εκείνες που υλοποιούν τις ιδιαίτερες λειτουργικές απαιτήσεις κάθε εφαρμογής λογισμικού. Η διάκριση αυτή απαιτεί μια αυστηρότητα στον ορισμό των μονάδων λογισμικού. Μια μονάδα η οποία εκτελεί έναν υπολογισμό (εργασία επιχειρησιακής λογικής) δε θα πρέπει να τυπώνει η ίδια το αποτέλεσμά του στην οθόνη (εργασία παρουσίασης).

Η απλούστερη διάταξη λογισμικού είναι η μονολιθική (σχήμα που ακολουθεί). Σε αυτήν ολόκληρη η εφαρμογή τρέχει σε ένα υπολογιστικό σύστημα. Η διάταξη αυτή είναι κατάλληλη για μικρές εφαρμογές με σχετικά περιορισμένες απαιτήσεις και λειτουργίες και, όπως είναι αναμενόμενο, υπήρξε η πρώτη διάταξη που για μεγάλο χρονικό διάστημα χρησιμοποιήθηκε.

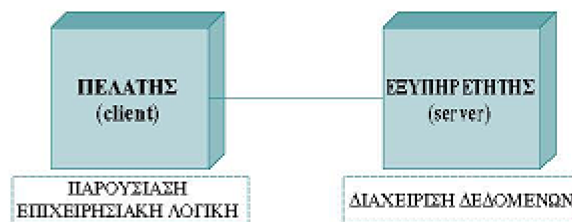


Η μονολιθική διάταξη λογισμικού

Ο συμβολισμός που χρησιμοποιείται στο σχήμα παραπάνω αναφέρεται ως διάγραμμα διάταξης λογισμικού (deployment diagram) και περιγράφει την ανάθεση τμημάτων της εφαρμογής σε υπολογιστικούς πόρους. Τα τμήματα αυτά συμβολίζονται με ένα τρισδιάστατο παραλληλεπίπεδο σκιασμένο όπως στο σχήμα, στην μπροστινή πλευρά του οποίου αναγράφεται η ονομασία κάθε τμήματος.

Η αύξηση των απαιτήσεων από το λογισμικό, της πολυπλοκότητας αλλά και του όγκου των δεδομένων που διαχειρίζεται μια εφαρμογή, κατέστησαν τη μονολιθική διάταξη ανεπαρκή για την ικανοποίηση πολλών απαιτήσεων. Παράλληλα, η ανάπτυξη των συστημάτων διαχείρισης σχεσιακών βάσεων

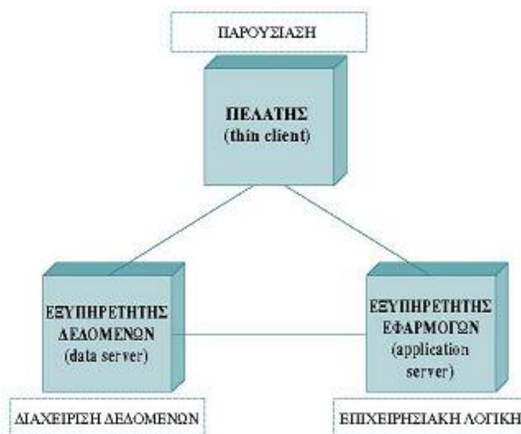
δεδομένων, αλλά και των δικτύων, επέτρεψαν την εξέλιξη της μονολιθικής διάταξης σε αυτή του πελάτη–εξυπηρετητή (client–server). Στο σχήμα που ακολουθεί φαίνεται το διάγραμμα διάταξης πελάτη–εξυπηρετητή. Οι εργασίες που σχετίζονται με τη διαχείριση δεδομένων ανατίθενται σε ένα ξεχωριστό τμήμα της εφαρμογής, το οποίο τρέχει συνήθως σε ένα αφιερωμένο στη διαχείριση δεδομένων υπολογιστικό σύστημα. Οι εργασίες παρουσίασης και επιχειρησιακής λογικής τρέχουν σε ένα άλλο τμήμα, το οποίο επικοινωνεί μέσω δικτύου με τον εξυπηρετητή ζητώντας του την παροχή σχετικών με δεδομένα υπηρεσιών. Η ιδέα, πρωτοποριακή για την εποχή της, έλυσε το πρόβλημα των ολοένα και μεγαλύτερων υπολογιστικών απαιτήσεων από τα γιγαντωμένα μονολιθικά συστήματα, οι οποίες μεγάλωναν μαζί με τον αριθμό των χρηστών αλλά και την πολυπλοκότητα των εργασιών που εκτελούσαν. Με τον καιρό, διάφορα προβλήματα της διάταξης αυτής αναδείχτηκαν. Το σημαντικότερο εντοπίζεται στην ανάγκη συντήρησης όλων των συστημάτων πελάτη (τα οποία μπορούσαν να είναι πολυάριθμα), καθώς συνέβαιναν οποιεσδήποτε μεταβολές στο επίπεδο της επιχειρησιακής λογικής. Επίσης, όσο μεγάλωνε η πολυπλοκότητα των λειτουργιών, τόσο περισσότερο τα συστήματα όπου έτρεχαν τα συστήματα πελάτη αποδεικνύονταν ανεπαρκή από πλευράς υπολογιστικής ισχύος.



Η διάταξη πελάτη–εξυπηρετητή

Εμφανίστηκαν, λοιπόν, νέες εκδοχές της διάταξης πελάτη–εξυπηρετητή που διαχωρίζουν ακόμη περισσότερο τις «αρμοδιότητες» του λογισμικού. Ο πελάτης ελαφρύνεται και μένει μόνο με την ευθύνη της παρουσίασης, ενώ εμφανίζεται και ένας δεύτερος τύπος εξυπηρετητή, ο εξυπηρετητής εφαρμογών, ο οποίος κάνει τις εργασίες του επιπέδου της επιχειρησιακής λογικής. Στην περίπτωση αυτή, ο πελάτης χαρακτηρίζεται ως ελαφρύς (thin client), ακριβώς διότι κάνει λιγότερα πράγματα απ' ό,τι στην αρχική εκδοχή της διάταξης πελάτη–εξυπηρετητή. Τα συστήματα των εξυπηρετητών δεδομένων και εφαρμογών είναι συνήθως μεγάλα

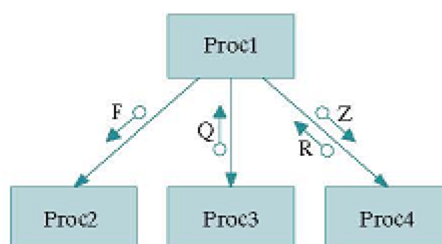
κεντρικά υπολογιστικά συστήματα. Τα προβλήματα της διάταξης πελάτη–εξυπηρετητή περιορίζονται, διότι οι απαιτήσεις συντήρησης των πελατών υφίστανται μόνο όταν συμβαίνουν μεταβολές στο επίπεδο της παρουσίασης. Πάντως, δεν παύουν να υπάρχουν.



Η τριμερής διάταξη λογισμικού

Το επόμενο αναμενόμενο βήμα είναι η αφαίρεση και της αρμοδιότητας της παρουσίασης από τον πελάτη και η ανάθεσή της σε έναν εξυπηρετητή παρουσίασης. Η ιδέα του εξυπηρετητή παρουσίασης γεννήθηκε με την εμφάνιση του παγκόσμιου ιστού (world wide web) και του Internet και έγινε δυνατή με την ανάπτυξη τεχνολογιών που επιτρέπουν την αλληλεπίδραση μεταξύ του web server και του συνδεδεμένου σε αυτόν πελάτη (browser). Τέτοιες τεχνολογίες είναι τα scripts, η Java, καθώς και το μοντέλο DCOM, και σήμερα είναι ήδη αρκετά διαδεδομένες. Η διάταξη ονομάστηκε πολυμερής (multi–tier) ή βασισμένη στο web (web based) και εμφανίζεται στο σχήμα πιο κάτω. Τα τμήματα της εφαρμογής λογισμικού διατάσσονται με τρόπο ώστε όλες οι εργασίες των τριών κατηγοριών (παρουσίασης, διαχείρισης δεδομένων και επιχειρησιακής λογικής) να εκτελούνται σε έναν ή περισσότερους για κάθε κατηγορία εξυπηρετητές. Ο εξυπηρετητής παρουσίασης δεν είναι παρά ένας εξυπηρετητής υπηρεσιών web (web server). Ο πελάτης δεν απαιτείται να διαθέτει κανένα τμήμα της εφαρμογής, παρά μόνο τη δυνατότητα επικοινωνίας με τον web server, δηλαδή μια δικτυακή σύνδεση και ένα πρόγραμμα πλοήγησης στο web (browser). Για το λόγο αυτό, ο πελάτης αναφέρεται και ως «web client». Τα προβλήματα ανάγκης συντήρησης των συστημάτων των πελατών εκμηδενίζονται, γεννιούνται όμως άλλα, αυτά της ταχύτητας και της ασφάλειας των δικτυακών συνδέσεων.

Η μέθοδος της δομημένης σχεδίασης βασίζεται στα διαγράμματα ροής δεδομένων, από τα οποία, με κάποιο συστηματικό αλλά όχι αυτόματο τρόπο, παράγεται το αρχιτεκτονικό σχέδιο του λογισμικού. Το σχέδιο αυτό αποτυπώνεται με τη βοήθεια ενός διαγράμματος που ονομάζεται «διάγραμμα δομής προγράμματος», όπως αυτό που φαίνεται στο σχήμα. Στο διάγραμμα δομής προγράμματος οι μονάδες λογισμικού παριστάνονται με ένα παραλληλόγραμμο. Η κλήση της μονάδας Β από τη μονάδα Α υποδηλώνεται με ένα βέλος που συνδέει την Α με τη Β. Το πέρασμα παραμέτρων προς αμφότερες υποδηλώνεται με ένα βέλος με έναν κύκλο στο άκρο της αρχής του, σημειωμένο σε διεύθυνση παράλληλη με τη διεύθυνση του βέλους που περιγράφει την κλήση, το οποίο φέρει το όνομα της παραμέτρου.



Σχ.10 Συμβολισμοί διαγραμμάτων δομής προγράμματος

Η διαδικασία εκτελείται για καθένα από τα διαγράμματα ροής δεδομένων του συστήματος. Στα διαγράμματα αυτά εντοπίζονται δύο τύποι χαρακτηριστικών περιοχών, οι κεντρικοί μετασχηματισμοί και τα κέντρα δοσοληψιών. Όταν εντοπιστεί μια τέτοια περιοχή, δημιουργείται ή συμπληρώνεται ένα επίπεδο του διαγράμματος δομής προγράμματος και η διαδικασία επαναλαμβάνεται μέχρις ότου να εξαντληθεί το διάγραμμα ροής δεδομένων.

Δεν υπάρχει αυτόματος τρόπος για τον εντοπισμό των κεντρικών μετασχηματισμών. Η εμπειρία, ο αυτοσχεδιασμός και η διαίσθηση του μηχανικού λογισμικού έχουν και εδώ τον πρώτο λόγο. Επίσης, η επιλογή ενός κεντρικού μετασχηματισμού δεν είναι μοναδική. Αυτό σημαίνει ότι στο ίδιο διάγραμμα ροής δεδομένων δύο ή περισσότερες περιοχές μπορούν να χαρακτηριστούν ως κεντρικός μετασχηματισμός, χωρίς να είναι απαραίτητα λάθος ο ένας από τους δύο χαρακτηρισμούς. Η σύλληψη και η λεπτομέρεια του διαγράμματος ροής δεδομένων παίζουν, όπως είναι φανερό, καθοριστικό ρόλο στο σημείο αυτό. Αξίζει να σημειωθεί ότι, ενώ ένας κεντρικός μετασχηματισμός μπορεί να περιλαμβάνει

περισσότερους του ενός απλούς μετασχηματισμούς του διαγράμματος ροής δεδομένων, ένα κέντρο δοσοληψιών περιλαμβάνει μόνο έναν. Χαρακτηριστική περίπτωση κέντρου δοσοληψιών είναι ένας μετασχηματισμός ελέγχου ροής προγράμματος, όπου, ανάλογα με την επιλογή του χρήστη, η ροή μεταφέρεται σε διαφορετικούς μετασχηματισμούς, όπως η περίπτωση ενός μενού.

Η μετάβαση από το διάγραμμα ροής δεδομένων στο διάγραμμα δομής γίνεται με διαδοχική επανάληψη κάποιων βημάτων, μέχρι να έχουν προσδιοριστεί μονάδες για όλους τους μετασχηματισμούς που περιέχονται στα διαγράμματα ροής δεδομένων της εφαρμογής. Τα βήματα αυτά είναι:

1. Εντοπισμός κεντρικού μετασχηματισμού. Για κάθε τμήμα του διαγράμματος ροής δεδομένων εντοπίζεται ο κεντρικός μετασχηματισμός και διακρίνονται οι μετασχηματισμοί εισόδου και εξόδου σε αυτόν.
2. Απεικόνιση του κεντρικού μετασχηματισμού σε διάγραμμα δομής. Δημιουργείται ένα επίπεδο του διαγράμματος δομής που αντιστοιχεί στον κεντρικό μετασχηματισμό.
3. Παραγοντοποίηση (factoring). Για το αριστερό και το δεξί τμήμα του κεντρικού μετασχηματισμού (είσοδοι και έξοδοι) δημιουργούνται τα διαγράμματα δομής, που αντιστοιχούν στους μετασχηματισμούς που περιέχονται σε αυτά. Κάθε τέτοιος μετασχηματισμός απεικονίζεται σε μια μονάδα ελέγχου και σε δύο άλλες μονάδες. Από αυτές, η πρώτη λαμβάνει τα δεδομένα εισόδου, ενώ η δεύτερη πραγματοποιεί τη μετατροπή. Η μονάδα ελέγχου διαθέτει τα δεδομένα στο παραπάνω επίπεδο. Κατά την παραγοντοποίηση ενδεχομένως να αναγνωριστούν και άλλοι κεντρικοί μετασχηματισμοί ή κέντρα δοσοληψιών, τα οποία αντιμετωπίζονται όπως περιγράφηκε. Η διαδικασία επαναλαμβάνεται μέχρις ότου να φτάσουμε στις πηγές και τους αποδέκτες των δεδομένων, δηλαδή το χρήστη, εξωτερικά συστήματα, συσκευές ή αρχεία.
4. Συνένωση. Η τελευταία εργασία που πρέπει να γίνει είναι αυτή της συνένωσης. Για τις περιπτώσεις όπου τα δεδομένα δε λαμβάνονται από εξωτερική πηγή ή δεν καταλήγουν σε εξωτερικό αποδέκτη, η μονάδα που τα εισάγει στον κεντρικό μετασχηματισμό αντικαθίσταται από τη μονάδα ελέγχου του μετασχηματισμού που τα παρέχει.

Κατά τη διαδικασία αυτή ενδεχομένως να διαπιστωθεί ότι είναι χρήσιμο να πραγματοποιηθούν τροποποιήσεις στα διαγράμματα ροής δεδομένων, γεγονός

που συνιστά οπισθοδρόμηση στη διαδικασία ανάπτυξης λογισμικού, η οποία όμως είναι χρήσιμο να πραγματοποιηθεί.

Έχοντας διαθέσιμο το αρχιτεκτονικό σχέδιο, είναι δυνατή η λεπτομερής σχεδίαση των μονάδων λογισμικού που περιλαμβάνονται σε αυτό. Κατά τη λεπτομερή σχεδίαση θα προσδιοριστεί η εσωτερική δομή κάθε μονάδας, δηλαδή θα δοθεί μια περιγραφή του πηγαίου κώδικα. Μέχρι το σημείο αυτό είναι γνωστή μόνο η ονομασία κάθε μονάδας και οι παράμετροι εισόδου και εξόδου αυτής. Στοιχεία που σχετίζονται με την περιγραφή της συμπεριφοράς της είναι διαθέσιμα από τη φάση της προδιαγραφής των απαιτήσεων. Με το υλικό αυτό ο σχεδιαστής λογισμικού καλείται να κατασκευάσει το λεπτομερές σχέδιο. Σε περίπτωση που το διάγραμμα δομής προγράμματος είναι επαρκώς λεπτομερές, καθεμία μονάδα του διαγράμματος αντιστοιχεί σε μια μονάδα κώδικα. Αν όμως αυτό δεν ισχύει, είναι ανάγκη κάθε μονάδα του διαγράμματος δομής να απεικονιστεί, ενδεχομένως, σε περισσότερες από μία μονάδες λογισμικού. Η εργασία αυτή γίνεται αναλύοντας κάθε πρόβλημα σχεδίασης σε διαδοχικά βήματα. Σε κάθε βήμα δημιουργούμε ένα σύνολο από μικρότερα προβλήματα σχεδίασης, καθένα εκ των οποίων επιλύουμε και πάλι αναλύοντάς το κ.ο.κ., μέχρις ότου φτάσουμε σε απλές δομικές μονάδες λογισμικού, όπως οι «διαδικασίες» και οι «συναρτήσεις». Η αντιμετώπιση αυτή ονομάζεται εκλέπτυνση σε διαδοχικά βήματα (stepwise refinement) ή συναρτησιακή αποσύνθεση (functional decomposition).

Υπάρχουν αρκετοί τρόποι για να περιγράφονται τα αποτελέσματα της λεπτομερούς σχεδίασης. Ο επικρατέστερος είναι με χρήση μιας γλώσσας σχεδίασης προγράμματος (PDL: program description language). Μια γλώσσα σχεδίασης προγράμματος μοιάζει με γλώσσα προγραμματισμού, χωρίς να έχει την αυστηρότητα στη σύνταξη και τη γραμματική που χαρακτηρίζει τις γλώσσες προγραμματισμού. Σκοπός της είναι να παρέχει μια εικόνα του λογισμικού η οποία να μπορεί να υλοποιηθεί εύκολα σε κάποια γλώσσα προγραμματισμού. Οι γλώσσες σχεδίασης προγράμματος περιέχουν τις βασικές δομές ελέγχου που απαντώνται στις γλώσσες προγραμματισμού. Οι γλώσσες σχεδίασης προγράμματος αναφέρονται συχνά και ως «ψευδοκώδικας».

1.2 Οι ιδιαιτερότητες της ανάπτυξης λογισμικού

Το λογισμικό ως προϊόν έχει αρκετές ιδιαιτερότητες που κάνουν τη διαχείρισή της ανάπτυξής του περίπλοκη. Το λογισμικό, όπως γνωρίζετε πολύ καλά, σχεδιάζεται και αναπτύσσεται και δεν κατασκευάζεται με τρόπο αντίστοιχο της παραγωγής υλικών αγαθών. Βασικές αρχές της παραγωγής υλικών αγαθών δεν εφαρμόζονται στην ανάπτυξη λογισμικού. Τέτοιες αρχές είναι η επεξεργασία πρώτων υλών για τη δημιουργία τμημάτων του προϊόντος, η σύνθεση έτοιμων τμημάτων για τη δημιουργία του τελικού προϊόντος, η δημιουργία ενός πρωτοτύπου και η έμφαση στην πιστή αναπαραγωγή αντιγράφων με ελάχιστες αποκλίσεις από το πρωτότυπο. Αυτές οι αρχές δεν σχετίζονται άμεσα με την παραγωγή λογισμικού. Βέβαια ο αντικειμενοστραφής προγραμματισμός (object – oriented programming) και ακόμα περισσότερο ο προγραμματισμός που είναι βασισμένος σε ψηφίδες (component based programming) έχουν συντελέσει αρκετά στο να θεωρούμε ότι το λογισμικό αναπτύσσεται με σύνθεση τμημάτων, αλλά όχι ακόμα στους ρυθμούς και με την αποτελεσματικότητα παραγωγής υλικών αγαθών.

Επειδή η τεχνολογία των ηλεκτρονικών υπολογιστών -και κατά συνέπεια και του λογισμικού- εξελίσσεται με ταχύτατους ρυθμούς, για αρκετά έργα ανάπτυξης λογισμικού δεν έχουμε καθόλου ιστορικά δεδομένα, ή τα ιστορικά δεδομένα δεν είναι αξιοποιήσιμα. Ιστορικά δεδομένα αποκαλούμε δεδομένα από παρόμοια έργα που αναπτύχθηκαν κάτω από αντίστοιχες συνθήκες. Τέτοια δεδομένα, είτε δεν υπάρχουν γιατί έργα κάποιων κατηγοριών αναπτύσσονται (συνήθως) για πρώτη φορά, είτε δεν μπορούν να χρησιμοποιηθούν (έστω κι αν προέρχονται από αντίστοιχα έργα) γιατί οι μηχανισμοί, οι διαδικασίες και τα εργαλεία ανάπτυξης έχουν αλλάξει σημαντικά. Η διαδικασία ανάπτυξης λογισμικού δεν είναι τόσο διαφανής όσο είναι η διαδικασία κατασκευής σε αντίστοιχα κατασκευαστικά έργα ή έργα παραγωγής υλικών αγαθών, αλλά δεν είναι (ή δεν θα έπρεπε να είναι) αδιαφανής για τον υπεύθυνο του έργου. Για παράδειγμα, ένα κατασκευαστικό έργο (π.χ. την κατασκευή ενός σπιτιού) και πόσο εύκολα ακόμα και κάποιος που δεν έχει γνώση της διαδικασίας ανάπτυξης μπορεί να παρακολουθεί την εξέλιξη της κατασκευής και δείτε την αντιστοιχία του με ένα έργο ανάπτυξης λογισμικού. Στην πραγματικότητα ένας από τους στόχους της διαχείρισης της ανάπτυξης λογισμικού

είναι η διαφάνεια στην ανάπτυξη. Η διαφάνεια αυτή, όπως θα δούμε και στη διάρκεια της μελέτης σας σχετίζεται, μεταξύ άλλων, με την ωριμότητα της επιχείρησης. Σε έρευνα του Curtis[1] αναφέρεται ότι η πλειοψηφία των έμπειρων διαχειριστών έργων λογισμικού θεωρεί τους ανθρώπους ως το σημαντικότερο παράγοντα από τον οποίο εξαρτάται η επιτυχία του έργου. Σε αντίθεση με την παραγωγή υλικών αγαθών όπου τα εργαλεία και οι πρώτες ύλες είναι πολύ σημαντικές, στην ανάπτυξη λογισμικού, όπου τα εργαλεία και οι τεχνικές εξελίσσονται ραγδαία, η σημασία των ανθρώπων (με τις ιδιαιτερότητες που αυτό συνεπάγεται) είναι κυρίαρχη. Επίσης, στην ανάπτυξη λογισμικού ένας μεγάλος αριθμός έργων είναι προσαρμοσμένο λογισμικό (custom software), δηλαδή λογισμικό το οποίο αναπτύσσεται για συγκεκριμένο πελάτη ο οποίος εντάσσεται στη διαδικασία ανάπτυξης, γεγονός που εμπλέκει έναν ακόμα ανθρώπινο παράγοντα, αλλά γι’αυτό θα μιλήσουμε σε μια παρακάτω παράγραφο.

Αρχικά, γίνεται κατανοητό ότι το λογισμικό για να είναι εύχρηστο, θα πρέπει να «επικοινωνεί» με τον χρήστη. Σύμφωνα με το διεθνές πρότυπο ISO 9241-11, η ευχρηστία ορίζεται ως η ικανότητα ενός προϊόντος να επιτυγχάνει συγκεκριμένους στόχους αποτελεσματικά, αποδοτικά και παρέχοντας υποκειμενική ικανοποίηση στους χρήστες του, όταν χρησιμοποιείται σε συγκεκριμένο πλαίσιο χρήσης.

Ο παραπάνω ορισμός τείνει να γίνει ο κύριος ορισμός αναφοράς της ευχρηστίας συστημάτων. Πέρα από το γεγονός ότι αναγνωρίζεται ως ο καθιερωμένος ορισμός για την ευχρηστία σε έναν συνεχώς αυξανόμενο αριθμό άρθρων της βιβλιογραφίας του χώρου, είναι και ο καθιερωμένος ορισμός του Common Industry Format (CIF) για έλεγχο και αξιολόγηση της ευχρηστίας, το οποίο στη συνέχεια έγινε πρότυπο από τον οργανισμό προτυποποίησης ANSI (ANSI/NCITS 354-2001) και εξελίχθηκε σε πρότυπο από τον οργανισμό προτυποποίησης ISO (ISO/IEC 25062:2006), μόλις το 2006. Επιπλέον, το πρότυπο ISO 13407, το οποίο παρέχει οδηγίες για χρηστοκεντρικό σχεδιασμό κάνει χρήση του παραπάνω ορισμού από το ISO 9241-11.

Ο εξίσου γνωστός και, ενδεχομένως, το ίδιο ευρέως χρησιμοποιούμενος στη βιβλιογραφία ορισμός της ευχρηστίας που έχει δοθεί από τον Nielsen θεωρεί ότι η ευχρηστία ενός συστήματος αναλύεται σε πέντε παραμέτρους :

[1] *«Three Problems Overcome with Behavioral Models of the Software Development Process», Bill Curtis, 1988*

• ευκολία εκμάθησης (learnability). Το σύστημα πρέπει να είναι εύκολο στην εκμάθησή του, έτσι ώστε ο χρήστης να μπορεί να ξεκινήσει την εργασία του γρήγορα.

• υψηλή απόδοση εκτέλεσης έργου (efficiency). Το σύστημα πρέπει να είναι αποδοτικό, έτσι ώστε όταν ο χρήστης μάθει τη χρήση του, να μπορεί να επιτύχει ένα υψηλό επίπεδο παραγωγικότητας.

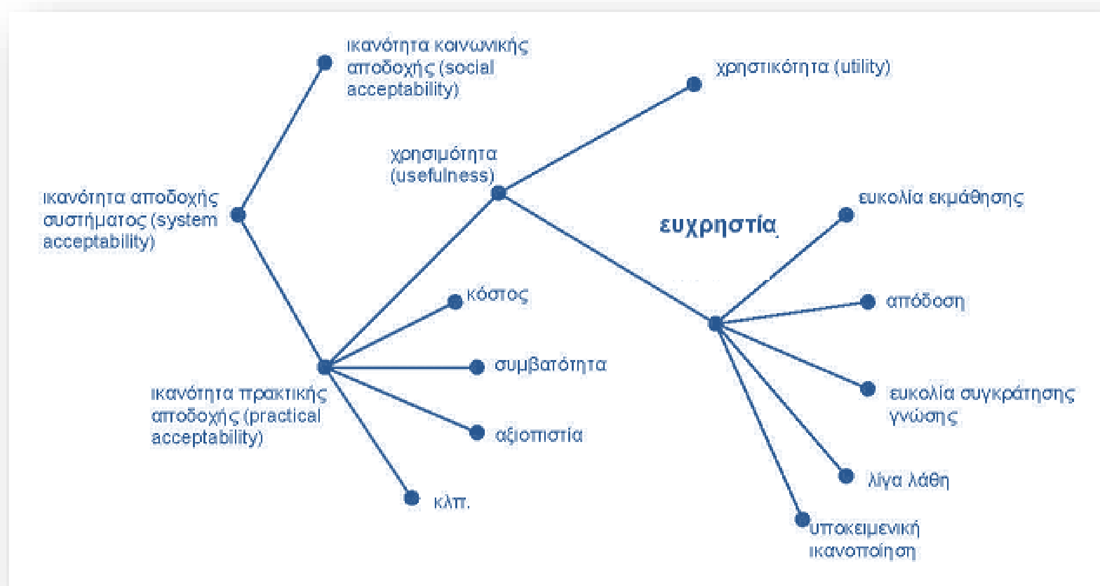
• ευκολία συγκράτησης της γνώσης του χρήστη (memorability). Ένας περιστασιακός χρήστης μπορεί να επιστρέψει στο σύστημα και να το χρησιμοποιήσει με επιτυχία χωρίς να πρέπει να το ξαναμάθει από την αρχή, ακόμα και αν έχει πολύ καιρό να το χρησιμοποιήσει.

• χαμηλή συχνότητα σφαλμάτων χρήστη. Οι χρήστες του συστήματος πρέπει να υποχρεώνονται σε όσο το δυνατό λιγότερα λάθη κατά τη χρήση του και να μπορούν να ανακάμπτουν εύκολα από αυτά όταν συμβαίνουν. Επιπλέον, το σύστημα δεν πρέπει να επιτρέπει καταστροφικά λάθη.

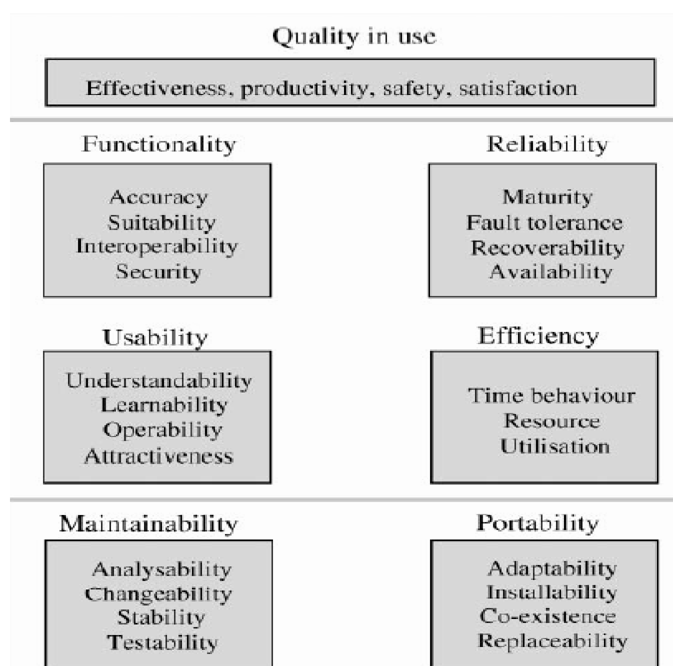
• υποκειμενική ικανοποίηση του χρήστη. Το σύστημα πρέπει να είναι ευχάριστο στη χρήση του, ώστε να προσφέρει υποκειμενική ικανοποίηση στους χρήστες.

Ο ορισμός κατά Nielsen, όντας πιο ανεπίσημος και εμπειρικός, είναι λίγο πιο αναλυτικός και δείχνει να έχει τη μορφή γενικών κατευθυντήριων οδηγιών, ενώ φαίνεται πως μπορεί να αναλυθεί περαιτέρω σε πιο συγκεκριμένες οδηγίες οι οποίες μπορούν να ληφθούν υπόψη από επαγγελματίες του χώρου, όπως αναλυτές, σχεδιαστές και προγραμματιστές συστημάτων.

Αξίζει να σημειωθεί ότι η έννοια της ευχρηστίας κατά τον Nielsen ορίζεται στο πλαίσιο της ικανότητας αποδοχής συστήματος (system acceptability), δηλαδή κατά πόσο ένα σύστημα έχει την ικανότητα να ικανοποιήσει όλες τις ανάγκες των χρηστών και των άλλων ενδιαφερομένων, όπως οι πελάτες των χρηστών και οι διευθυντές τους. Στο πλαίσιο αυτό, η ευχρηστία μπορεί να θεωρηθεί ιδιότητα της χρησιμότητας του συστήματος, η οποία με τη σειρά της θεωρείται ιδιότητα της ικανότητας πρακτικής αποδοχής του συστήματος, όπως στο παρακάτω μοντέλο.



Σύμφωνα με την πιο επίσημη προσέγγιση του προτύπου ISO/IEC 9126-1, η ευχρηστία αποτελεί μια ιδιότητα της ποιότητας λογισμικού. Το συγκεκριμένο πρότυπο, που θεσπίστηκε το 2000, αντικατέστησε και αποτελεί εξέλιξη του προτύπου ISO 9126 του 1991, το οποίο με τη σειρά του βασίζεται σε παλαιότερα, δοκιμασμένα και ευρέως χρησιμοποιούμενα μοντέλα ποιότητας λογισμικού, όπως τα FCM (Factors - Criteria - Metrics) και CSQ (Characteristics of Software Quality). Το πρότυπο ISO/IEC 9126-1 περιγράφει την ευχρηστία ως μία από τις έξι κατηγορίες ποιότητας λογισμικού που σχετίζονται με την ανάπτυξη ενός προϊόντος: λειτουργικότητα, αξιοπιστία, ευχρηστία, αποδοτικότητα, συντηρησιμότητα (ευκολία συντήρησης) και φορητότητα.



Το μοντέλο ποιότητας κατά ISO/IEC 9126-1

Ο ορισμός που δίνεται για την ευχρηστία λογισμικού στο συγκεκριμένο πρότυπο είναι η ικανότητα ενός προϊόντος λογισμικού να μπορεί να κατανοηθεί, να είναι εύκολο στη μάθησή του, να χρησιμοποιηθεί και να είναι ελκυστικό στον χρήστη, όταν χρησιμοποιείται υπό συγκεκριμένες συνθήκες.

Αξίζει να σημειωθεί ότι η φράση «όταν χρησιμοποιείται υπό συγκεκριμένες συνθήκες» (η οποία είναι αντίστοιχη της «όταν χρησιμοποιείται σε συγκεκριμένο πλαίσιο χρήσης» στο ISO 9241-11) υπαινίσσεται ότι ένα προϊόν δεν μπορεί να θεωρηθεί ότι κατέχει εγγενή ευχρηστία, παρά μόνο μια δυνατότητα να χρησιμοποιηθεί υπό κάποιες συγκεκριμένες συνθήκες. Έτσι, η ευχρηστία δεν μπορεί να εκτιμηθεί μελετώντας ένα προϊόν απομονωμένα, αλλά μόνο σε συνδυασμό με τους χρήστες, τις ενέργειες που θα πραγματοποιήσουν και το γενικότερο περιβάλλον.

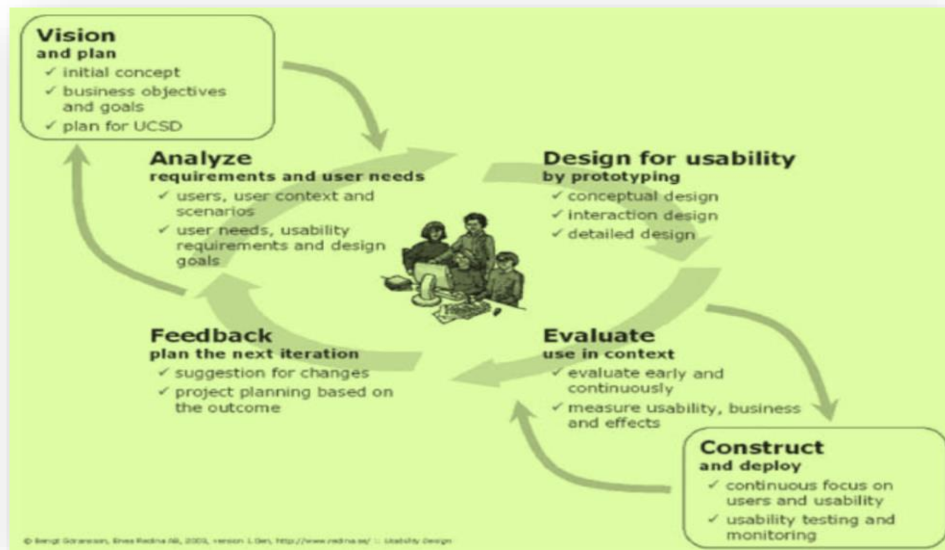
Ο ρόλος και η αυξανόμενη σημασία της ευχρηστίας στην ποιότητα λογισμικού φαίνεται και από το γεγονός ότι σύγχρονες προσεγγίσεις από τη βιβλιογραφία του χώρου θεωρούν ότι ο παραπάνω ορισμός της ευχρηστίας που δίνεται από το ISO 9126-1 δεν είναι αρκετά ευρύς, επισημαίνοντας ότι ο ορισμός που δίνεται από το πρότυπο στον όρο ποιότητα κατά τη χρήση (quality in use) είναι πιο αντιπροσωπευτικός της ευχρηστίας. Η ποιότητα κατά τη χρήση ορίζεται στο ISO 9126-1 ως, η ικανότητα ενός προϊόντος λογισμικού να επιτρέπει σε

συγκεκριμένους χρήστες να επιτύχουν συγκεκριμένους στόχους με αποτελεσματικότητα, παραγωγικότητα, ασφάλεια και υποκειμενική ικανοποίηση σε συγκεκριμένο πλαίσιο χρήσης.

Όπως όμως αναφέραμε και πριν, το λογισμικό θα πρέπει να είναι εύχρηστο στον άνθρωπο. Έτσι λοιπόν εμφανίζεται στο προσκήνιο η μεγαλύτερη ιδιαιτερότητα του σχεδιασμού του λογισμικού. Η προσέγγιση αυτή του σχεδιασμού, η οποία δίνει έμφαση στους ανθρώπους που θα χρησιμοποιήσουν το προϊόν ονομάζεται ανθρωποκεντρικός σχεδιασμός (human-centred design). Το διεθνές πρότυπο ISO 13407, το οποίο αφορά την ανθρωποκεντρική διαδικασία σχεδιασμού και αποτελεί τη βάση για πολλές αντίστοιχες μεθοδολογίες, ορίζει μία αρκετά γενική διαδικασία για την ενσωμάτωση ανθρωποκεντρικών δραστηριοτήτων στον κύκλο ζωής ανάπτυξης (development lifecycle) ενός προϊόντος, χωρίς όμως να καθορίζει συγκεκριμένες μεθόδους. Το μοντέλο σχεδιασμού που παρουσιάζεται στο παραπάνω πρότυπο και φαίνεται στο σχήμα παρακάτω, είναι ένα αρκετά γενικό μοντέλο που αποσκοπεί στην ενσωμάτωση κάποιων αρχών και κατευθυντήριων οδηγιών ανθρωποκεντρικού χαρακτήρα στη διαδικασία ανάπτυξης.

Νεότερες προσπάθειες ορισμού της έννοιας του ανθρωποκεντρικού σχεδιασμού αναδεικνύουν μία αυξανόμενη τάση για έμφαση στην ευχρηστία. Χαρακτηριστικά, ο ανθρωποκεντρικός σχεδιασμός ορίζεται ως μία διαδικασία η

οποία εστιάζει στην ευχρηστία καθ' όλη τη διαδικασία ανάπτυξης και επιπρόσθετα καθ' όλο τον κύκλο ζωής ενός συστήματος.



Το γενικό μοντέλο που περιγράφει τον ανθρωποκεντρικό σχεδιασμό κατά τον ορισμό αυτό φαίνεται στο παραπάνω σχήμα. Στο παραπάνω μοντέλο είναι προφανές ότι η ευχρηστία κατέχει κυρίαρχο ρόλο στη διαδικασία ανθρωποκεντρικού σχεδιασμού.

Εδώ αξίζει να σημειωθεί πως, αν και στο συγκεκριμένο πρότυπο η διαδικασία σχεδιασμού αναφέρεται ως ανθρωποκεντρική, πολύ συχνά γίνεται λόγος στη βιβλιογραφία σε διαδικασίες χρηστοκεντρικού (user-centred) σχεδιασμού, περιγράφοντας και εννοώντας ακριβώς την ίδια διαδικασία. Η διαφορά στους δύο όρους είναι πολύ λεπτή και δυσδιάκριτη με αποτέλεσμα πρακτικά οι δύο όροι να χρησιμοποιούνται στη βιβλιογραφία κατ' εναλλαγή.

Πιο συγκεκριμένα, η ανθρωποκεντρική προσέγγιση για το σχεδιασμό υπολογιστικών συστημάτων θεωρείται πως ακολουθεί μια κοινωνικο-τεχνική θεώρηση, προσπαθώντας να ισορροπήσει τις απαιτήσεις δύο «ανταγωνιστικών» συστημάτων :

• Το κοινωνικό σύστημα που σχετίζεται με την αλληλεπίδραση ανθρώπινων δραστηριοτήτων, πολλαπλούς (και συχνά αντικρουόμενους) στόχους, την ανθρώπινη κατανόηση και γνώση, το επιχειρηματικό πλαίσιο, και τις πρακτικές και τη γενικότερη κουλτούρα όσον αφορά συγκεκριμένες εφαρμογές και ενέργειες.

• Το τεχνικό σύστημα που σχετίζεται με αυστηρές, τυπικές και βασισμένες σε κανόνες διαδικασίες και την αντίστοιχη τεχνολογία, ενώ για τη διαχείριση και διοίκησή του χρησιμοποιούνται δείκτες απόδοσης και τρόποι χειρισμού εξαιρέσεων.

Από την άλλη πλευρά, η χρηστοκεντρική προσέγγιση θεωρείται πως εστιάζει περισσότερο στην τεχνολογία (δηλ. το τεχνικό σύστημα) και λιγότερο στον άνθρωπο (το κοινωνικό σύστημα).

Για λόγους συνέπειας σε αυτή την εργασία χρησιμοποιείται ο όρος «ανθρωποκεντρικός» για να περιγραφεί αυτή η προσέγγιση στο σχεδιασμό και την ανάπτυξη υπολογιστικών συστημάτων, καθώς κύριο αντικείμενο της μελέτης δεν είναι τα μοντέλα και οι διαδικασίες σχεδιασμού και ανάπτυξης, αλλά οι μέθοδοι και τα εργαλεία που επιτρέπουν και διευκολύνουν το σχεδιασμό και την ανάπτυξη της φιλοσοφίας αυτής. Εξάλλου, φαίνεται πως ο όρος «ανθρωποκεντρικός» χρησιμοποιείται περισσότερο στην ελληνική βιβλιογραφία για να περιγράψει την προσέγγιση που έχει περιγραφεί παραπάνω.

2. Σχεδίαση Βάσει Μοντέλου (MDD)

Η δραματική αύξηση της πολυπλοκότητας των λογισμικών συστημάτων ώθησε τους σχεδιαστές συστημάτων στη χρήση γλωσσών σχεδιασμού των συστημάτων. Η στροφή του ερευνητικού ενδιαφέροντος προς τις γλώσσες μοντελοποίησης συστημάτων οδήγησε στην δημιουργία μιας πληθώρας τεχνικών και τεχνολογιών μοντελοποίησης τις πιο σημαντικές από τις οποίες αναφέρουμε παρακάτω.

Σήμερα, κυρίαρχο ρόλο στη σχεδίαση λογισμικών συστημάτων παίζει η Unified Modeling Language (UML). Η UML είναι μια γραφική γλώσσα μοντελοποίησης γενικού σκοπού η οποία έχει σχεδιαστεί έτσι ώστε να περιγράφει και να προδιαγράφει τη δομή και τη λειτουργία ενός συστήματος για την εκπλήρωση ενός συγκεκριμένου σκοπού. Για την μοντελοποίηση και την περιγραφή των συστημάτων λογισμικού η UML διαθέτει μια πληθώρα διαγραμμάτων, τα οποία περιγράφουν το σύστημα το καθένα από διαφορετική σκοπιά και προδιαγράφουν συγκεκριμένα χαρακτηριστικά του συστήματος. Πιο αναλυτικά η UML χρησιμοποιείται για να μοντελοποιήσει τις εξής πτυχές ενός λογισμικού συστήματος:

α) Λειτουργικές απαιτήσεις .

- Διαγράμματα χρήσης

Με την χρήση των διαγραμμάτων χρήσης μοντελοποιούνται συνήθως οι λειτουργικές απαιτήσεις του συστήματος διαφόρων σεναρίων αλληλεπίδρασης των χρηστών με το σύστημα. Περιγράφουν δηλαδή τι προβλέπεται να κάνει το σύστημα. Οι έννοιες που χρησιμοποιούνται στα διαγράμματα χρήσης είναι οι δράστες, τα σενάρια χρήσης και το υποκείμενο σύστημα. Το υποκείμενο σύστημα είναι το σύστημα που περιγράφουμε τη συμπεριφορά του σε κάθε σενάριο που εξετάζουμε. Οι δράστες είναι οι χρήστες του συστήματος ή άλλα συστήματα τα οποία είναι εξωτερικά του υποκείμενου συστήματος και αλληλεπιδρούν με αυτό.

β) Στατική δομή του συστήματος.

- Διαγράμματα κλάσεων.
- Διαγράμματα αντικειμένων
- Ψηφιδικά διαγράμματα
- Παραταξιακά διαγράμματα

Με τη χρήση αυτών των διαγραμμάτων απεικονίζουμε στην UML τη στατική δομή του συστήματος και τις σχέσεις που έχουν οι διάφορες δομικές μονάδες μεταξύ τους.

γ) Δυναμική συμπεριφορά του συστήματος

• ακολουθιακά διαγράμματα

• συνεργατικά διαγράμματα

Με τη χρήση αυτών των διαγραμμάτων μοντελοποιούνται τα δυναμικά χαρακτηριστικά του συστήματος. Τα διαγράμματα αυτά απεικονίζουν τον τρόπο επικοινωνίας μεταξύ των διαφόρων δομικών στοιχείων του συστήματος.

δ) Συμπεριφορά συγκεκριμένων αντικειμένων.

• διαγράμματα κατάστασης

Τα διαγράμματα κατάστασης περιγράφουν τις καταστάσεις στις οποίες μπορεί να βρεθεί ένα αντικείμενο σε σχέση με τα μηνύματα που λαμβάνει ή τα γεγονότα που παρατηρεί. Πρόκειται στην ουσία για ένα αυτόματο περιορισμένων ή άπειρων καταστάσεων το οποίο μοντελοποιεί τη συμπεριφορά ενός συγκεκριμένου αντικειμένου.

ε) Στάδια εκτέλεσης μιας διαδικασίας

• διαγράμματα δραστηριότητας

Τα διαγράμματα αυτά μοντελοποιούν στάδια διαδικασιών σε εφαρμογές ανταλλαγής δεδομένων ή σε εφαρμογές που έχουν πολύπλοκη λογική ροή.

Αν και τα UML διαγράμματα περιγράφονται αναλυτικά στα φύλλα προδιαγραφής[2] της OMG, κρίνεται απαραίτητο να δώσουμε την σημειογραφία των ακολουθιακών διαγραμμάτων η επεξεργασία των οποίων είναι το αντικείμενο της εργασίας αυτής. Τα ακολουθιακά διαγράμματα χρησιμοποιούνται[3] κυρίως για να απεικονίσουν τις αλληλεπιδράσεις μεταξύ των στοιχείων του συστήματος με την χρονική σειρά που αυτές συμβαίνουν. Κατά τη διάρκεια της φάσης της σχεδίασης ενός συστήματος οι μηχανικοί και οι προγραμματιστές χρησιμοποιούν τα ακολουθιακά διαγράμματα για να καθορίσουν και να απεικονίσουν τις αλληλεπιδράσεις ενός αντικειμένου και με αυτόν τον τρόπο να καταστήσουν πιο ολοκληρωμένη την περιγραφή ενός συστήματος.

---

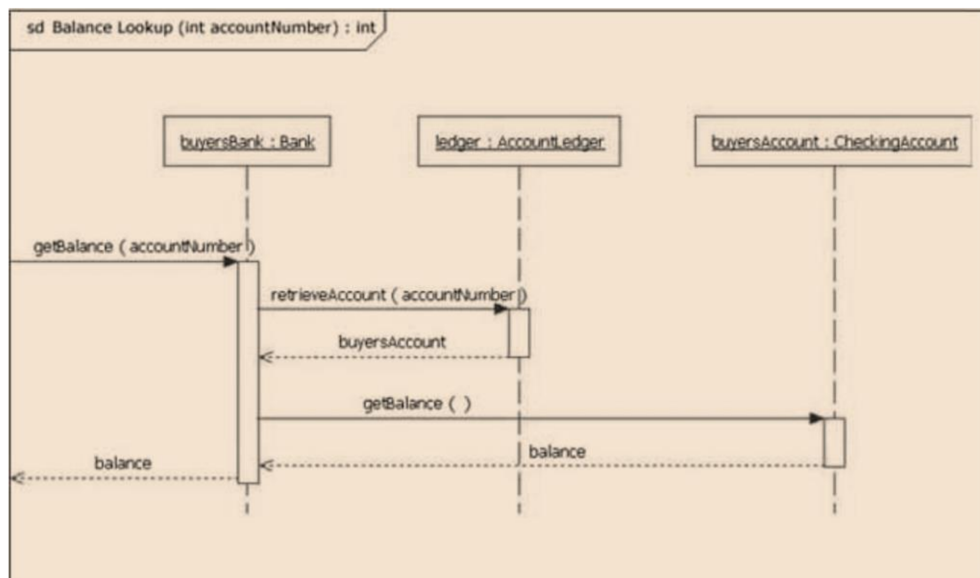
[2] *Object Management Group, Inc., UML Infrastructure Specification, version 2.1.2, November 2007 & Object Management Group, Inc., UML Superstructure Specification, version 2.1.2, November 2007*

[3] *Donald Bell UML's Sequence Diagram http://www.ibm.com/developerworks/rational/library/3101.html*

Μια από τις κυριότερες χρήσεις των ακολουθιακών διαγραμμάτων είναι η μετάβαση από τις λειτουργικές απαιτήσεις, οι οποίες μοντελοποιούνται μέσω των διαγραμμάτων χρήσης, σε ένα επόμενο και πιο φορμαλιστικό επίπεδο προδιαγραφής. Τα σενάρια χρήσης μετατρέπονται σε ένα ή περισσότερα ακολουθιακά διαγράμματα. Εκτός από τη φάση της σχεδίασης τα ακολουθιακά διαγράμματα μπορούν να έχουν και έναν πρόσθετο ρόλο. Μπορούν να χρησιμοποιηθούν ως βοηθητικά δεδομένα για την κατανόηση του τρόπου αλληλεπίδρασης των διαφόρων δομικών στοιχείων ενός ήδη υπάρχοντος συστήματος. Σε αυτήν την περίπτωση τα ακολουθιακά διαγράμματα είναι πολύ χρήσιμα σε περιπτώσεις συντήρησης ενός λογισμικού συστήματος ή σε περίπτωση μεταφοράς του από ένα πρόσωπο ή οργανισμό σε έναν άλλο.

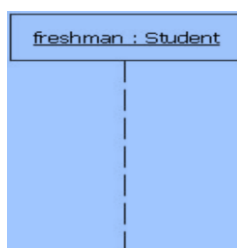
Στην ανάλυση μας, θα ακολουθήσουμε τις αρχές της UML 2 όπως αυτές ορίζονται από τον OMG στα κείμενα προδιαγραφών της UML 2[4] .Όπως σε κάθε UML διάγραμμα στην UML 2 αρχικά ορίζουμε το πλαίσιο του διαγράμματος. Το πλαίσιο χρησιμεύει πρωταρχικά για να ορίσουμε τον χώρο του διαγράμματος. Επίσης το πλαίσιο χρησιμεύει για να προσφέρει ένα συγκεκριμένο χώρο ονοματολογίας του διαγράμματος. Αν και δεν είναι απαραίτητο το πλαίσιο περιέχει το στοιχείο «ετικέτα» το οποίο είναι το όνομα του διαγράμματος και χρησιμεύει στην αναφορά του διαγράμματος από άλλα διαγράμματα ή κείμενα. Το πλαίσιο έχει έναν επιπλέον σημαντικό ρόλο στα διαγράμματα τα οποία απεικονίζουν αλληλεπιδράσεις, όπως τα ακολουθιακά διαγράμματα. Στα ακολουθιακά διαγράμματα τα εισερχόμενα ή τα εξερχόμενα από το σύστημα μηνύματα μπορούν να συνδεθούν με το πλαίσιο.

[4] *Object Management Group, Inc.,Meta Object Facility (MOF) Specification version 1.4.1, May 2005*



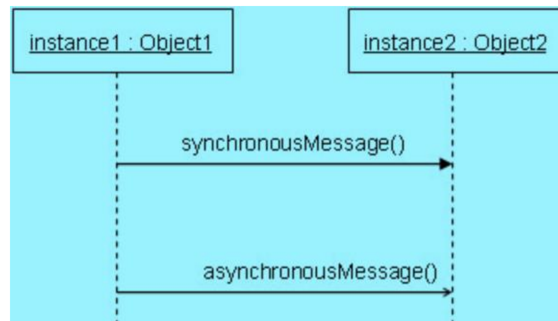
Το πλαίσιο ενός ακολουθιακού διαγράμματος

Γραμμές ετοιμότητας: Όταν σχεδιάζουμε ένα ακολουθιακό διάγραμμα κάθε αντικείμενο που συμμετέχει στην διαδικασία που μοντελοποιούμε παριστάνεται με μία γραμμή ετοιμότητας. Η γραμμή ετοιμότητας (lifeline) σχεδιάζεται ως ένα παραλληλόγραμμο με μία γραμμή κάθετη στην κάτω βάση του. Το όνομα της γραμμής ετοιμότητας τοποθετείται μέσα στο παραλληλόγραμμο. Η τυπική ονομασία κάθε αντικειμένου σε ένα ακολουθιακό διάγραμμα είναι η εξής:
Όνομα Αντικειμένου : Όνομα Κλάσης.



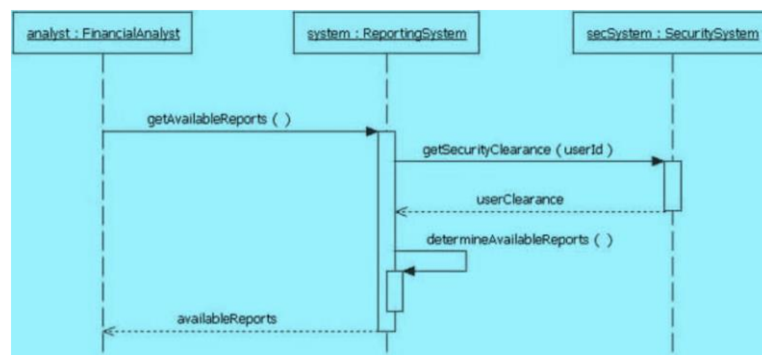
Μηνύματα: Τα μηνύματα στα ακολουθιακά διαγράμματα μοντελοποιούν την επικοινωνία και τις αλληλεπιδράσεις μεταξύ των στοιχείων του συστήματος. Τα μηνύματα μπορούν να αντιπροσωπεύουν κλήση κάποιας συνάρτησης, δημιουργία κάποιου αντικειμένου ή καταστροφή του. Για να δείξουμε ένα αντικείμενο το οποίο στέλνει μήνυμα σε ένα άλλο αντικείμενο σχεδιάζουμε ένα βέλος του οποίου η αρχή είναι στον αποστολέα (source) και το τέλος στον παραλήπτη (target). Το όνομα του μηνύματος ή της μεθόδου που καλείται τοποθετείται πάνω από το βέλος. Στην περίπτωση σύγχρονου μηνύματος η μύτη του βέλους είναι συμπαγής ενώ σε

περίπτωση ασύγχρονου μηνύματος η μύτη του βέλους είναι μη συμπαγής. Η απόκριση ενός αντικείμενου σε μήνυμα παριστάνεται με διακεκομμένη γραμμή.



Σύγχρονο και ασύγχρονο μήνυμα

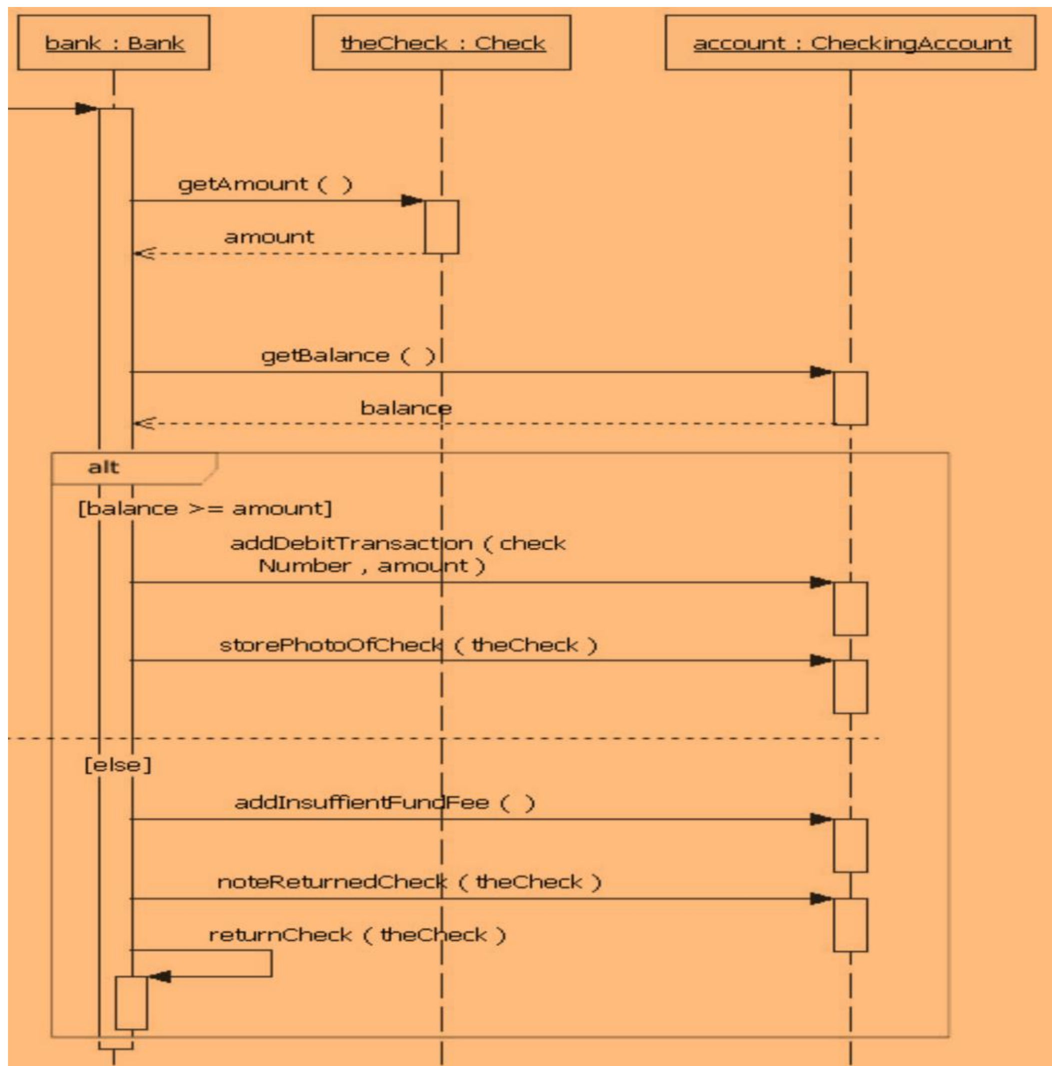
<u>Σύνθετα τμήματα</u>: Όταν μοντελοποιούμε κάποιες αλληλεπιδράσεις μεταξύ αντικειμένων υπάρχουν φορές που κάποια μηνύματα αποστέλλονται μόνο όταν ικανοποιούνται μία ή περισσότερες συνθήκες. Επίσης είναι πιθανό να υπάρχουν βρόγχοι επανάληψης αποστολής και λήψης μηνυμάτων μέχρι να ικανοποιηθεί κάποια συνθήκη. Για την μοντελοποίηση της ροής ελέγχου μιας διαδικασίας χρησιμοποιούνται στην UML 2 τα σύνθετα τμήματα (combined fragments) τα οποία απεικονίζουν στα διαγράμματα τον έλεγχο ροής.



Απλό ακολουθιακό διάγραμμα

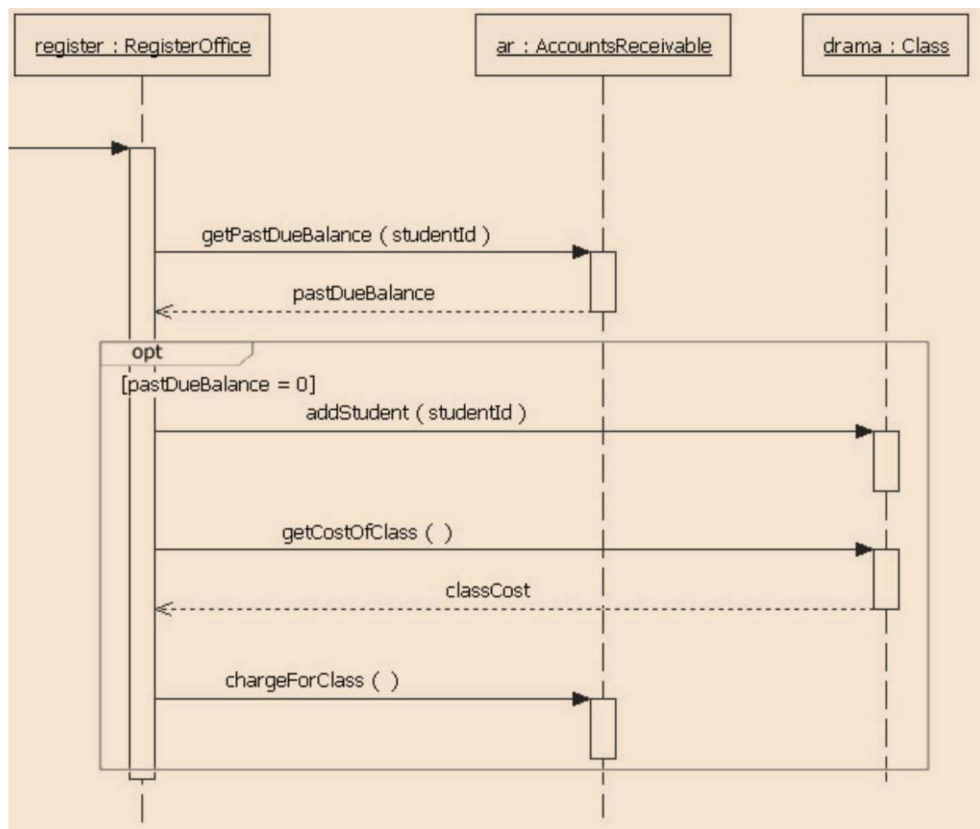
<u>Εναλλακτικές</u>: Τα alternatives -όπως ονομάζονται- χρησιμοποιούνται για να μοντελοποιήσουν τη συνήθη δομή *if (condition) then ... else*. Η δομή του alternative αποτελείται από δύο τμήματα. Τα δύο τμήματα ορίζονται στο διάγραμμα μέσα σε ένα πλαίσιο το οποίο χωρίζεται σε δύο υποπλαίσια. Σαν πρώτο τμήμα, ορίζεται το πρώτο υποπλαίσιο με ετικέτα *alt* και την συνθήκη σύγκρισης η οποία τοποθετείται πάνω στην γραμμή ετοιμότητας (lifeline) του αντικειμένου με το οποίο σχετίζεται η σύγκριση. Περιέχει την συμπεριφορά του συστήματος αν ικανοποιείται η συνθήκη σύγκρισης. Το δεύτερο τμήμα ορίζεται

από το δεύτερο υποπλαίσιο με ετικέτα *else* και περιέχει την συμπεριφορά του συστήματος όταν δεν ικανοποιείται η συνθήκη.



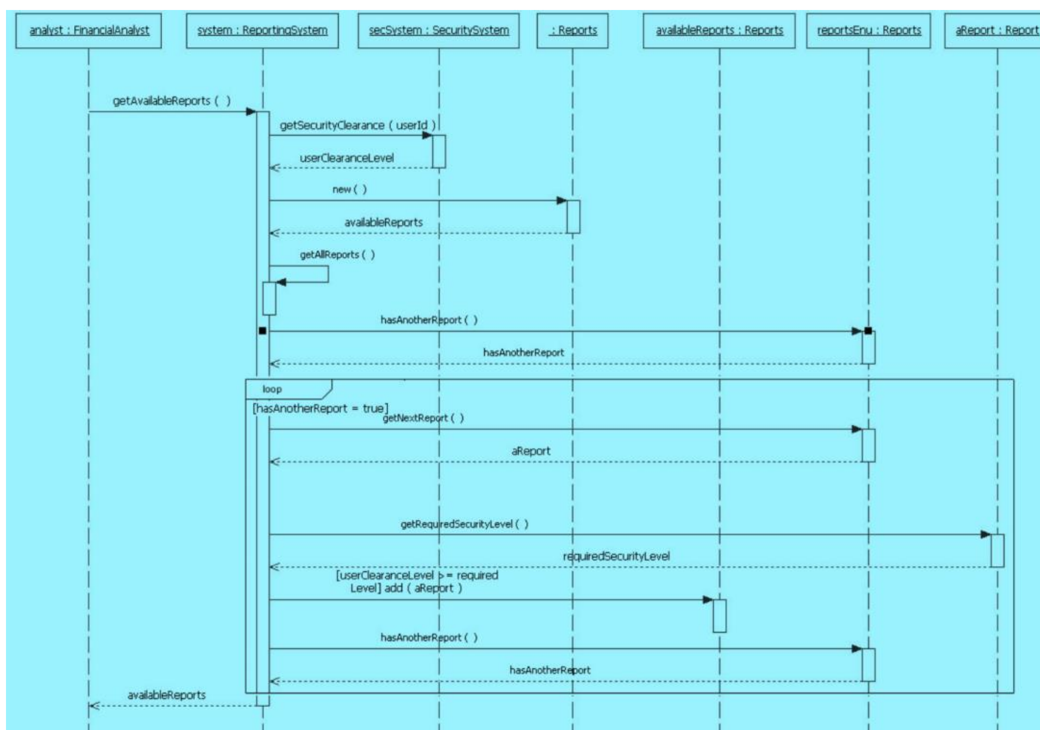
Διαγράμμα με alternative

<u>Επιλογές (Options)</u>: Το option χρησιμοποιείται για να μοντελοποιηθεί μια δομή *if (condition) then*, χωρίς την ύπαρξη του τμήματος else. Η δομή option ορίζεται από ένα πλαίσιο το οποίο περιέχει την ετικέτα opt και την συνθήκη σύγκρισης η οποία τοποθετείται πάνω στην γραμμή ετοιμότητας (lifeline) του αντικειμένου με το οποίο σχετίζεται η σύγκριση. Το πλαίσιο περιέχει την συμπεριφορά του συστήματος στην περίπτωση που ικανοποιείται η συνθήκη. Αν δεν ικανοποιείται η συνθήκη το πλαίσιο παραλείπεται και θεωρούμε ότι η ροή ελέγχου του συστήματος συνεχίζεται αμέσως μετά την δομή option.



Διαγράμματος με χρήση δομής option

<u>Βρόγχοι (Loops)</u>: Για να μοντελοποιήσουμε βρόγχους επαναλαμβανόμενων μηνυμάτων χρησιμοποιούμε τη δομή loop. Η δομή του loop είναι παρόμοια με αυτή του option. Και εδώ η δομή loop ορίζεται από ένα πλαίσιο το οποίο έχει την ετικέτα loop. Πάνω στη γραμμή ετοιμότητας του αντικειμένου που σχετίζεται με τον βρόγχο ορίζεται η συνθήκη του βρόγχου. Το πλαίσιο περιέχει τις αλληλεπιδράσεις των αντικειμένων που επαναλαμβάνονται κατά τη διάρκεια του βρόγχου.



Διαγράμματος με δομή loop.

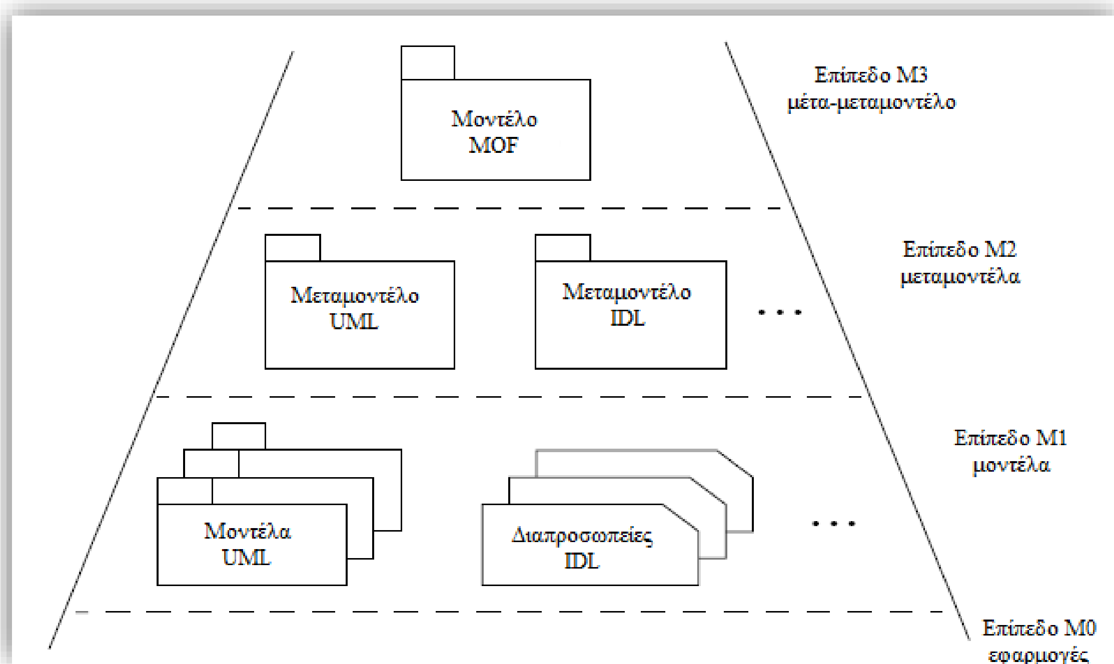
2.1 Μοντελοκεντρική Αρχιτεκτονική

Συνεχίζοντας, περνάμε στην Μοντελοκεντρική Αρχιτεκτονική. Το Meta Object Facility (MOF) αποτελεί μια προσπάθεια να συστηματοποιηθεί ο τρόπος μοντελοποίησης των συστημάτων λογισμικού. Σύμφωνα με την Μοντελοκεντρική Αρχιτεκτονική που προτείνει ο OMG, κάθε μοντέλο πρέπει να υπακούει σε κάποιο μεταμοντέλο. Το μεταμοντέλο[5] είναι ένα μοντέλο το οποίο μοντελοποιεί την υλοποίηση ενός άλλου μοντέλου. Το MOF[6] ορίζει μια αφηρημένη γλώσσα και ένα περιβάλλον πλαίσιο για τον ορισμό και την διαχείριση τεχνολογικά ανεξάρτητων μεταμοντέλων. Η προσέγγιση του MOF στοχεύει στην επεκτασιμότητα των δυνατοτήτων των μοντέλων των λογισμικών συστημάτων. Ο σκοπός δηλαδή είναι να παρέχει ένα πλαίσιο το οποίο να υποστηρίζει κάθε είδος μεταδεδομένων και να επιτρέπει νέα είδη μεταδεδομένων να προστίθενται όποτε αυτό είναι αναγκαίο. Για να επιτευχθεί αυτός σκοπός, το MOF βασίζεται σε μια διαστρωματωμένη αρχιτεκτονική 4 επιπέδων. Η αρχιτεκτονική αυτή είναι αρκετά διαδεδομένη σε διαδικασίες και οργανισμούς προτυποποίησης όπως ο ISO και ο CDIF. Το κλειδί

[5] *Object Management Group, Inc., Meta Object Facility (MOF) Core Specification version 2.0, January 2006.*
[6] *Object Management Group, Inc., Meta Object Facility (MOF) Specification version 1.4.1, May 2005*

αυτής της αρχιτεκτονικής είναι το επίπεδο του μετα-μεταμοντέλου το οποίο συνδέει τα μοντέλα και τα αντίστοιχα σε αυτά μεταμοντέλα.

Η δομή αυτής της αρχιτεκτονικής των 4 επιπέδων είναι η εξής: *Στη βάση* της ιεραρχίας των 4 επιπέδων βρίσκεται το επίπεδο των πληροφοριών ή επίπεδο M0. Αποτελείται δηλαδή αυτό το επίπεδο από τις πληροφορίες που θέλουμε να περιγράψουμε. *Το αμέσως ανώτερο επίπεδο* είναι το επίπεδο του μοντέλου ή M1. Το επίπεδο αυτό αποτελείται από τα μεταδεδομένα τα οποία περιγράφουν τα δεδομένα του επιπέδου πληροφοριών. *Το τρίτο επίπεδο* είναι το επίπεδο του μεταμοντέλου ή M2. Το επίπεδο αυτό αποτελείται από την περιγραφή της δομής και της σημασιολογίας των μεταδεδομένων. Πρόκειται δηλαδή για μια συλλογή μέτα-μεταδεδομένων. Η συλλογή αυτή των μέτα-μεταδεδομένων αποτελεί ένα μεταμοντέλο. *Στην κορυφή* της ιεραρχίας είναι το επίπεδο του μέτα-μεταμοντέλου ή M3. Αποτελείται από την περιγραφή της δομής και της σημασιολογίας των μέτα-μεταδεδομένων. Με άλλα λόγια είναι μια αφηρημένη γλώσσα ορισμού διαφορετικών ειδών μεταδεδομένων .



Κύριο χαρακτηριστικό αυτής της αρχιτεκτονικής είναι ότι κάθε επίπεδο αποτελεί ένα στιγμιότυπο του αμέσως ανώτερου επιπέδου. Το επίπεδο M3 ορίζεται αναδρομικά από το ίδιο το επίπεδο. Με αυτόν τον τρόπο αποφεύγουμε τον ορισμό άπειρων επιπέδων. Η αρχιτεκτονική του MOF βασίζεται στην αρχιτεκτονική των 4 επιπέδων που μόλις περιγράψαμε. Το παρακάτω παράδειγμα

είναι ένα τυπικό στιγμιότυπο της αρχιτεκτονικής του MOF με μεταμοντέλα που αντιπροσωπεύουν τα UML διαγράμματα και την OMG IDL.

Η αρχιτεκτονική του MOF παρουσιάζει κάποια σημαντικά πλεονεκτήματα που την κάνουν να υπερέχει από προηγούμενες αρχιτεκτονικές μεταμοντέλων.

• Το μοντέλο MOF είναι αντικειμενοστραφές και οι δομές που χρησιμοποιεί είναι εναρμονισμένες με την UML και τις αντικειμενοστραφείς γλώσσες προγραμματισμού.

• Τα επίπεδα του MOF δεν είναι σταθερά. Αν και τα επίπεδα είναι συνήθως 4, θα μπορούσαν να να είναι περισσότερα ή λιγότερα, ανάλογα με το πώς αναπτύσσεται η μοντελοποίηση. Το MOF δεν απαιτεί να υπάρχουν ξεχωριστά επίπεδα μοντελοποίησης. Τα επίπεδα του MOF είναι μια σύμβαση για την κατανόηση των σχέσεων μεταξύ δεδομένων και μεταδεδομένων.

• Το μοντέλο MOF περιγράφει τον εαυτό του. Μπορούμε να το ορίσουμε πλήρως χρησιμοποιώντας τις δικές του δομές μοντελοποίησης.

Η τρέχουσα έκδοση του MOF είναι το MOF2. Το MOF2 έχει επηρεάσει και είναι ταυτόχρονα επηρεασμένο από την δομή της UML2 λόγω της θέλησης του OMG να δημιουργηθεί ένας κοινός πυρήνας εννοιών μοντελοποίησης μεταξύ της UML, του MOF, αλλά και των υπολοίπων προτύπων του OMG. Το σημαντικό είναι ότι η UML2 και το MOF2 χρησιμοποιούν μια επαναχρησιμοποιήσιμη βιβλιοθήκη υποδομής της UML 2 και αυτό έχει πολλά πλεονεκτήματα από τα οποία τα κυριότερα είναι τα εξής:

• Οι κανόνες μοντελοποίησης μεταδεδομένων είναι πιο απλοί από την προηγούμενη έκδοση του MOF.

• Οι διάφορες τεχνολογικές απεικονίσεις του MOF έχουν εφαρμογή σε μεγαλύτερο εύρος UML μοντέλων

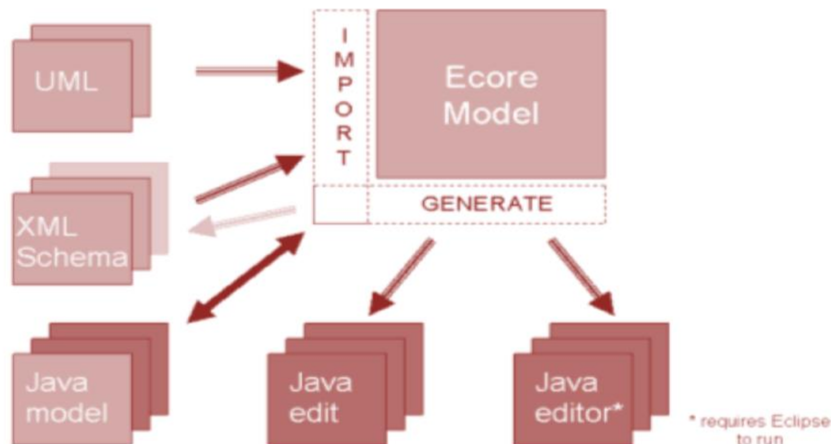
• Λόγω των κοινών εννοιών UML 2 και MOF 2, μπορεί να χρησιμοποιηθεί οποιοδήποτε εργαλείο μοντελοποίησης σε UML για την σχεδίαση ενός MOF μεταμοντέλου.

Στην βελτιωμένη έκδοση του MOF προβλέπονται δύο εκδόσεις του MOF. Το βασικό MOF (Essential MOF-EMOF) και το πλήρες MOF (Complete MOFCMOF). Το EMOF μοντέλο αποτελεί ένα υποσύνολο του MOF το οποίο αντιστοιχεί στις έννοιες των αντικειμενοστραφών γλωσσών προγραμματισμού και του XML. Η σημασία του EMOF είναι το γεγονός ότι παρέχει ένα πλαίσιο για την άμεσες απεικονίσεις μοντέλων MOF σε εφαρμογές όπως η JMI και η XMI για απλά

μεταμοντέλα. Ένας πρωτεύων στόχος του EMOF είναι ο ορισμός απλών μεταμοντέλων ενώ ταυτόχρονα θα υπάρχει υποστήριξη επεκτάσεων των απλών αυτών μεταμοντέλων ώστε να προκύψουν πιο πολύπλοκα μεταμοντέλα χρησιμοποιώντας το CMOF. Χαρακτηριστικό του EMOF είναι ότι ενώ ορίζεται σαν ένα CMOF μοντέλο μπορεί να οριστεί χωρίς τη χρήση του CMOF αλλά αναδρομικά χρησιμοποιώντας τον εαυτό του.

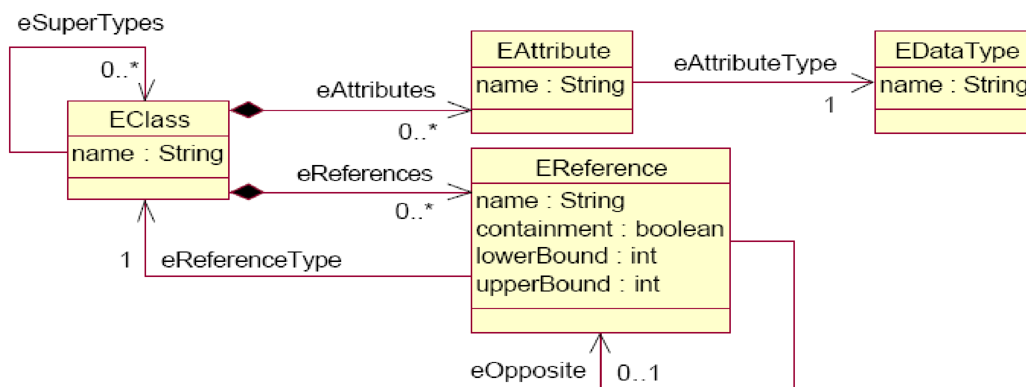
Το CMOF είναι το μεταμοντέλο που χρησιμοποιείται για να ορίσει άλλα μεταμοντέλα όπως τη UML 2. Το CMOF παράγεται από το EMOF και από το πακέτο Core::Constructs της UML 2. Όπως το EMOF, έτσι και το CMOF μπορεί να οριστεί χωρίς τη χρήση του EMOF, αλλά αναδρομικά χρησιμοποιώντας τον εαυτό του. Παρατηρούμε τη στενή σχέση μεταξύ του MOF 2 και της UML 2 αλλά και την ευελιξία του MOF, καθώς τόσο το EMOF όσο και το CMOF ενώ μπορούν να σταθούν σαν ανεξάρτητα μεταμοντέλα, ταυτόχρονα προκύπτουν το ένα από το άλλο.

Τέλος, έχουμε το Eclipse Modeling Framework (EMF) αποτελεί μια προσπάθεια υλοποίησης της Μοντελοκεντρικής Αρχιτεκτονικής από την IBM και βασίζεται στην πλατφόρμα Eclipse. Το EMF δίνει στον σχεδιαστή την δυνατότητα να σχεδιάσει μοντέλα που περιγράφουν ένα σύστημα λογισμικού. Δεδομένου ότι ο πηγαίος κώδικας είναι ένα μοντέλο περιγραφής του συστήματος, τα UML διαγράμματα είναι επίσης μοντέλα περιγραφής του συστήματος όπως επίσης και τα XML σχήματα, το EMF προσπαθεί να ενοποιήσει τις υπάρχουσες τεχνικές μοντελοποίησης και να δώσει στον μηχανικό λογισμικού την δυνατότητα να ορίσει το σύστημα στο μοντέλο με το οποίο αυτός είναι πιο εξοικειωμένος αλλά ταυτόχρονα να έχει και την περιγραφή των άλλων μοντέλων. Σχηματικά η αρχιτεκτονική του EMF φαίνεται στο παρακάτω σχήμα.



Η αρχιτεκτονική του Eclipse Modeling Framework

Το μεταμοντέλο το οποίο χρησιμοποιείται για την μοντελοποίηση των συστημάτων είναι το Ecore. Το Ecore είναι εμπνευσμένο και παραπλήσιο του MOF. Το Ecore είναι ένα υποσύνολο της UML όπως ακριβώς και το EMOF. Η αντιστοιχία των τύπων του MOF με το Ecore είναι σχεδόν 1-1 και οι διαφορές τους είναι κυρίως λεξικογραφικού χαρακτήρα. Συνεπώς είναι αρκετά εύκολη η μετατροπή ενός μοντέλου το οποίο υπακούει στο EMOF ή στο CMOF σε ένα μοντέλο το οποίο να υπακούει στο Ecore και αντίστροφα. Η ομοιότητα των δύο μεταμοντέλων φαίνεται και στο παρακάτω σχήμα στο οποίο απεικονίζεται ο πυρήνας του Ecore (Ecore kernel).

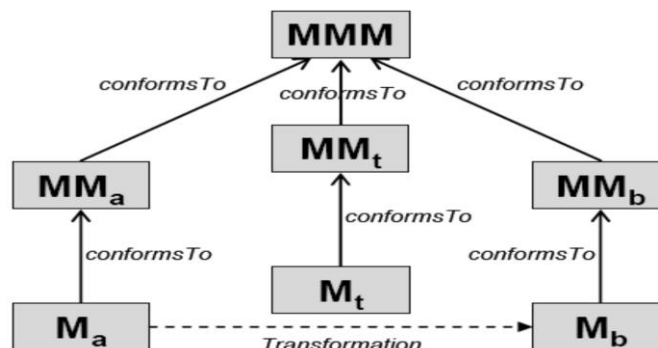


Ο πυρήνας του Ecore

Η ύπαρξη διαφορετικών προτύπων μοντελοποίησης, αλλά και η πληθώρα των διαγραμμάτων που χρειάζεται για την απεικόνιση όλων των πλευρών ενός

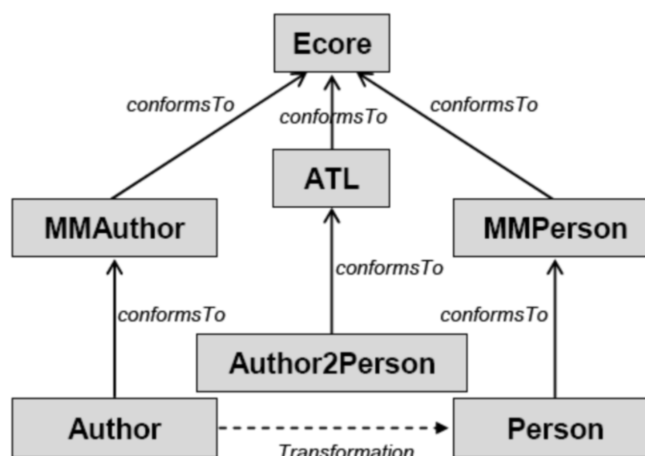
λογισμικού συστήματος οδήγησε στην ανάπτυξη εργαλείων μετασχηματισμού μοντέλων έτσι ώστε να είναι δυνατή η σύνθεση πληροφοριών από ένα ή περισσότερα μοντέλα για την παραγωγή νέων μοντέλων και διαγραμμάτων.

Η Atlas Transformation Language (ATL) για παράδειγμα, αποτελεί ένα έργο των οργανισμών Laboratoire d'Informatique de Nantes Atlantique (LINA) καιτου Institut national de recherche en informatique et en automatique (INRIA) . Σκοπός της ATL[7] είναι να προσφέρει στους σχεδιαστές λογισμικών συστημάτων ένα μέσο για την παραγωγή ενός συνόλου μοντέλων από ένα άλλο σύνολο μοντέλων. Ένα πρόγραμμα ATL αποτελείται από ένα σύνολο κανόνων μετασχηματισμού οι οποίοι ορίζουν πως τα στοιχεία των αρχικών μοντέλων συνδυάζονται και χρησιμοποιούνται για την αρχικοποίηση και παραγωγή των στοιχείων των τελικών μοντέλων. Η ATL βασίζεται πάνω στην πλατφόρμα του Eclipse και προσφέρει μια μεγάλη ποικιλία εργαλείων για τον σχεδιασμό και την υλοποίηση των κανόνων μετασχηματισμού. Η ATL είναι ένας συνδυασμός δηλωτικού και προστακτικού προγραμματισμού. Υπακούει στο Ecore μεταμοντέλο και ενώ τα στοιχεία και οι τύποι που χρησιμοποιεί βασίζονται στην OCL (Object Constraint Language) του OMG. Στην ATL αλλά και γενικότερα στην Μοντελοκεντρική Αρχιτεκτονική Λογισμικού, ο μετασχηματισμός ενός μοντέλου σε ένα άλλο μοντέλο αποτελεί και αυτός ένα μοντέλο το οποίο πρέπει να υπακούει σε ένα μεταμοντέλο. Επίσης τα μεταμοντέλα που ορίζουν τα μοντέλα που λαμβάνουν μέρος στο μετασχηματισμό θα πρέπει να υπακούν στο ίδιο μετα-μεταμοντέλο. Στο σχήμα Α που ακολουθεί, συνοψίζεται η πλήρης διαδικασία του μετασχηματισμού των μοντέλων ενώ στο σχήμα Β περιγράφεται ο μετασχηματισμός μοντέλων χρησιμοποιώντας την ATL.



Σχήμα Α : Διαδικασία μετασχηματισμού μοντέλων

[7] *Atlas Group, LINA & INRIA, Atlas Transformation Language, ATL User Manual, version 0.7, February2006*



Σχήμα Β: Μετασχηματισμός μοντέλου σε ATL

Το Model Transformation Framework είναι ένα έργο της IBM και αποτελείται από ένα σύνολο εργαλείων τα οποία βοηθούν τους σχεδιαστές λογισμικού να μεταβαίνουν από ένα EMF μοντέλο σε ένα άλλο EMF μοντέλο. Προφανώς το MTF βασίζεται πάνω στην πλατφόρμα του Eclipse και στο Eclipse Modeling Framework. Τα μοντέλα αυτά μπορούν να υπακούν στο ίδιο μεταμοντέλο ή και σε διαφορετικά μεταμοντέλα. Το MTF προσφέρει μια γλώσσα ορισμού κανόνων οι οποίοι ορίζουν την μορφή του τελικού μοντέλου έτσι ώστε να είναι το τελικό μοντέλο συνεπές ως προς το αρχικό μοντέλο. Το MTF δεν είναι τόσο αυστηρό ως προς τον ορισμό των αρχικών μοντέλων και μεταμοντέλων. Μπορούμε να ορίσουμε τους κανόνες μετασχηματισμού και να καθορίσουμε το αρχικό και τελικό μοντέλο κατά την εκτέλεση του μετασχηματισμού.

Ένα από τα σημαντικότερα και πιο ολοκληρωμένα εργαλεία στο χώρο της σχεδίασης, ανάπτυξης και ανάλυσης συστημάτων είναι το Eclipse Modeling Framework (EMF). Το EMF είναι ένα περιβάλλον-πλαίσιο σχεδίασης και ανάπτυξης συστημάτων Java. Το EMF χρησιμοποιεί όπως προαναφέραμε το μεταμοντέλο Ecore για την μοντελοποίηση των συστημάτων, το οποίο είναι αρκετά παραπλήσιο του EMOF και είναι και αυτό υποσύνολο της UML. Το EMF δίνει τη δυνατότητα στον μηχανικό λογισμικού να σχεδιάσει το σύστημα του χρησιμοποιώντας το πρότυπο της επιλογής του και το EMF αναλαμβάνει να δημιουργήσει το μοντέλο του συστήματος σύμφωνα και με τα υπόλοιπα πρότυπα που υποστηρίζει το Eclipse. Για παράδειγμα ο σχεδιαστής ενός συστήματος μπορεί να ορίσει το μοντέλο του χρησιμοποιώντας Annotated Java και το EMF θα κατασκευάσει σύμφωνα με το μοντέλο αυτό τα Ecore και UML διαγράμματα καθώς

επίσης και τα XMI κείμενα που θα περιγράφουν τα διαγράμματα αυτά. Τέλος, από τα παραγόμενα αυτά μοντέλα το EMF παράγει τον κώδικα του συστήματος ο οποίος είναι συνεπής ως προς τα μοντέλα και συντακτικά σωστός. Το Eclipse έχει επίσης τη δυνατότητα να ενσωματώνει εργαλεία (plugins) και να τα χρησιμοποιεί σε συνδυασμό με το EMF ή άλλα εργαλεία. Με αυτόν τον τρόπο αυξάνονται σημαντικά οι δυνατότητες του Eclipse και του EMF με βασικότερη δυνατότητα αυτή της εύκολης επέκτασης των λειτουργιών του περιβάλλοντος-πλαισίου. Χρησιμοποιώντας αυτήν την ιδιότητα του Eclipse έχουν αναπτυχθεί πολλές εφαρμογές οι οποίες βασίζονται πάνω στην πλατφόρμα του Eclipse και παρέχουν σημαντικές δυνατότητες σχεδίασης και ανάλυσης λογισμικών συστημάτων. Μια βασική εφαρμογή η οποία έχει υλοποιηθεί σαν ενσωματωμένο εργαλείο του Eclipse είναι το ATL tool το οποίο είναι η πρακτική εφαρμογή της ATL γλώσσας. Δίνει τη δυνατότητα στον χρήστη της πλατφόρμας του Eclipse και του EMF να παράξει αυτοματοποιημένα ένα Ecore μοντέλο από ένα άλλο Ecore μοντέλο με τη χρήση ενός συνόλου κανόνων μετασχηματισμού. Παρόμοιες δυνατότητες παρέχει στον μηχανικό λογισμικού το Metadata Repository (MDR) το οποίο βασίζεται στην πλατφόρμα NetBeans της εταιρίας Sun. Το MDR[8] είναι μια υλοποίηση των προτύπων MOF, XMI και JMI. Παρέχει πρόσβαση σε μεταδεδομένα μέσω ενός συνόλου διεπαφών προγραμματισμού εφαρμογών (API). Κύρια διαφοροποίηση του MDR από το EMF είναι το γεγονός ότι το MDR υλοποιεί το πρότυπο MOF χωρίς καμία τροποποίηση του προτύπου σε αντίθεση με το EMF το οποίο υλοποιεί το Ecore το οποίο διαφοροποιείται λίγο από το MOF.

Σε αυτό το σημείο αξίζει να αναφέρουμε και την AndroMDA το οποίο αποτελεί ένα περιβάλλον πλαίσιο ανοιχτού κώδικα το οποίο υλοποιεί την μοντελοκεντρική αρχιτεκτονική στην ανάπτυξη εφαρμογών. Ο χρήστης μπορεί να σχεδιάσει ένα μοντέλο της εφαρμογής το οποίο θα είναι ανεξάρτητο από την πλατφόρμα υλοποίησης (PIM-μοντέλο) και να εισάγει αυτό το μοντέλο στην AndroMDA[9]. Η AndroMDA αναλαμβάνει να μετασχηματίσει το PIM-μοντέλο σε ένα μοντέλο συγκεκριμένης πλατφόρμας (PSM-μοντέλο) και να συνεισφέρει στην υλοποίηση της εφαρμογής με την παραγωγή κώδικα. Ο χρήστης της AndroMDA έχει τη δυνατότητα να παράξει κώδικα σε σχεδόν οποιαδήποτε γλώσσα προγραμματισμού επιθυμεί. Επίσης ο κώδικας που παράγεται από την AndroMDA

[8] *Matula M. ,NetBeans Metadata Repository., March 2003, URL:http://mdr.netbeans.org/MDR-whitepaper.pdf*
[9] *What is AndroMDA?, URL:http://galaxy.andromda.org/docs/whatisit.html*

μπορεί να χρησιμοποιηθεί και σε άλλα περιβάλλοντα πλαίσια όπως το Hibernate, το Spring και το Struts. Τέλος είναι σημαντικό να προσθέσουμε ότι στόχος της ομάδας ανάπτυξης της AndroMDA είναι η ενσωμάτωση της AndroMDA με το Eclipse.

Κατά την φάση της υλοποίησης του συστήματος τα μοντέλα που παρήχθησαν στη φάση της σχεδίασης θα πρέπει να μετατραπούν σε κώδικα. Αρχικά τα διαγράμματα που χρησιμοποιήθηκαν για την αυτόματη παραγωγή κώδικα ήταν τα διαγράμματα κλάσεων λόγω της παρόμοιας δομής των διαγραμμάτων αυτών με την δομή των αντικειμενοστραφών γλωσσών. Στη συνέχεια όμως, το ερευνητικό ενδιαφέρον στράφηκε στην αυτοματοποιημένη επεξεργασία και άλλων τύπων διαγραμμάτων και την εξαγωγή πληροφοριών και συμπερασμάτων από αυτά. Τα δεδομένα που προκύπτουν από την επεξεργασία των ποικίλων διαγραμμάτων χρησιμοποιούνται στον συγχρονισμό των διαγραμμάτων μεταξύ τους καθώς και την ανάλυση με μεγαλύτερη λεπτομέρεια της δομής, των λειτουργιών αλλά και τον κώδικα του συστήματος. Οι Blech, Glesner και Leitner[10] ασχολούνται με την μελέτη διαγραμμάτων κατάστασης και την επαλήθευση της διαδικασίας της παραγωγής Java κώδικα από τα διαγράμματα κατάστασης. Οι Sangal, Farrell, Lieberman[11] και Lorenz επεξεργάζονται ακολουθιακά διαγράμματα σε συνδυασμό με διαγράμματα κλάσεων τα οποία θεωρούν ότι είναι συνεπή μεταξύ τους. Από την επεξεργασία των διαγραμμάτων κλάσεων προκύπτουν τα βασικά στοιχεία των κλάσεων του κώδικα του συστήματος που μελετούν, όπως τα πεδία και οι μέθοδοι κάθε κλάσης, ενώ από τα ακολουθιακά διαγράμματα προκύπτουν πληροφορίες σχετικές με τις αλληλουχίες των κλήσεων των μεθόδων που πραγματοποιούνται κατά την εκτέλεση μιας λειτουργίας του συστήματος. Κατασκευάζεται επίσης και ένας σκελετός του σώματος των μεθόδων των κλάσεων του συστήματος. Οι Mathematics Thongmak και Pornsiri Muenchaisri[12] χρησιμοποιούν ακολουθιακά διαγράμματα και διαγράμματα κλάσης για να παράγουν κώδικα. Ορίζουν και χρησιμοποιούν ένα μεταμοντέλο ακολουθιακών διαγραμμάτων το οποίο βασίζεται

---

[10] *Jan Olaf Blech, Sabine Glesner, Johannes Leitner, Formal Verification of Java Code Generation from UML Models, Fujaba Days, september 2005*

[11] *Sangal, N.; Farrell, E.; Lieberherr, K.; Lorenz, D., Interaction schemata: compiling interactions to code ,in Technology of Object- Oriented Languages and Systems, 1999. TOOLS 30.Proceedings , vol., no., pp.268-277, Aug 1999 URL:http://ieeexplore.ieee.org/stamp/stamp.jsp? arnumber=787555&isnumber=17059*

[12] *Mathupayas Thongmak , Pornsiri Muenchaisri, Design of Rules for Transforming UML Sequence Diagrams into Java code, in Proceedings of the Ninth Asia-Pacific Software Engineering Conference, p.485, December 04-06, 2002 .*

στο μεταμοντέλο της UML[13]. Επίσης, ορίζουν[14] ένα πολύ πιο απλό μεταμοντέλο ακολουθιακών διαγραμμάτων από αυτό που προδιαγράφεται στη UML και περιγράφουν τον μετασχηματισμό ακολουθιακών διαγραμμάτων που υπακούν στο μεταμοντέλο της UML σε ακολουθιακά διαγράμματα που υπακούν στο δικό τους μοντέλο. Το σημαντικότερο στοιχείο της εργασίας τους αποτελεί ένα σύνολο κανόνων που αναγνωρίζουν μοτίβα ακολουθιακών διαγραμμάτων, διαγραμμάτων κλάσης ή και συνδυασμό των δύο. Το αποτέλεσμα που προκύπτει είναι κώδικας σε Java ο οποίος περιέχει τους ορισμούς των μεθόδων και των πεδίων των κλάσεων όπως προκύπτουν από τα διαγράμματα κλάσης τις αρχικοποιήσεις των πεδίων και τις λοιπές κλήσεις μεθόδων όπως προκύπτουν από τα ακολουθιακά διαγράμματα. Παρατηρούμε ότι όπως και στις προηγούμενες εργασίες η εξαγωγή της αρχιτεκτονικής του συστήματος βασίζεται στην ανάγνωση διαγραμμάτων κλάσης και τα ακολουθιακά διαγράμματα χρησιμοποιούνται για να συμπληρώσουν τα συμπεράσματα που έχουν προκύψει από τα διαγράμματα κλάσης.

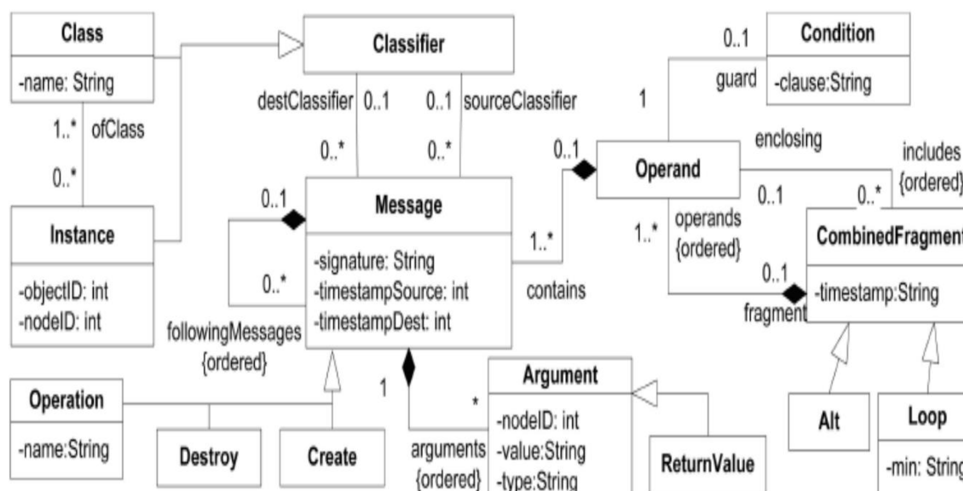
Οι Engels, Hücking, Sauer, Wagner[15] βασίζονται σε συνεργατικά διαγράμματα και διαγράμματα κλάσεων για να παράγουν κώδικα Java ο οποίος θα είναι συντακτικά ορθός. Σε αυτήν την επιστημονική εργασία αρχικά θεωρείται ότι είναι διαθέσιμα τόσο τα διαγράμματα κλάσεων όσο και τα συνεργατικά διαγράμματα και ότι τα διαγράμματα αυτά είναι συντακτικά σωστά και ότι είναι συνεπή μεταξύ τους. Η ανάλυση του συστήματος γίνεται χρησιμοποιώντας ένα σύνολο κανόνων αναγνώρισης μοτίβων στα διαγράμματα και αποτελείται από δύο βήματα. Στο πρώτο βήμα αναλύεται η δομή του συστήματος από τα διαγράμματα κλάσεων. Από την επεξεργασία των διαγραμμάτων κλάσεων παράγονται οι κλάσεις του συστήματος, οι μέθοδοι και τα πεδία της κάθε κλάσης. Επίσης καθίσταται δυνατό να προσδιοριστούν οι τύποι των δεδομένων των πεδίων των κλάσεων, των παραμέτρων των μεθόδων καθώς και των δεδομένων που επιστρέφουν οι μέθοδοι. Στο δεύτερο βήμα προστίθενται πληροφορίες που εξάγονται από τα συνεργατικά διαγράμματα. Με βάση τις πληροφορίες αυτές εξάγονται συμπεράσματα για τις αναθέσεις τιμών των πεδίων των κλάσεων, δηλαδή ποιες μέθοδοι καλούνται για τον προσδιορισμό των τιμών των πεδίων των

[13] *Object Management Group, Inc., UML Superstructure Specification, version 2.1.2, November 2007*

[14] *L.C. Briand, Y. Labiche and Y. Miao.Towards the Reverse Engineering of UML Sequence Diagrams.in Proceedings of the IEEE Working Conference in Reverse Enginnering. pp.150-169,2004*

[15] *Engels G., Hücking G., Sauer S. , Wagner A.: UML Collaboration Diagrams and Their Transformation to Java. In France R, Rumbe B. (eds) : Proc. UML '99- Beyond the Standard, Fort Collins,CO, USA, LNCS 1723, Springer, 1999 pp 473-488*

κλάσεων. Το αποτέλεσμα που παράγεται είναι κώδικας Java ο οποίος αποτελείται από τις κλάσεις, τις μεθόδους και τα πεδία που παρήχθησαν από το πρώτο βήμα της ανάλυσης και τις κλήσεις των συναρτήσεων και τις αναθέσεις τιμών που παρήχθησαν από το δεύτερο βήμα της ανάλυσης. Οι Brian, Labiche και Leduc προσπαθούν να παράγουν sequence diagrams από την ανάλυση του κώδικα της εφαρμογής καθώς και την δυναμική εκτέλεση της εφαρμογής. Συγκεκριμένα τρέχουν την εφαρμογή σύμφωνα με κάποια σενάρια χρήσης ώστε να μπορέσουν να απεικονίσουν και να μοντελοποιήσουν τη συμπεριφορά του συστήματος και την επικοινωνία μεταξύ των διαφόρων στοιχείων του συστήματος. Για την μοντελοποίηση της λειτουργίας του συστήματος προτείνουν ένα μεταμοντέλο για την αναπαράσταση των ακολουθιακών διαγραμμάτων το οποίο είναι εμπνευσμένο από το μεταμοντέλο της UML. Το μοντέλο αυτό είναι αρκετά πιο απλό και λιτό όπως φαίνεται και στο παρακάτω σχήμα. Παρά τον μινιμαλισμό του, το μοντέλο διατηρεί σε μεγάλο βαθμό την περιγραφικότητα του μεταμοντέλου των ακολουθιακών διαγραμμάτων της UML. Ονομάζουν τα διαγράμματα που υπακούν σε αυτό το μεταμοντέλο scenario diagrams για να τα διαχωρίσουν από τα ακολουθιακά διαγράμματα αλλά και για να τονίσουν ότι πρόκειται για διαγράμματα που απεικονίζουν την επικοινωνία μεταξύ των διαφόρων μερών του συστήματος όπως αυτή καταγράφηκε από την εκτέλεση ενός συγκεκριμένου σεναρίου και όχι για την συνολική απεικόνιση της επικοινωνίας των λειτουργικών μερών του συστήματος γενικότερα. Μετά την εκτέλεση των διαφόρων σεναρίων και την ανάλυση του κώδικα του συστήματος συγχωνεύουν τα scenario diagrams για να προκύψει ένα τελικό scenario diagram το οποίο θα δίνει μια όσο το δυνατό πληρέστερη μοντελοποίηση της επικοινωνίας των μονάδων του συστήματος καθώς και της λειτουργίας του. Το τελικό μοντέλο μπορεί να συγκριθεί με τα ακολουθιακά διαγράμματα τω προδιαγραφών του συστήματος.



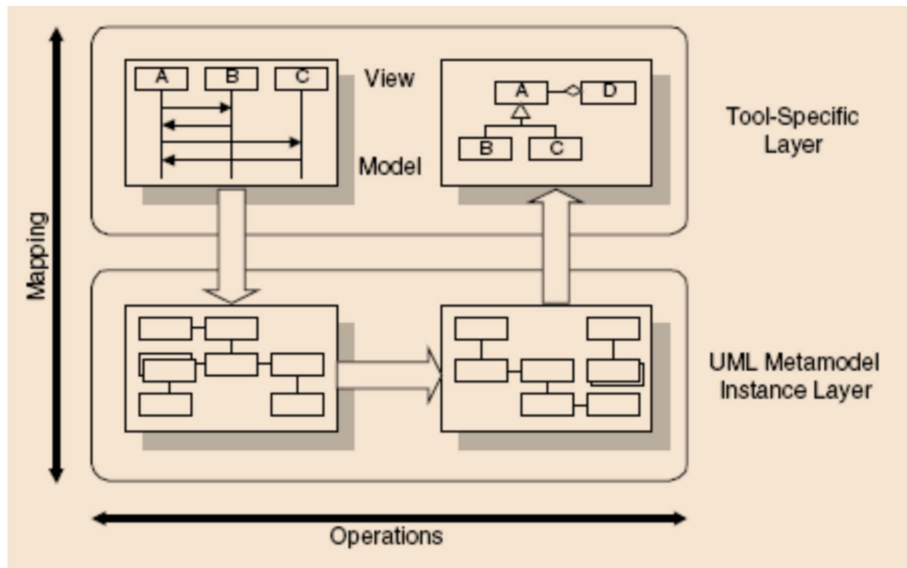
<u>Το scenario diagram μεταμοντέλο της επιστημονικής εργασίας των Brian,Labiche και Leduc</u>

Σημαντική προσπάθεια έχει καταβληθεί στον μετασχηματισμό διαγραμμάτων από τον έναν τύπο στον άλλο, στον συγχρονισμό των διαγραμμάτων ώστε να είναι μεταξύ τους συνεπή, καθώς επίσης και στον συγχρονισμό των διαγραμμάτων με τα μεταμοντέλα τους.

Οι Selonen, Koskimies και Sakkinen[16] ασχολούνται με τον μετασχηματισμό UML διαγραμμάτων σε UML διαγράμματα άλλου τύπου. Αναφέρουν ότι τα UML διαγράμματα αποτελούν ένα σύνολο διαγραμμάτων τα οποία απεικονίζουν ένα σύστημα από διαφορετικές πτυχές και συνεπώς τα διάφορα μοντέλα του συστήματος είναι εξαρτώμενα μεταξύ τους και επικαλυπτόμενα. Βασιζόμενοι σε αυτό το γεγονός προτείνουν ένα σύνολο μετασχηματισμών πάνω στα μοντέλα ενός συστήματος. Οι μετασχηματισμοί αυτοί είναι ο έλεγχος συνέπειας μεταξύ των διαγραμμάτων, ο εμπλουτισμός της πληροφορίας που περιέχει ένα διάγραμμα με τη χρήση άλλων διαγραμμάτων, η προβολή μιας πτυχής μόνο του συστήματος και τέλος η δημιουργία ενός διαγράμματος με βάση τις πληροφορίες που προκύπτουν από τα διαγράμματα άλλου τύπου. Η τεχνική που χρησιμοποιούν στηρίζεται σε ένα σύνολο κανόνων μετασχηματισμών μοντέλων. Οι κανόνες αυτοί βασίζονται στην ύπαρξη μεταμοντέλων τα οποία προδιαγράφουν τη δομή των UML διαγραμμάτων και περιγράφουν τη μετάβαση από ένα στιγμιότυπο ενός

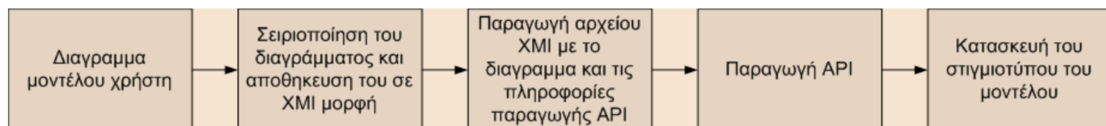
[16]*P. Selonen, K. Koskimies, and M. Sakkinen, How to make apples from oranges in UML, hicss, vol. 03, p. 3054, 2001.*

μεταμοντέλου σε ένα στιγμιότυπο ενός άλλου μεταμοντέλου. Ο τρόπος λειτουργίας των κανόνων μετασχηματισμού απεικονίζεται γραφικά και στο παρακάτω σχήμα.



Μετασχηματισμός μεταμοντέλων

Το Eclipse με τη χρήση του EMF δίνει τη δυνατότητα στον χρήστη να ορίσει μοντέλα τα οποία υπακούν στο μεταμοντέλο του EMF, το Ecore. Με τη χρήση των εργαλείων που προσφέρει το Eclipse, ο σχεδιαστής έχει τη δυνατότητα, χρησιμοποιώντας το γραφικό περιβάλλον του Eclipse, να σχεδιάσει το μοντέλο του όπως θα το σχεδίαζε και στο χέρι ή με οποιοδήποτε άλλο σχεδιαστικό εργαλείο. Μέσω μιας αυτοματοποιημένης διαδικασίας, με αρχική πηγή πληροφοριών μόνο το γράφημα του μοντέλου, μπορεί να παράξει μια ολοκληρωμένη διεπαφή προγραμματισμού εφαρμογών (API) για την παραγωγή στιγμιοτύπων του μοντέλου καθώς και μια λειτουργική εφαρμογή παραγωγής στιγμιοτύπων.



Η διαδικασία ορισμού μοντέλων

Όπως φαίνεται και στο παραπάνω σχήμα ο σχεδιαστής δημιουργεί ένα αρχείο ecorediag το οποίο περιέχει την γραφική απεικόνιση του μοντέλου του σχεδιαστή. Καθώς ο χρήστης σχεδιάζει το διάγραμμα του μοντέλου του, το Eclipse παράγει ταυτόχρονα το αρχείο .ecore το οποίο είναι το μοντέλο του συστήματος

σειριοποιημένο σε ένα αρχείο. Το αρχείο ecore είναι απόλυτα συμβατο με το πρότυπο XMI της OMG. Χρησιμοποιώντας τη βιβλιοθήκη εργαλείων του EMF μπορεί ο σχεδιαστής να παράξει ένα αρχείο .genmodel, το οποίο είναι και αυτό συμβατό με το πρότυπο XMI και περιέχει όλες τις πληροφορίες που περιέχει το αρχείο ecore, έχει όμως επιπρόσθετα, πληροφορίες για την παραγωγή του API. Από το αρχείο αυτό, παράγεται με τη βοήθεια της βιβλιοθήκης εργαλείων του EMF το API του μοντέλου του σχεδιαστή. Ο σχεδιαστής μετά την παραγωγή του API, έχει τη δυνατότητα να παράξει στιγμιότυπα του μοντέλου του, τα οποία αποθηκεύονται αυτόματα σε σειριοποιημένη μορφή σε αρχεια XMI. Τα στιγμιότυπα του μοντέλου, μπορούν να παραχθούν είτε προγραμματιστικά με τη χρήση του API, είτε με την χρήση του editor που κατασκευάζει το Eclipse, αποκλειστικά για τα στιγμιότυπα του μοντέλου που όρισε ο σχεδιαστής. Με αυτόν τον τρόπο ο σχεδιαστής λογισμικού από ένα απλό διάγραμμα του μοντέλου του παράγει ένα εργαλείο, το οποίο του επιτρέπει να υλοποιήσει το μοντέλο του χωρίς να χρειαστεί να εμπλακεί με τη διαχείριση των αρχείων XMI.

Τα μοντέλα αυτά έχουν γνωρίσει πολύ μεγάλη αποδοχή, παρόλη τη σχετικά σύντομη διάρκεια ζωής τους. Αυτό οφείλεται κυρίως στο ότι τα αποτελέσματα της ακαδημαϊκής έρευνας αλλά και η εφαρμογή τους στην πράξη απέδειξαν, πώς η συστηματική χρήση τους οδηγεί στην ανάπτυξη καλά δομημένου, συντηρήσιμου και επαναχρησιμοποιήσιμου λογισμικού. Ας δούμε λοιπόν ένα ένα και αναλυτικότερα τα μοντέλα αυτά.

2.2 Πρότυπα Σχεδίασης

Το πρότυπο σχεδίασης "Προσαρμογέας" (Adapter) έχει ως στόχο τη μετατροπή της διασύνδεσης μιας κλάσης σε μια άλλη που αναμένει το πρόγραμμα πελάτης. Ο προσαρμογέας επιτρέπει τη συνεργασία κλάσεων, η οποία σε διαφορετική περίπτωση θα ήταν αδύνατη λόγω ασύμβατων διασυνδέσεων. Συχνά, ο κώδικας μιας κλάσης προσφέρεται για επαναχρησιμοποίηση, αλλά αυτή δεν είναι δυνατή λόγω του ότι τα προγράμματα που επιθυμούν να χρησιμοποιήσουν τις λειτουργίες της, αναμένουν διαφορετική διασύνδεση. Έστω για παράδειγμα ότι μια κλάση σχεδίασης είναι σε θέση να σχεδιάσει γραμμές, αλλά απαιτεί ως παραμέτρους τις συντεταγμένες στη μορφή (x1, y1, x2, y2), ενώ τα προγράμματα

πελάτες είναι σε θέση να παρέχουν τις συντεταγμένες στη μορφή (x1, x2, y1, y2). Στη συνήθη περίπτωση όπου τα προγράμματα πελάτες δεν είναι δυνατόν να τροποποιηθούν (καθώς βρίσκονται ήδη εγκατεστημένα σε διάφορα πεδία εφαρμογών), ενώ και η κλάση σχεδίασης είναι επιθυμητό να χρησιμοποιηθεί χωρίς τροποποίηση (καθώς οποιαδήποτε επέμβαση στον κώδικα μιας μεθόδου είναι δυνατόν να προκαλέσει σφάλματα στις υπόλοιπες μεθόδους), βρίσκει εφαρμογή το πρότυπο "Προσαρμογέας".

Το πρότυπο σχεδίασης "Σύνθετο" (Composite) επιτρέπει τη σύνθεση αντικειμένων σε δενδροειδείς δομές για την αναπαράσταση ιεραρχιών τμήματος-όλου. Το μοντέλο-πρότυπο σχεδίασης "Σύνθετο" επιτρέπει στα προγράμματα πελάτες να διαχειρίζονται με ενιαίο τρόπο τόσο τα ανεξάρτητα αντικείμενα όσο και συνθέσεις αντικειμένων. Πολύ συχνά, σε μια εφαρμογή, εκτός από μεμονωμένα αντικείμενα (π.χ. Τροχός, Μηχανή, Κάθισμα), υφίστανται και σύνθετα αντικείμενα που περιέχουν ή περιλαμβάνουν άλλα αντικείμενα (π.χ. Αυτοκίνητο). Η σχέση περιεκτικότητας μπορεί να είναι ασθενούς μορφής (συσσωμάτωση) είτε ισχυρής μορφής (σύνθεση). Η συνήθης αντιμετώπιση μιας τέτοιας περίπτωσης με τη χρήση μιας σχέσης περιεκτικότητας μεταξύ της κλάσης που αντιπροσωπεύει το όλον (περικλείουσα κλάση) και των κλάσεων που αντιπροσωπεύουν τα τμήματα, έχει το μειονέκτημα ότι δεν επιτρέπει τον ομοιόμορφο χειρισμό των αντικειμένων από ένα πρόγραμμα πελάτη. Για παράδειγμα, μια άλλη εφαρμογή, θα πρέπει να διατηρεί δείκτες και προς αντικείμενα τύπου Τροχός, Μηχανή, Κάθισμα, αλλά και δείκτες προς αντικείμενα τύπου Αυτοκίνητο. Λαμβάνοντας υπόψη τους πρωταρχικούς στόχους του αντικειμενοστρεφούς προγραμματισμού, αν προστεθεί στο σύστημα μια νέα σύνθετη κλάση (π.χ. Λεωφορείο), τότε ο κώδικας του προγράμματος πελάτη, θα πρέπει να τροποποιηθεί για να είναι δυνατός ο χειρισμός των νέων αντικειμένων τύπου Λεωφορείο. Η αντιμετώπιση αυτού του προβλήματος, επιτυγχάνεται με κομψό τρόπο με το πρότυπο σχεδίασης "Σύνθετο".

Το πρότυπο σχεδίασης "Μοναδιαίο" (Singleton) εξασφαλίζει ότι μια κλάση θα έχει μόνο ένα στιγμιότυπο και παρέχει ένα καθολικό σημείο πρόσβασης σε αυτό. Συνήθως μεταξύ κλάσεων και στιγμιοτύπων τους υπάρχει μια σχέση ένα-προς-πολλά. Κατά τη διαδικασία ανάλυσης, η ύπαρξη πολλών στιγμιότυπων της ίδιας έννοιας στο σύστημα υποδηλώνει την αναγκαιότητα μιας κλάσης. Τα αντικείμενα δημιουργούνται δεσμεύοντας χώρο στη μνήμη όποτε κρίνεται σκόπιμο

και διαγράφονται από τη μνήμη όταν τερματιστεί η χρήση τους. Ορισμένες όμως φορές, απαιτείται η ύπαρξη κλάσεων από τις οποίες παράγεται ένα μόνο αντικείμενο. Πολύ συχνά, το αντικείμενο αυτό συνήθως δημιουργείται κατά την έναρξη της εφαρμογής και διαγράφεται με το πέρας της. Ο ρόλος του μοναδικού αυτού αντικειμένου είναι η διαχείριση των υπολοίπων αντικειμένων της εφαρμογής και για το λόγο αυτό, αποτελεί λογικό σφάλμα να δημιουργηθούν περισσότερα του ενός τέτοια αντικείμενα-διαχειριστές (managers ή controllers). Σε μια τέτοια περίπτωση, η εφαρμογή θα έχει περισσότερα του ενός σημεία εκκίνησης, και ο υποθετικός χρήστης, ανάλογα με το σημείο εκκίνησης, μπορεί να καταλήξει να χρησιμοποιεί ένα υποσύνολο των αντικειμένων του συστήματος. Επιπλέον, αν υπάρχουν περισσότεροι του ενός διαχειριστές, ενώ ο επιθυμητός στόχος είναι η ακολουθιακή εκτέλεση δραστηριοτήτων, πολλές δραστηριότητες θα εκτελούνται παράλληλα. Το πρότυπο σχεδίασης "Μοναδιαίο", εξασφαλίζει τη δημιουργία ενός και μόνο αντικειμένου, περιλαμβάνοντας μια ειδική μέθοδο κατασκευής στιγμιοτύπων:

-Όταν καλείται αυτή η μέθοδος, ελέγχει αν κάποιο αντικείμενο έχει ήδη δημιουργηθεί. Αν ναι, η μέθοδος επιστρέφει απλώς έναν δείκτη προς το υπάρχον αντικείμενο. Αν όχι, η μέθοδος δημιουργεί ένα νέο αντικείμενο και επιστρέφει δείκτη προς αυτό.

-Για να εξασφαλισθεί ότι αυτός είναι ο μοναδικός τρόπος δημιουργίας αντικειμένων από αυτή την κλάση, ο κατασκευαστής της κλάσης δηλώνεται ως προστατευμένος (protected) ή ιδιωτικός (private). Με τον τρόπο αυτό, δεν είναι δυνατόν να δημιουργηθεί ένα αντικείμενο παρακάμπτοντας την παραπάνω ειδική μέθοδο.

Το πρότυπο σχεδίασης "Γέφυρα" (Bridge Pattern) έχει ως στόχο την αποσύνδεση μιας αφαίρεσης από την υλοποίησή της, ώστε να μπορούν να μεταβάλλονται ανεξάρτητα. Όταν μια αφαίρεση μπορεί να έχει περισσότερες από μία υλοποιήσεις, ο συνήθης τρόπος οργάνωσης είναι με τη χρήση κληρονομικότητας. Με τον όρο αφαίρεση νοείται μια αφηρημένη κλάση που ορίζει μια διασύνδεση (ένα σύνολο υπογραφών), ενώ υλοποιήσεις είναι οι συγκεκριμένες παράγωγες κλάσεις οι οποίες υλοποιούν τις μεθόδους της αφηρημένης κλάσεις. Η προσέγγιση αυτή ωστόσο, συνδέει με μόνιμο τρόπο την αφαίρεση και τις υλοποιήσεις, καθιστώντας δύσκολη την επέκταση, τροποποίηση και επαναχρησιμοποίηση αφαιρέσεων και υλοποιήσεων ανεξάρτητα. Το πρότυπο

"Γέφυρα" είναι σχετικά δύσκολο στην κατανόησή του. Χρησιμοποιείται ωστόσο σε πληθώρα περιπτώσεων όπου εντοπίζονται:

-Μεταβολές στην αφαίρεση μιας έννοιας

-Μεταβολές στον τρόπο υλοποίησης της έννοιας αυτής

Η Γέφυρα αντίκειται στη συνήθη τάση χειρισμού αντίστοιχων καταστάσεων μόνο με κληρονομικότητα. Ικανοποιεί όμως δύο από τους βασικούς κανόνες της αντικειμενοστρεφούς κοινότητας: «Εντοπίστε αυτό που μεταβάλλεται και ενσωματώστε το», και «προτιμήστε τη σύνθεση αντικειμένων από την κληρονομικότητα κλάσεων».

Το πρότυπο σχεδίασης «Παρατηρητής» (Observer) είναι επίσης γνωστό ως πρότυπο «Δημοσίευση-Εγγραφή» (Publish-Subscribe). Ορίζει μια σχέση εξάρτησης ένα-προς-πολλά μεταξύ αντικειμένων έτσι ώστε όταν μεταβάλλεται η κατάσταση ενός αντικειμένου, όλα τα εξαρτώμενα αντικείμενα να ενημερώνονται και τροποποιούνται αυτόματα. Καθώς ο στόχος της αντικειμενοστρεφούς σχεδίασης είναι η δημιουργία ενός συνόλου αλληλεπιδρώντων αντικειμένων, ένα από τα συχνά προβλήματα είναι η αναγκαιότητα συνεργασίας μεταξύ κλάσεων, γεγονός που οδηγεί σε υψηλή σύζευξη. Ορισμένα από τα πρότυπα σχεδίασης, με χαρακτηριστικότερο το πρότυπο «Παρατηρητής», επιδιώκουν να μειώσουν τη σύζευξη μεταξύ των αντικειμένων, παρέχοντας αυξημένη δυνατότητα επαναχρησιμοποίησης και τροποποίησης του συστήματος. Το συγκεκριμένο πρότυπο, επιτρέπει την αυτόματη ειδοποίηση και ενημέρωση ενός συνόλου αντικειμένων τα οποία «αναμένουν» ένα γεγονός, που εκδηλώνεται ως αλλαγή στην κατάσταση ενός αντικειμένου. Ο στόχος είναι η από-σύζευξη των παρατηρητών από το παρακολουθούμενο αντικείμενο (υποκείμενο), έτσι ώστε κάθε φορά που προστίθεται ένας νέος παρατηρητής (με διαφορετική διασύνδεση ενδεχομένως), να μην απαιτούνται αλλαγές στο παρακολουθούμενο αντικείμενο. Το συγκεκριμένο πρότυπο είναι από τα πλέον ευρέως χρησιμοποιούμενα και υλοποιείται με σχετική ευκολία σε διάφορες γλώσσες προγραμματισμού. Η εφαρμογή του προτύπου προϋποθέτει τον εντοπισμό των εξής δύο τμημάτων: ενός υποκειμένου και του παρατηρητή. Μεταξύ των δύο υφίσταται μια συσχέτιση ένα-προς-πολλά. Το υποκείμενο θεωρείται ότι διατηρεί το μοντέλο των δεδομένων και η λειτουργικότητα που αφορά στην παρατήρηση των δεδομένων κατανέμεται σε διακριτά αντικείμενα -παρατηρητές. Οι παρατηρητές καταχωρούνται στο υποκείμενο κατά τη δημιουργία τους. Οποτεδήποτε το υποκείμενο αλλάζει,

«ανακοινώνει» προς όλους τους καταχωρημένους παρατηρητές το γεγονός της αλλαγής, και κάθε παρατηρητής ερωτά το υποκείμενο για το υποσύνολο της κατάστασης του υποκειμένου που το ενδιαφέρει. Στο ανωτέρω πρωτόκολλο επικοινωνίας η πληροφορία «αντλείται», αντί να αποστέλλεται στους παρατηρητές. Το πρότυπο «Παρατηρητής» εφαρμόζεται για χρόνια στην ευρέως γνωστή αρχιτεκτονική Μοντέλου-Όψης-Έλεγκτή (Model-View-Controller).

Το πρότυπο σχεδίασης «Επισκέπτης» (Visitor Pattern) έχει ως στόχο την αναπαράσταση μιας λειτουργίας που πρόκειται να πραγματοποιηθεί στα στοιχεία μιας δομής αντικειμένων. Το πρότυπο επιτρέπει τον ορισμό μιας νέας λειτουργίας χωρίς την τροποποίηση των κλάσεων των στοιχείων στα οποία επιδρά. Πολύ συχνά απαιτείται η προσθήκη μιας νέας μεθόδου σε μια υπάρχουσα ιεραρχία κλάσεων αλλά είναι εξαιρετικά δύσκολο να τροποποιηθούν οι ίδιες οι κλάσεις της ιεραρχίας. Η προσέγγιση αυτή ενθαρρύνει τη σχεδίαση ιεραρχιών από στοιχεία «ελαφρού τύπου» καθώς οι αντίστοιχες κλάσεις έχουν περιορισμένες αρμοδιότητες. Νέα λειτουργικότητα μπορεί εύκολα να προστεθεί στην αρχική ιεραρχία, χωρίς την τροποποίηση των ίδιων των κλάσεων, μόνο με τη δημιουργία μιας νέας υποκλάσης στο πρότυπο «Επισκέπτης». Το πρότυπο «Επισκέπτης» έχει αρκετά μεγάλη πολυπλοκότητα στη λειτουργία του καθώς υλοποιεί τη λεγόμενη «διπλή αποστολή» (double dispatch ή dual dispatch). Τα μηνύματα στον αντικειμενοστρεφή προγραμματισμό κατά κανόνα επιδεικνύουν συμπεριφορά που αντιστοιχεί στην «απλή αποστολή» - η λειτουργία που εκτελείται εξαρτάται από το όνομα της αίτησης και τον τύπο του (μοναδικού) αποδέκτη. Στη «διπλή αποστολή» η λειτουργία που εκτελείται εξαρτάται από το όνομα της αίτησης και τον τύπο δύο αποδεκτών (στο συγκεκριμένο πρότυπο από τον τύπο του «Επισκέπτη» και τον τύπο του στοιχείου που επισκέπτεται).

Ο σκοπός του προτύπου σχεδίασης "Αφηρημένο Εργοστάσιο" (Abstract Factory) είναι η παροχή μιας διασύνδεσης για τη δημιουργία οικογενειών συσχετιζόμενων ή εξαρτημένων αντικειμένων χωρίς να προσδιορίζεται η συγκεκριμένη κλάση τους. Όλες οι αντικειμενοστρεφείς γλώσσες προγραμματισμού έχουν κάποιο προγραμματιστικό ιδίωμα για τη δημιουργία νέων αντικειμένων (π.χ. τον τελεστή new). Τα πρότυπα σχεδίασης που ανήκουν στην κατηγορία των κατασκευαστικών προτύπων (Creational Patterns) επιτρέπουν τη συγγραφή μεθόδων που δημιουργούν νέα αντικείμενα, χωρίς την άμεση χρήση των συγκεκριμένων ιδιωμάτων. Το γεγονός αυτό επιτρέπει να αναπτυχθούν

μέθοδοι κλάσεων που παράγουν ομάδες διαφορετικών αντικειμένων καθώς και την επέκτασή τους για νέα αντικείμενα, χωρίς την τροποποίηση του κώδικα των μεθόδων! Σε αντίθετη περίπτωση, η δημιουργία ενός διαφορετικού αντικειμένου για κάθε περίπτωση, θα απαιτούσε τη χρήση ελέγχων if/else ή εντολών switch, στοιχεία που υποδηλώνουν κακή εφαρμογή των αρχών του αντικειμενοστρεφούς προγραμματισμού, καθώς δυσχεραίνουν τη συντήρηση του λογισμικού.

Το πρότυπο σχεδίασης "Στρατηγική" (Strategy) ορίζει μια οικογένεια αλγορίθμων, τους ενσωματώνει και επιτρέπει την εναλλαγή μεταξύ αυτών. Το πρότυπο δίνει τη δυνατότητα μεταβολής των αλγορίθμων, ανεξάρτητα από τους πελάτες που τους χρησιμοποιούν. Το συγκεκριμένο πρότυπο σχετίζεται άμεσα με την αναγκαιότητα συχνής τροποποίησης του λογισμικού και είναι ίσως το πλέον ευρέως χρησιμοποιούμενο, ανεξαρτήτως του αν η εφαρμογή του γίνεται αντιληπτή από τον σχεδιαστή. Αναγνωρίζει ότι σε πολλές εφαρμογές, υπάρχει μια γενική φιλοσοφία-στρατηγική που είναι κοινή σε πολλές περιπτώσεις, η συγκεκριμένη υλοποίηση όμως κάθε περίπτωσης είναι διαφορετική. Ουσιαστικά, υπάρχει ένας κοινός γενικός αλγόριθμος (π.χ. εύρεση ελαχίστου, μεγίστου και διαμέσου με ταξινόμηση), η λεπτομερής όμως υλοποίηση του είναι διαφορετική σε κάθε περίπτωση (π.χ. χρήση ταξινόμησης φυσαλίδας ή ταξινόμησης με επιλογή). Στον αντικειμενοστρεφή προγραμματισμό, τα ανωτέρω χαρακτηριστικά παραπέμπουν συνήθως στην κληρονομικότητα. Το πρότυπο «Στρατηγική» αποτελεί την αντιμετώπιση του προβλήματος στηριζόμενο όχι μόνο στην κληρονομικότητα, αλλά και στη σύνθεση αντικειμένων. Το πρότυπο εφαρμόζει την αρχή της Αντιστροφής των Εξαρτήσεων αναγκάζοντας και τον γενικό αλγόριθμο αλλά και τις λεπτομερείς υλοποιήσεις να εξαρτώνται από αφαιρέσεις.

Το πρότυπο σχεδίασης «Μέθοδος Υπόδειγμα» (Template Method) ορίζει το περίγραμμα ενός αλγορίθμου σε μια λειτουργία, αφήνοντας ορισμένα βήματα στις παράγωγες κλάσεις. Το πρότυπο επιτρέπει στις παράγωγες κλάσεις να επαναορίσουν ορισμένα βήματα του αλγορίθμου χωρίς να αλλάξουν τη δομή του. Το πρότυπο επιλύει ένα παρόμοιο πρόβλημα με αυτό που συζητήθηκε στο πρότυπο «Στρατηγική». Στόχος είναι ο διαχωρισμός ενός γενικού αλγορίθμου από συγκεκριμένες υλοποιήσεις. Ωστόσο, ενώ το πρότυπο «Στρατηγική» αξιοποιεί τη δυνατότητα μεταφοράς μιας αρμοδιότητας σε ένα άλλο αντικείμενο μέσω της διαβίβασης μηνυμάτων (delegation), το πρότυπο «Μέθοδος Υπόδειγμα» εκμεταλλεύεται το μηχανισμό της κληρονομικότητας. Σημειώνεται ότι και αυτό το

πρότυπο εφαρμόζεται πολύ συχνά από τους σχεδιαστές αντικειμενοστρεφούς λογισμικού, ακόμα και αν δεν γίνεται αντιληπτό ως ξεχωριστή τεχνική.

3. Έλεγχος ποιότητας μοντέλου (QDD)

Σκοπός του κεφαλαίου αυτού είναι η ανάλυση της έννοιας της ποιότητας λογισμικού, με την περιγραφή των πιο γνωστών μοντέλων ποιότητας και την ανάλυσή της σε επιμέρους χαρακτηριστικά. Επίσης, παρουσιάζεται συνοπτικά το σύστημα ποιότητας λογισμικού και η χρήση και συνεισφορά του στην ανάπτυξη λογισμικού.

Η έννοια της ποιότητας του λογισμικού οφείλει να χωριστεί από την συγγραφέα σε δύο τμήματα ώστε να γίνει πιο κατανοητή στον μέσο αναγνώστη. Την εξωτερική ποιότητα και την εσωτερική. Η εξωτερική βρίσκεται κοντά στα ερεθίσματα του χρήστη του λογισμικού και είναι απλή. Είναι εύκολη στην δοκιμή αλλά και στην εύρεση των σφαλμάτων και των λαθών. Το δύσκολο στην εξωτερική ποιότητα είναι η μέτρηση αυτής καθώς είναι απίθανο πολλοί χρήστες να συμφωνήσουν στην μια και μοναδική «τέλεια» σχεδίαση. Την εσωτερική είναι δύσκολο για τον απλό χρήστη να την κατανοήσει. Το βασικό της χαρακτηριστικό είναι η ποσοτική αναπαράσταση που δίνετε μέσω κάποιον μετρικών μοντέλων. Αποτελεί ουσιαστικά την πραγματική ποιότητα του λογισμικού καθώς βασίζεται στην σαφήνεια, ευελιξία, αναγνωσιμότητα και συντηρησιμότητα καθώς και άλλων που θα δούμε παρακάτω .

Ακριβώς επειδή όλα όμως αυτά μπερδεύουν οδηγούν στο συμπέρασμα ότι η έννοια της ποιότητας λογισμικού είναι αρκετά αφηρημένη και δεν επιτρέπει τον καθορισμό εύκολα μετρήσιμων-απτών στόχων, προέκυψε η ανάγκη επιμερισμού της ποιότητας σε χαρακτηριστικά τα οποία θα συνθέτουν αυτή την αφηρημένη έννοια «ποιότητα». Αυτά τα χαρακτηριστικά ονομάζονται παράγοντες ποιότητας (quality factors). Η διαδικασία διάσπασης της ποιότητας σε παράγοντες ποιότητας και του εντοπισμού των ποιοτικών χαρακτηριστικών των οποίων η διασφάλιση είναι σημαντική για το εκάστοτε έργο (με στόχο τη διενέργεια μετρήσεων για τον έλεγχο αυτών των χαρακτηριστικών), είναι σήμερα βασική διαδικασία κάθε προγράμματος ποιότητας λογισμικού. Το σκεπτικό αυτό παρουσιάστηκε περίπου την ίδια χρονική περίοδο -τέλη της δεκαετίας του ’80- τόσο από τον McCall[17] όσο και τον Boehm[18]. Βασισμένο σε αυτά τα δύο μοντέλα -αρκετά χρόνια μετά

---

[17] *J. A. McCall et al., «Factors in Software Quality, Vols I, II, III», US Rome Air Development Center Reports NTIS AD/A – 049 014, 015, 055, (1977)*
[18] *B. Boehm et al., «Characteristics of Software Quality», North Holland, (1978)*

δημιουργήθηκε το πρότυπο ISO 9126, το αποτέλεσμα μιας διεθνούς προσπάθειας να αναπτυχθεί ένα πρότυπο για τη μέτρηση της ποιότητας λογισμικού[19] .

Υπάρχουν πολλές προσεγγίσεις στην έννοια της ποιότητας. Ο Garvin[20] ορίζει την ποιότητα ως μια πολύπλοκη και πολυπρόσωπη έννοια που μπορεί να περιγραφεί με πέντε διαφορετικές θεωρήσεις:

• Εμπειρική θεώρηση (transcendental view): Θεωρεί ότι η ποιότητα είναι κάτι που μπορεί να αναγνωριστεί εμπειρικά, αλλά όχι να οριστεί ούτε να επιτευχθεί πλήρως.
• Θεώρηση από την πλευρά του χρήστη (user view): Αντιμετωπίζει την ποιότητα ως καταλληλότητα για χρήση.
• Κατασκευαστική θεώρηση (manufacturing view): Αντιμετωπίζει την ποιότητα ως ικανοποίηση των κατασκευαστικών προδιαγραφών του χρήστη.
• Θεώρηση προϊόντος (product view): Θεωρεί ότι η ποιότητα ταυτίζεται με τα ενδογενή (εσωτερικά) χαρακτηριστικά του προϊόντος.
• Θεώρηση βάσει της αξίας (value - based view): Θεωρεί ότι η ποιότητα εξαρτάται από το ποσό που διατίθεται να πληρώσει ο χρήστης για το προϊόν.

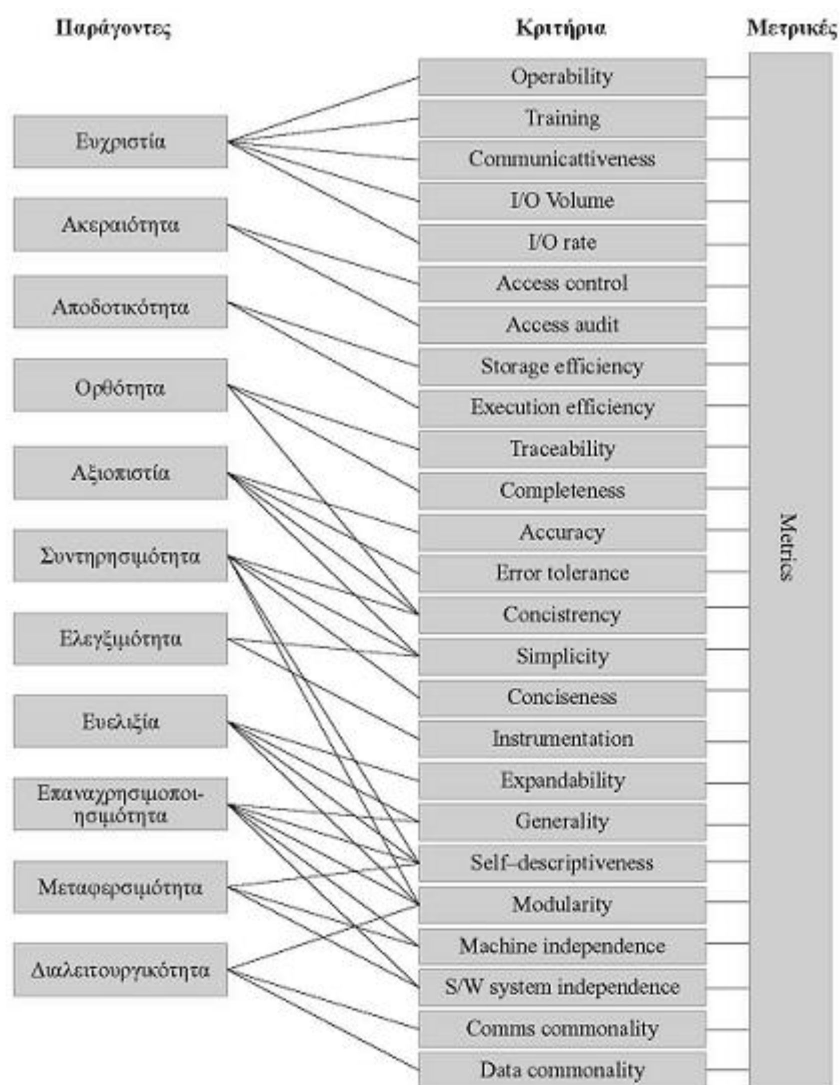
Ανάλογα με τη θεώρηση της ποιότητας, προκύπτουν και αντίστοιχοι παράγοντες ποιότητας που συνεισφέρουν σε αυτό που αφηρημένα καλούμε «ποιότητα του προϊόντος». Αυτοί οι παράγοντες μπορεί να διαφέρουν για την επιχείρηση που αναπτύσσει το λογισμικό (θεώρηση του κατασκευαστή) σε σχέση με αυτούς που ενδιαφέρουν τους τελικούς χρήστες. Ακόμα, μπορεί να βασίζονται σε πολιτισμικές ή κοινωνικές αντιλήψεις για την ποιότητα. Οι τελικοί χρήστες ενδιαφέρονται κυρίως για παράγοντες όπως η λειτουργικότητα (functionality) και η ευχρηστία (usability). Ακριβώς επειδή στον ορισμό της λειτουργικότητας μιλάμε και για «συνεπαγόμενες ανάγκες», είναι σαφές ότι ο παράγοντας αυτός σχετίζεται και με τις κοινωνικές αντιλήψεις για την ποιότητα. Επίσης, για τους τελικούς χρήστες είναι σημαντικό, πέρα από τις λειτουργίες που επιτελεί το λογισμικό, η χρήση του να είναι εύκολη και κατανοητή από αυτούς. Αναφορικά με την ομάδα υλοποίησης, το ενδιαφέρον εντοπίζεται σε παράγοντες ποιότητας όπως η συντηρησιμότητα (maintainability), η ελεγξιμότητα (testability), η επαναχρησιμοποιησιμότητα (reusability) και η μεταφερσιμότητα (portability). Η ομάδα ανάπτυξης ενδιαφέρεται να μπορεί εύκολα να υλοποιεί αλλαγές στο λογισμικό. Για να μπορεί η ομάδα

[19] *ISO, «Information technology – Evaluation of software – Quality characteristics and guides for their use», International Standard, ISO/IEC 9126: (1991)*
[20] *David Garvin, «What Does Product Quality Really Mean?», Sloan Management Review, Fall, (1984)*

ανάπτυξης να υλοποιεί αλλαγές, χρειάζεται να μπορεί να ελέγχει το λογισμικό για λάθη και παραλείψεις, αλλά και να μπορεί να επαναχρησιμοποιεί τμήματα του λογισμικού που αναπτύχθηκε για ένα έργο σε κάποιο άλλο έργο. Τέλος, η ομάδα ανάπτυξης επιθυμεί να μπορεί να μεταφέρει το λογισμικό που αναπτύχθηκε για ένα έργο εύκολα σε διαφορετικές πλατφόρμες υλικού ή λειτουργικών συστημάτων. Για την κοινωνία, όπως εύκολα μπορούμε να παρατηρήσουμε, δε νοείται το λογισμικό να μην είναι αξιόπιστο και αποτελεσματικό. Αυτοί οι παράγοντες ποιότητας, δηλαδή η αξιοπιστία (reliability) και η αποδοτικότητα (efficiency) είναι και αυτοί που συνήθως υπονοούνται στις προδιαγραφές των μικρών έργων. Φυσικά σε εξειδικευμένα έργα όπου η αξιοπιστία ή η αποδοτικότητα είναι καθοριστικοί παράγοντες, τότε αυτοί παύουν να αφορούν κοινωνικές αντιλήψεις και μεταφέρονται στις απαιτήσεις του χρήστη. Για παράδειγμα σε μία εφαρμογή πραγματικού χρόνου, η αξιοπιστία και η αποδοτικότητα θα αντιμετωπιστούν τελείως διαφορετικά από μία εφαρμογή αυτοματισμού γραφείου.

Ο McCall προτείνει την τμηματοποίηση της ποιότητας σε παράγοντες (factors) ποιότητας. Επειδή τόσο η ίδια η ποιότητα, όσο και οι ποιοτικοί παράγοντες είναι εξαιρετικά αφηρημένες έννοιες, ο McCall πρότεινε επίσης την τμηματοποίηση των παραγόντων σε κριτήρια (criteria) που βρίσκονται σε χαμηλότερο επίπεδο αφαίρεσης και τα οποία μπορούν να μετρηθούν άμεσα με μετρικές (metrics). Αν και για τις μετρικές θα μιλήσουμε στο επόμενο κεφάλαιο, στο σημείο αυτό πρέπει να αναφερθεί ότι ο McCall είχε προτείνει οι μετρήσεις για κάθε κριτήριο να προκύπτουν από απαντήσεις σε ερωτήσεις για το κριτήριο. Από τα ονόματα των τριών επιπέδων αφαίρεσης το μοντέλο αυτό ονομάστηκε μοντέλο FCM (Factors – Criteria – Metrics).

Όπως φαίνεται και στο παρακάτω σχήμα, ο McCall πρότεινε 11 παράγοντες ποιότητας, 25 κριτήρια και 41 μετρικές. Οι 11 παράγοντες είναι οι: ευχρηστία (usability), ακεραιότητα (integrity), αποδοτικότητα (efficiency), ορθότητα (correctness), αξιοπιστία (reliability), συντηρησιμότητα (maintainability), ελεγξιμότητα (testability), ευελιξία (flexibility), επαναχρησιμοποιησιμότητα (reusability), μεταφερσιμότητα (portability) και διαλειτουργικότητα (interoperability).

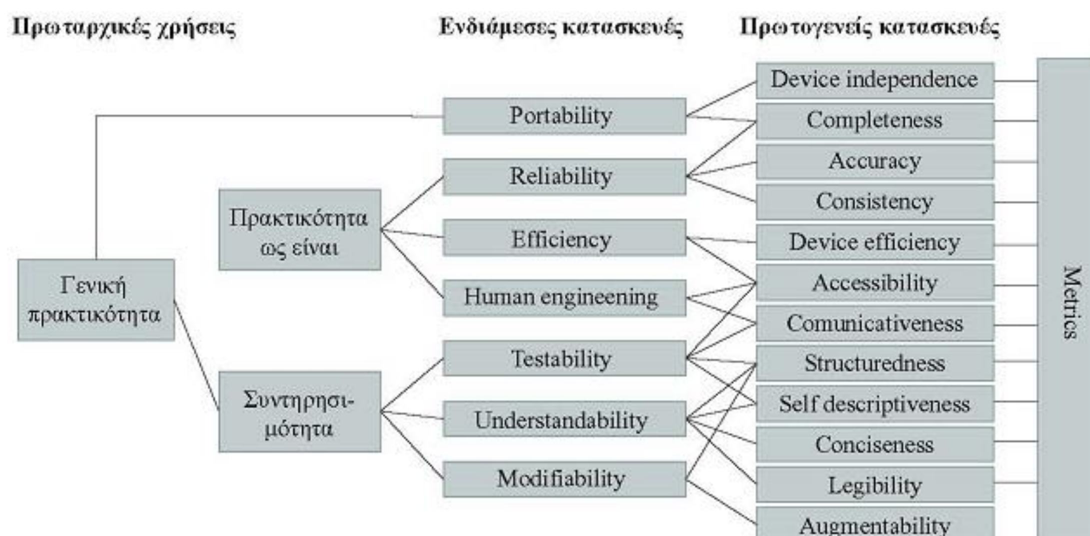


Μοντέλο FCM

Το μοντέλο αυτό αποτέλεσε ένα από τα πιο ολοκληρωμένα μοντέλα της εποχής του και έγινε η βάση για το διεθνές πρότυπο ISO 9126. Παρά τα προβλήματα της υποκειμενικότητας των ερωτήσεων, της ύπαρξης περιορισμένης κλίμακας (το μοντέλο δεχόταν μόνο απαντήσεις «ΝΑΙ» και «ΟΧΙ») και της αδυναμίας συνδυασμού μετρικών, το μοντέλο αυτό γνώρισε ευρεία αποδοχή και ακόμα και σήμερα, που το ISO9126 κυριαρχεί, αρκετές επιχειρήσεις βασίζουν το σύστημα ποιότητάς τους στην τμηματοποίηση του FCM μοντέλου. Ο βασικότερος λόγος για αυτό είναι ότι είναι καλύτερα προσαρμοσμένο στην ομάδα υλοποίησης και πιο αναλυτικό από το ISO 9126.

Το μοντέλο του Boehm ακολουθεί παρόμοια ιεραρχική δομή με αυτή του FCM μοντέλου, σύμφωνα με την οποία διασπά την ποιότητα του λογισμικού σε

πρωταρχικές χρήσεις (primary uses) και αυτές σε ενδιάμεσες κατασκευές (intermediate constructs), ανάλογες με τα κριτήρια ποιότητας του προηγούμενου μοντέλου, όπως φαίνεται και στο σχήμα παρακάτω. Οι ενδιάμεσες κατασκευές με τη σειρά τους διασπώνται σε πρωτογενείς κατασκευές (primitive constructs) οι όποιες μετρώνται άμεσα με μετρικές (metrics).



Μοντέλο του Boehm

Το μοντέλο του Boehm, παρόλο που ακολουθεί τη λογική της ανάλυσης της ποιότητας σε ποιοτικά χαρακτηριστικά, δεν είναι τόσο δομημένο όσο το FCM μοντέλο. Όμως, ήταν το πρώτο μοντέλο που εισήγαγε την πρακτική της χρήσης μετρικών λογισμικού αντί για ερωτήσεις στο κατώτερο επίπεδο. Αυτή η πρακτική εφαρμόστηκε με σχετική επιτυχία για τη μέτρηση των ποιοτικών χαρακτηριστικών, άσχετα αν το γενικό μοντέλο είναι το FCM, του Boehm, ή το ISO 9126.

Το πρότυπο ISO 9126 αποτελεί ένα μοντέλο ποιότητας λογισμικού που εξελίχθηκε σε διεθνές πρότυπο από το διεθνή οργανισμό τυποποίησης ISO. Ως μοντέλο ποιότητας λογισμικού διαφέρει από τα προγενέστερα μοντέλα τόσο στην ορολογία, όσο και στη δομή, καθώς είναι απόλυτα ιεραρχικό. Στο ISO 9126 κάθε χαρακτηριστικό ανήκει αυστηρά σε ένα παράγοντα ποιότητας χωρίς να υπάρχουν επικαλύψεις. Επίσης, κάθε χαρακτηριστικό είναι ορατό στο χρήστη και δεν αποτελεί εσωτερικό χαρακτηριστικό του προϊόντος. Με αυτή τη λογική, το ISO 9126 αντικατοπτρίζει περισσότερο τη θεώρηση από την πλευρά του χρήστη, σε

αντίθεση με τα δύο προγενέστερα μοντέλα που αντικατοπτρίζουν την προσέγγιση της ομάδας ανάπτυξης. Αυτός είναι και ο βασικός λόγος που –αν και πρότυπο– αρκετές επιχειρήσεις (οι ομάδες ανάπτυξης δηλαδή) εμμένουν ακόμα και σήμερα στο FCM μοντέλο. Άλλοι λόγοι είναι ότι το ISO 9126 δεν ορίζει καθαρά πώς μπορούν να μετρηθούν απευθείας τα επιμέρους χαρακτηριστικά του και η μη καλά ορισμένη διάσπαση των ποιοτικών χαρακτηριστικών σε επιμέρους υποχαρακτηριστικά.

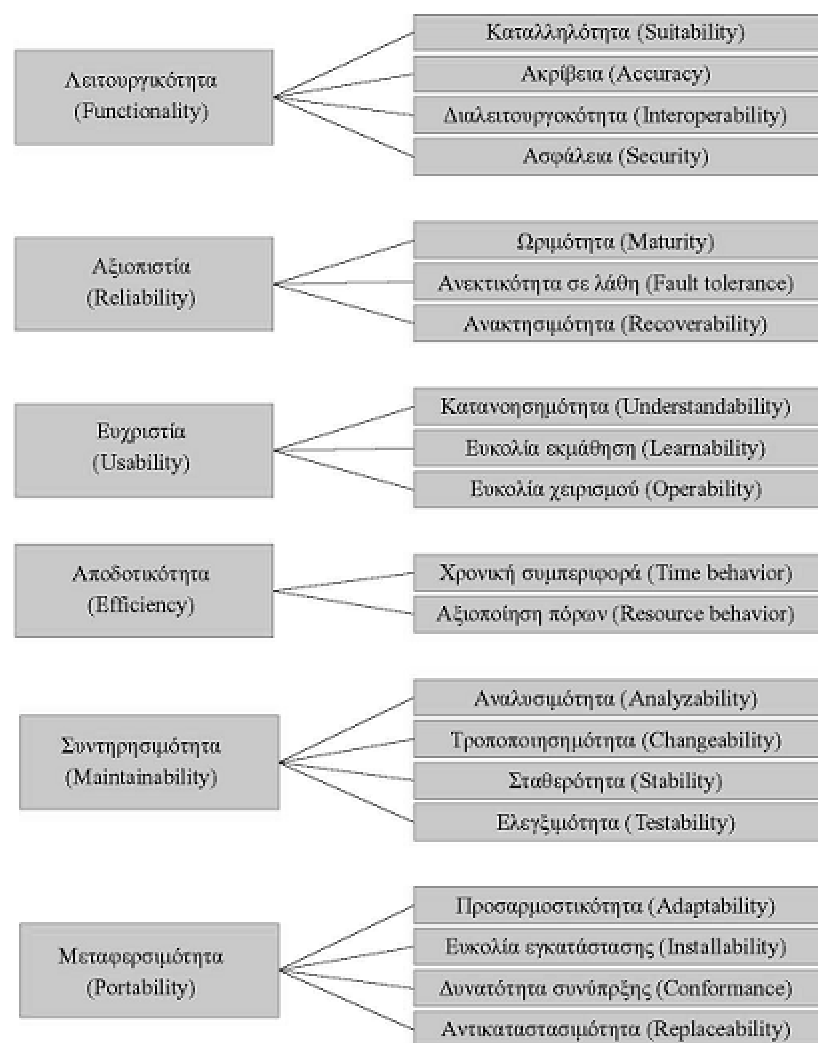
Στο σχήμα που ακολουθεί παρουσιάζεται το πρότυπο ISO 9126. Η λειτουργικότητα που, όπως είπαμε, είναι η δυνατότητα του λογισμικού να παρέχει όλες τις απαιτούμενες λειτουργίες, εμπεριέχει τα χαρακτηριστικά:

• Καταλληλότητα (suitability) που είναι η παροχή από το λογισμικό κατάλληλου συνόλου υπηρεσιών για προκαθορισμένο έργο και επιδιώξεις του χρήστη.

• Ακρίβεια (accuracy) που είναι η δυνατότητα του λογισμικού να παρέχει σωστά και αποδεκτά αποτελέσματα ή ενέργειες.

• Διαλειτουργικότητα (interoperability) που είναι η δυνατότητα του λογισμικού να αλληλεπιδρά με ένα ή περισσότερα προεγκατεστημένα ή προκαθορισμένα συστήματα.

• Ασφάλεια (security) που είναι η δυνατότητα του λογισμικού να προλαμβάνει αθέλητη προσπέλαση και να αντιστέκεται σε εσκεμμένες επιθέσεις που σκοπό έχουν τη μη εξουσιοδοτημένη πρόσβαση σε εμπιστευτικές πληροφορίες, ή τη μη εξουσιοδοτημένη τροποποίηση της περιεχόμενης πληροφορίας ή του προγράμματος, με σκοπό είτε να αποκτήσει ο εισβολέας κάποιο πλεονέκτημα είτε να παρεμποδίσει εξουσιοδοτημένους χρήστες να έχουν πρόσβαση στις υπηρεσίες του προγράμματος.



Το πρότυπο ISO 9126

Η αξιοπιστία που, όπως είπαμε, είναι η δυνατότητα του λογισμικού να λειτουργεί με σταθερό και συγκεκριμένο τρόπο κάτω από καθορισμένες συνθήκες, εμπεριέχει τα χαρακτηριστικά:

• Ωριμότητα (maturity) που είναι η δυνατότητα να αποφεύγονται προβλήματα που είναι αποτέλεσμα λαθών στο λογισμικό.

• Ανεκτικότητα σε λάθη (fault tolerance) που είναι η υποστήριξη από το λογισμικό καθορισμένου επιπέδου εφαρμογής σε περιπτώσεις λαθών στο λογισμικό ή παραβιάσεων στο περιβάλλον διεπαφής.

Η ευχρηστία που, όπως είπαμε, σχετίζεται με την ευκολία αντίληψης, εκμάθησης, χρήσης και ικανοποίησης του χρήστη, εμπεριέχει τα χαρακτηριστικά:

• Κατανοησιμότητα (understandability) που είναι η ιδιότητα του λογισμικού να καθιστά το χρήστη ικανό να καταλάβει πότε αυτό είναι κατάλληλο για τις ανάγκες

του και πώς μπορεί να χρησιμοποιηθεί για συγκεκριμένο έργο και συνθήκες χρήσης.

• Ευκολία εκμάθησης (learnability) που αφορά το βαθμό ευκολίας στον οποίο ο χρήστης μπορεί να μάθει την εφαρμογή.

• Ευκολία χειρισμού (operability) που είναι η δυνατότητα του λογισμικού να καθιστά το χρήστη ικανό να χειρίζεται και να ελέγχει την εφαρμογή.

• Ανακτησιμότητα (recoverability) που είναι η θεμελίωση από το λογισμικό του επιπέδου των εφαρμογών και η αποκατάσταση των δεδομένων που άμεσα επηρεάζονται σε περίπτωση αποτυχίας.

Η αποδοτικότητα που, όπως είπαμε, είναι η ικανότητα του λογισμικού να αποδίδει στο σύνολο των εφαρμογών που χρησιμοποιούνται, κάτω από καθορισμένες συνθήκες, εμπεριέχει τα χαρακτηριστικά:

• Χρονική συμπεριφορά (time behaviour) που είναι η ικανότητα του λογισμικού να παρέχει καθορισμένο και αποδεκτό χρόνο απόκρισης και εκτέλεσης διαδικασίας ή ενεργειών σε καθορισμένες συνθήκες.

• Αξιοποίηση πόρων (resource behaviour) που είναι το επίπεδο χρήσης συγκεκριμένων πόρων σε καθορισμένο χρόνο, όταν εκτελείται μια διαδικασία σε καθορισμένες συνθήκες.

Η συντηρησιμότητα που, όπως είπαμε, σχετίζεται με την ευκολία του λογισμικού να τροποποιείται, εμπεριέχει τα χαρακτηριστικά:

• Αναλυσιμότητα (analyzability) που είναι η δυνατότητα διάγνωσης του βαθμού ανεπάρκειας, ή των βλαβών ή λαθών στο λογισμικό, ή στα τμήματα που έχουν τροποποιηθεί.

• Τροποποιησιμότητα (changeability) που είναι η ευκολία υλοποίησης αλλαγών και τροποποίησης του λογισμικού.

• Σταθερότητα (stability) που είναι η δυνατότητα ελαχιστοποίησης ανεπιθύμητων αποτελεσμάτων που οφείλονται στις τροποποιήσεις του λογισμικού.

• Ελεγξιμότητα (testability) που είναι η δυνατότητα ελέγχου της αξιοπιστίας του λογισμικού που έχει τροποποιηθεί, ή πρόκειται να τροποποιηθεί.

Και τέλος, η μεταφερσιμότητα που, όπως είπαμε, σχετίζεται με τη δυνατότητα του λογισμικού να μπορεί να μεταφερθεί από ένα περιβάλλον σε άλλο.

• Προσαρμοστικότητα (adaptability) που είναι η δυνατότητα του λογισμικού να μπορεί να τροποποιηθεί, ώστε να εκτελεστεί σε διαφορετικά λειτουργικά περιβάλλοντα χωρίς να απαιτεί διαφορετικές πρακτικές χρήσης.

• Ευκολία εγκατάστασης (installability) που είναι η δυνατότητα εγκατάστασης σε κάποιο περιβάλλον.
• Δυνατότητα συνύπαρξης (conformance) που αφορά τη δυνατότητα συνύπαρξής του ως ανεξάρτητου λογισμικού σε περιβάλλον κοινό με άλλες εφαρμογές.
• Αντικαταστασιμότητα (replaceability) που είναι η δυνατότητα του λογισμικού να μπορεί να χρησιμοποιηθεί στο περιβάλλον ενός άλλου λογισμικού (αντικαθιστώντας κάποιο τμήμα του).

3.1 Έλεγχος παραγώμενου κώδικα

Προσπερνόντας όλα τα παραπάνω και αφήνοντας στην άκρη την έννοια της ποιότητας του τελικού προϊόντος που ονομάζουμε λογισμικό, θα προσπαθήσουμε τώρα να εμβαθύνουμε στην έννοια της ποιότητας του λογισμικού, πρίν αυτό γίνει τελικό προίόν, πιο συγκεκριμένα θα προσπαθήσουμε να δουμε τις διαδικασίες που ακολουθούνται ώστε να υπάρχει ένας έλεγχος τόσο της εγκυρότητας όσο και της ορθότητας του παραγόμενου κώδικα, ουσιαστικά θα δούμε την διαδικασία του ελέγχου της ποιότητας του κώδικα, γνωρίζοντας όμως γενικότερα ότι η διαδικασία ανάπτυξης λογισμικού είναι εξαιρετικά πολύπλοκη διαδικασία και στην οποία συμμετέχουν πολλά άτομα με ετερογενές υπόβαθρο. Οι συνεχείς αλλαγές απαιτήσεων από την πλευρά των πελατών καθώς και τα πιεστικά χρονικά όρια οδηγούν τους προγραμματιστές -πολλές φορές- σε προγραμματιστικές λύσεις που δεν υπακούν στις βασικές αρχές της αντικειμενοστρεφούς σχεδίασης λογισμικού. Το ίδιο αποτέλεσμα επιφέρει και η αύξηση του μεγέθους της εφαρμογής και συνεπώς και των γραμμών κώδικα. Οι πρόχειρες αυτές προσθήκες πολύ συχνά οδηγούν σε αλυσιδωτές αλλαγές και σε άλλα σημεία του λογισμικού. Όλα αυτά έχουν ως επακόλουθο η ποιότητα και η δομή του γραφόμενου κώδικα να μειώνεται συνεχώς. Η μείωση της ποιότητας σχεδίασης του κώδικα έχει ως συνέπεια την αυξανόμενη δυσκολία ενσωμάτωσης μελλοντικών αλλαγών στη λειτουργία του συστήματος οι οποίες προέρχονται από τις συνεχείς αλλαγές στις απαιτήσεις του πελάτη. Η δυσκολία αυτή προέρχεται από τη καταστρατήγηση τόσο των ευρετικών κανόνων (Heuristics) όσο και προτύπων σχεδίασης (Design Patterns) τα οποία όταν εφαρμόζονται εγγυώνται έναν καθορισμένο αλγοριθμικό και εύκολο τρόπο ενσωμάτωσης των αλλαγών στη λειτουργία του λογισμικού.

Αυτός ο τρόπος ενσωμάτωσης αφήνει αναλλοίωτη την έως τότε σχεδίαση του συστήματος. Όσο χειρότερη είναι η σχεδίαση του κώδικα, τόσο περισσότερος κόπος, χρόνος και συνεπώς πολύτιμες ανθρωποώρες απαιτούνται για να ενσωματωθεί μία νέα απαίτηση. Επειδή συνήθως δεν περισσεύουν ανθρωποώρες, οι νέες αλλαγές γίνονται ξανά με πρόχειρο τρόπο και η μείωση της ποιότητας της σχεδίασης συνεχίζεται.

Πρέπει λοιπόν με κάποιο τρόπο να γίνουν δραστικές αλλαγές στην σχεδίαση του συστήματος ούτως ώστε να σταματήσει η μείωση της ποιότητάς σχεδίασής του. Επειδή όμως όπως γίνεται αντιληπτό, δομικές αλλαγές σε υπάρχοντα συστήματα είναι πολύπλοκες απαιτείται ένας αυστηρά καθορισμένος τρόπος εντοπισμού των προβλημάτων και εφαρμογής των αλλαγών. Ένας τρόπος ο οποίος θα καθορίζεται όσο το δυνατόν αυστηρότερα, αν είναι δυνατόν αλγοριθμικά.

Η διαδικασία λοιπόν της Επαλήθευσης και Επικύρωσης - θα την αναφέρουμε από εδώ και στο εξής συντομογραφικά ως Ε&Ε - στην οποία οι δύο δραστηριότητες της φάσης Ε&Ε διαφέρουν μεταξύ τους τόσο ως προς το περιεχόμενο, όσο και ως προς το χρόνο εφαρμογής τους κατά τη διάρκεια του κύκλου ανάπτυξης του λογισμικού. Οι διαδικασίες επαλήθευσης (verification) εφαρμόζονται συνεχώς κατά τη διάρκεια του κύκλου ανάπτυξης και ορίζονται ως «οι διαδικασίες αξιολόγησης ενός συστήματος ή ενός τμήματος κάποιου συστήματος με στόχο να προσδιοριστεί κατά πόσο τα προϊόντα μιας συγκεκριμένης φάσης ανάπτυξης ικανοποιούν τις προδιαγραφές που είχαν τεθεί στην αρχή της φάσης» (IEEE[90]). Αντίθετα, οι διαδικασίες επικύρωσης (validation) εκτελούνται στο τέλος της διαδικασίας ανάπτυξης και ελέγχουν κατά πόσο το λογισμικό που αναπτύχθηκε συμφωνεί με τις αρχικές απαιτήσεις και τις προσδοκίες αυτών που θα το χρησιμοποιήσουν.

Η κακή ποιότητα του κώδικα και η καταστρατήγηση των αρχών σχεδίασης μπορεί να γίνει αντιληπτή από ορισμένα κομμάτια κώδικα τα οποία ονομάζονται χαρακτηριστικά «κακές οσμές» (Bad Smells). Ο υπεύθυνος σχεδίασης του έργου πρέπει να ανακαλύπτει αυτές τις κακές οσμές και να τις διορθώνει, βελτιώνοντας έτσι τη σχεδίαση του έργου. Επειδή τα σημεία με κακές οσμές σε ένα μεγάλο σύστημα είναι πάρα πολλά και είναι δύσκολο για έναν άνθρωπο να τα εντοπίσει με απλή αναζήτηση και ανάγνωση κώδικα, αναπτύχθηκαν (λίγα μέχρι στιγμής) εργαλεία που εντοπίζουν αυτόματα συγκεκριμένες κακές οσμές στον κώδικα και τις

παρουσιάζουν στον σχεδιαστή με έναν δομημένο τρόπο έτσι ώστε να κάνουν το έργο του ευκολότερο.

Η εύρεση των κακών οσμών είναι ο μισός δρόμος που πρέπει να διανυθεί μέχρι επιτευχθεί η δομική βελτίωση της ποιότητας του κώδικα. Ο άλλος μισός είναι η απαλοιφή της οσμής. Για την απαλοιφή της οσμής εφαρμόζεται μία διαδικασία η οποία ονομάστηκε από τους ειδικούς «Αναδόμηση», (Refactoring). Αναδόμηση είναι η διαδικασία που αλλάζει τον κώδικα ενός λογισμικού με τέτοιο τρόπο ώστε να βελτιώνεται εσωτερική σχεδίαση χωρίς να αλλάζει η εξωτερική συμπεριφορά του προγράμματος[21]. Είναι ο μοναδικός τρόπος απαλοιφής των κακών οσμών, ο οποίος εγγυάται ότι η λειτουργικότητα του συστήματος μετά την αλλαγή θα είναι ακριβώς ίδια με πριν.

Η διαδικασία εφαρμογής της αναδόμησης καθορίζεται από συγκεκριμένα βήματα τα οποία πρέπει να εκτελεστούν με μία συγκεκριμένη σειρά. Αυτός ο αλγοριθμικός τρόπος μας επιτρέπει να αυτοματοποιηθεί η διαδικασία εφαρμογής της αναδόμησης. Το γεγονός αυτό συνεπάγεται μεγάλα πλεονεκτήματα, καθώς η εφαρμογή της αναδόμησης «με το χέρι» από τον σχεδιαστή δεν είναι πάντα μια εύκολη υπόθεση. Πολλές φορές χρειάζεται να αλλάξουν περισσότερες από μία μονάδες λογισμικού και να ληφθούν υπόψη πολλοί διαφορετικοί περιορισμοί. Η αυτοματοποιημένη διαδικασία γλιτώνει τον σχεδιαστή από κόπο και χρόνο. Για τον σκοπό αυτό έχουν αναπτυχθεί πολλά εργαλεία τα οποία εφαρμόζουν αυτόματα συγκεκριμένες αναδομήσεις. Τα εργαλεία αυτά είτε είναι ενσωματωμένα ως λειτουργίες σε μεγάλα ολοκληρωμένα περιβάλλοντα ανάπτυξης λογισμικού, με διασημότερο από όλα αυτά το -ανοικτού κώδικα- Eclipse[22], είτε παρέχονται ως ξεχωριστά προγράμματα.

Στην ιδανική περίπτωση λοιπόν, ένα πρόγραμμα λογισμικού που παραδίδεται προς χρήση δεν θα πρέπει να παρουσιάζει σφάλματα εκτέλεσης. Στην πράξη, αυτό είναι σχεδόν αδύνατο, ιδιαίτερα όταν πρόκειται για ένα μεγάλο σύστημα λογισμικού. Επιπλέον, όσο πολλές δοκιμές εκτέλεσης (testing) του κώδικα ενός προγράμματος (source program code) και αν γίνουν, δεν αρκούν για να αποδείξουν ότι το λογισμικό δεν έχει λάθη. Η εμπειρία έχει δείξει ότι ο αριθμός των λαθών που απομένουν σε ένα πρόγραμμα είναι ανάλογος με τον αριθμό των

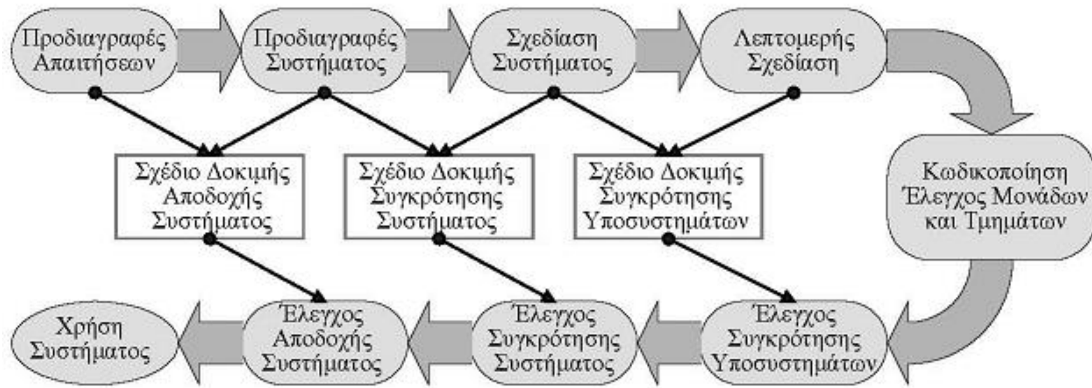
[21] *Fowler, M, Beck, K, Brant, J, Opdyke, W and Roberts, D 1999, Refactoring: Improving the Design of Existing Code, Addison Wesley, Boston, MA.*
[22] *www.eclipse.org*

λαθών που έχουν ήδη ανακαλυφθεί και διορθωθεί. Αυτό σημαίνει ότι είναι καλύτερο να εμπιστευόμαστε ένα λογισμικό στο οποίο βρέθηκαν λίγα λάθη μετά από εξαντλητικές δοκιμές, παρά ένα λογισμικό το οποίο έχει διορθωθεί πολλές φορές.

Η ενσωμάτωση διαδικασιών Ε&Ε σε μια φάση του κύκλου ζωής επιπλέον των δραστηριοτήτων ανάπτυξης της φάσης οδηγεί στην αύξηση του κόστους εκτέλεσης της φάσης και της πολυπλοκότητας του χρονοδιαγράμματος. Επιπλέον, περιορίζονται και οι διαθέσιμοι για την ανάπτυξη του λογισμικού πόροι, αφού προσωπικό, εξοπλισμός, χρόνος και χρήματα θα πρέπει να διατεθούν στις δραστηριότητες που υλοποιούν τις διαδικασίες Ε&Ε. Όμως, η σωστή και έγκαιρη εφαρμογή δραστηριοτήτων Ε&Ε συντελεί στον εντοπισμό λαθών και παρανοήσεων νωρίς στον κύκλο ανάπτυξης. Έτσι, μειώνεται τελικά το συνολικό κόστος του έργου, αφού διευκολύνονται (και συνεπώς κοστίζουν λιγότερο) οι τελικές διαδικασίες ελέγχου και αποδοχής, βελτιώνονται οι δυνατότητες διαχείρισης του έργου και μειώνεται ο κίνδυνος αποτυχίας.

Ταυτόχρονα, βελτιώνονται η ποιότητα και η αξιοπιστία του λογισμικού. Για παράδειγμα, η τελευταία ορίζεται ως «η πιθανότητα λειτουργίας του λογισμικού χωρίς αστοχία για ένα καθορισμένο χρονικό διάστημα, σε ένα καθορισμένο περιβάλλον και για ένα καθορισμένο σκοπό». Από την άλλη πλευρά, οι αναθεωρήσεις αποτελούν τη βασική μέθοδο ελέγχου της ποιότητας μιας διαδικασίας ανάπτυξης λογισμικού ή ενός προϊόντος λογισμικού.

Σε αναγνώριση της σημασίας των δραστηριοτήτων εγκυροποίησης, το διαδεδομένο μοντέλο κύκλου ζωής «καταρράκτη» επεκτάθηκε συσχετίζοντας κάθε φάση ανάπτυξης με μια δραστηριότητα Ε&Ε. Το μοντέλο ανάπτυξης λογισμικού που προέκυψε είναι γνωστό ως «μοντέλο V» και απεικονίζεται στο παρακάτω σχήμα. Σύμφωνα με αυτό, οι δραστηριότητες Ε&Ε εκτελούνται πάνω σε συγκεκριμένα σχέδια δοκιμών (test plans), τα οποία προκύπτουν με βάση τα εξαγόμενα των αντίστοιχων φάσεων ανάπτυξης.



Μοντέλο V

Οι δραστηριότητες Ε&Ε συνήθως ανατίθενται στην ομάδα διασφάλισης της ποιότητας του λογισμικού, η οποία εξετάζει τα προϊόντα του έργου προσπαθώντας να διασφαλίσει ότι οι μέθοδοι και τα εργαλεία που χρησιμοποιήθηκαν για τη σχεδίαση και ανάπτυξη του λογισμικού είναι αποδεκτά και οδηγούν τελικά στην παράδοση έγκυρου λογισμικού.

Ο έλεγχος των μονάδων του λογισμικού (software program units) γίνεται από τους προγραμματιστές που αναπτύσσουν το λογισμικό, συνήθως χωρίς οργανωμένο σχέδιο. Όταν όμως πολλές μονάδες συνδυαστούν σε ένα τμήμα ή υποσύστημα, τότε ο έλεγχος γίνεται από ανεξάρτητη ομάδα (π.χ. ομάδα διασφάλισης ποιότητας) με βάση το «σχέδιο δοκιμής συγκρότησης υποσυστήματος» που έχει ήδη συντεθεί κατά τις φάσεις σχεδίασης του συστήματος. Αντίστοιχα, η συγκρότηση (integration - αναφέρεται και ως «ολοκλήρωση») του συστήματος γίνεται με βάση το «σχέδιο δοκιμής συγκρότησης συστήματος» που έχει συντεθεί κατά την παραγωγή των προδιαγραφών, ενώ ο τελικός έλεγχος πριν την παράδοση του συστήματος γίνεται με βάση το «σχέδιο δοκιμής αποδοχής» που έχει ήδη συντεθεί κατά τον ορισμό των απαιτήσεων από το σύστημα.

Οι έλεγχοι και οι άλλες δραστηριότητες του μοντέλου V θα παρουσιαστούν αναλυτικότερα παρακάτω. Προς το παρόν σημειώνουμε ότι απαιτείται η σύνταξη ενός συνολικού «σχεδίου Ε&Ε», στο οποίο θα περιγράφονται επακριβώς οι διαδικασίες Ε&Ε που θα εκτελεστούν στα πλαίσια του έργου. Το σχέδιο Ε&Ε θεωρείται τόσο σπουδαίο, ώστε η δομή του να περιγράφεται από ένα πρότυπο του IEEE[23] ,το οποίο απαιτεί για κάθε φάση Ε&Ε να ορίζονται τα εξής:

• Εργασίες Ε&Ε

[23]*IEEE (1986), Software Verification and Validation Plans, ANSI – IEEE Standard 1012 – 1986, IEEE Press, New York.*

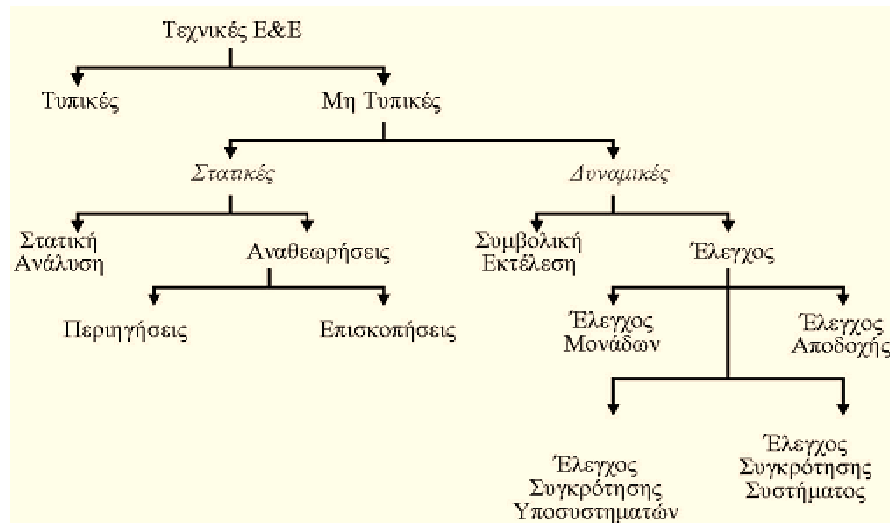
- Μέθοδοι και κριτήρια
- Είσοδοι και έξοδοι
- Χρονοπρογραμματισμός
- Πόροι
- Κίνδυνοι και παραδοχές
- Ρόλοι και υπευθυνότητες

Έως τώρα έχει προταθεί μια πληθώρα τεχνικών Ε&Ε, κάτι που δείχνει πόσο σημαντικές είναι αυτές οι δραστηριότητες στον κύκλο ανάπτυξης λογισμικού. Οι τεχνικές αυτές χωρίζονται σε δύο μεγάλες κατηγορίες και εμφανίζονται και στο παρακάτω σχήμα :

- στις τυπικές τεχνικές Ε&Ε (formal techniques), οι οποίες παράγουν μια αυστηρή μαθηματική απόδειξη ότι το λογισμικό λειτουργεί σύμφωνα με τις προδιαγραφές του. Το αποτέλεσμα μιας τέτοιας διαδικασίας είναι συνήθως δυαδικό: το δοκιμαζόμενο τμήμα κώδικα είτε ικανοποιεί τις προδιαγραφές του, είτε όχι
- στις μη τυπικές τεχνικές Ε&Ε (informal techniques), οι οποίες δεν μπορούν να αποδείξουν την ορθότητα του λογισμικού, αλλά επικεντρώνονται στην ανακάλυψη και αντιμετώπιση όσο το δυνατόν περισσότερων λαθών. Οι τεχνικές αυτές εφαρμόζονται επαναληπτικά μέχρι να αποκτηθεί ένας βαθμός εμπιστοσύνης στον ισχυρισμό ότι το λογισμικό είναι σωστό και δεν έχει λάθη (δηλαδή, το λογισμικό είναι «μάλλον ορθό»).

Οι μη τυπικές τεχνικές Ε&Ε είναι πιο φθηνές και εύκολες στην εφαρμογή, γι’ αυτό και είναι οι περισσότερο διαδεδομένες. Ανάλογα με τη μέθοδο ανακάλυψης λαθών που χρησιμοποιούν διακρίνονται σε:

- στατικές τεχνικές Ε&Ε (static techniques), οι οποίες ασχολούνται με την εξέταση διαφόρων αναπαραστάσεων του συστήματος λογισμικού. Τέτοιες αναπαραστάσεις είναι οι προδιαγραφές, το σχέδιο του συστήματος και βεβαίως ο κώδικας των προγραμμάτων. Η εφαρμογή των τεχνικών αυτών μπορεί να αυτοματοποιηθεί όταν η υπό εξέταση αναπαράσταση έχει περιγραφεί με κάποιο συντακτικά αυστηρό τρόπο
- δυναμικές τεχνικές Ε&Ε (dynamic techniques), με τις οποίες εξετάζεται ο τρόπος υλοποίησης και λειτουργίας ενός λογισμικού. Η πιο σημαντική δυναμική τεχνική είναι ο έλεγχος του προγράμματος (program testing).


Τεχνικές Ε&Ε
Τυπικές
Μη Τυπικές
Στατικές
Δυναμικές
Στατική Ανάλυση
Αναθεωρήσεις
Συμβολική Εκτέλεση
Έλεγχος
Περιηγήσεις
Επισκοπήσεις
Έλεγχος Μονάδων
Έλεγχος Αποδοχής
Έλεγχος Συγκρότησης Υποσυστηματών
Έλεγχος Συγκρότησης Συστήματος

4. Εργαλεία ποιότητας λογισμικού

Η στατική ανάλυση (static analysis) μπορεί να εφαρμοστεί για την επαλήθευση εκείνων των προϊόντων του κύκλου ανάπτυξης που μπορούν να περιγραφούν με κάποιο τυπικό ή δομημένο τρόπο. Τέτοια είναι οι προδιαγραφές, η σχεδίαση, αλλά κυρίως ο κώδικας του λογισμικού, στον οποίο εφαρμόζεται περισσότερο αυτή η τεχνική. Αν και η στατική ανάλυση του κώδικα μπορεί να γίνει και χειρωνακτικά, χρησιμοποιώντας τεχνικές περιήγησης ή επισκόπησης συνήθως χρησιμοποιούνται αυτόματα εργαλεία (οι στατικοί αναλυτές). Τα εργαλεία αυτά δέχονται στην είσοδο τον πηγαίο κώδικα ενός προγράμματος, εξετάζουν τη δομή του χωρίς να εκτελέσουν το πρόγραμμα και ανακαλύπτουν διάφορα λάθη, παραλείψεις ή ανωμαλίες στον τρόπο προγραμματισμού.

Για να το κάνουν αυτό, οι στατικοί αναλυτές δημιουργούν για κάθε πρόγραμμα[24] τα εξής:

• Έναν πίνακα συμβόλων (symbol table), όπου καταγράφονται πληροφορίες για όλες τις μεταβλητές που χρησιμοποιούνται στο πρόγραμμα (π.χ. τύπος, εντολή δήλωσης, εντολές καταχώρισης νέας τιμής, εντολές χρήσης κ.ά.).

• Ένα γράφημα ροής ελέγχου (control flow graph), οι κόμβοι του οποίου αναπαριστούν τα «βασικά τμήματα» του λογισμικού και οι πλευρές τη ροή ελέγχου ανάμεσα σε αυτά. Ένα βασικό τμήμα έχει την ιδιότητα ότι εάν εκτελεστεί η πρώτη εντολή του, τότε θα εκτελεστούν όλες οι εντολές που περιέχει.

• Ένα γράφημα κλήσεων (call graph), οι κόμβοι του οποίου αναπαριστούν τις υπορουτίνες (συναρτήσεις, διαδικασίες κ.λπ.) του λογισμικού και οι πλευρές μια πιθανή κλήση μιας υπορουτίνας από μια άλλη.

Η στατική ανάλυση περιλαμβάνει τα ακόλουθα στάδια[25] :

• Ανάλυση ροής ελέγχου: Εντοπίζονται βρόχοι (loops) με πολλαπλά σημεία εισόδου ή εξόδου και κώδικας που δεν εκτελείται ποτέ (πρόκειται για τμήματα κώδικα που αφενός βρίσκονται ανάμεσα σε δύο εντολές goto χωρίς συνθήκη και αφετέρου δεν γίνεται αναφορά σε αυτά από κανένα άλλο σημείο του προγράμματος).

[24] *R. Fairley (1985), Software Engineering Concepts. McGraw – Hill, Singapore*
[25] *I. Sommerville (1996), Software Engineering. Addison – Wesley, USA*

• Ανάλυση χρήσης δεδομένων: Εξετάζεται η χρήση των μεταβλητών και εντοπίζονται μεταβλητές που χρησιμοποιούνται χωρίς πρότερη αρχικοποίηση και μεταβλητές που ενώ έχουν δηλωθεί, δεν χρησιμοποιούνται ποτέ.
• Ανάλυση διεπαφών: Ελέγχεται η συνέπεια του τρόπου κλήσης των υπορουτινών του λογισμικού σε σχέση με τον τρόπο που αυτές έχουν δηλωθεί. Επιπλέον, εντοπίζονται υπορουτίνες που ενώ δηλώνονται, δεν χρησιμοποιούνται ποτέ οι ίδιες ούτε τα αποτελέσματα που επιστρέφουν.
• Ανάλυση ροής πληροφορίας: Για κάθε μεταβλητή εξόδου, εντοπίζονται όλες οι μεταβλητές εισόδου από τις οποίες αυτή εξαρτάται, καθώς και οι συνθήκες από τις οποίες εξαρτάται η τιμή της.
• Ανάλυση μονοπατιού εκτέλεσης: Εντοπίζονται όλα τα δυνατά μονοπάτια εκτέλεσης του λογισμικού και καταγράφονται οι εντολές που εκτελούνται σε κάθε μονοπάτι.

Στην πραγματικότητα, στα δύο τελευταία στάδια δεν γίνεται εντοπισμός λαθών. Ο στόχος είναι να παραχθεί ένα σύνολο πληροφοριών, το οποίο αποτελεί ουσιαστικά εναλλακτική περιγραφή του υπό ανάλυση λογισμικού. Ο (συνήθως τεράστιος) όγκος των πληροφοριών αυτών χρησιμοποιείται για ανάλυση του λογισμικού με κάποια άλλη τεχνική (π.χ. επισκόπηση των διαφορετικών μονοπατιών εκτέλεσης).

Στον πίνακα Α που ακολουθεί, αναφέρονται συγκεντρωτικά μερικοί από τους ελέγχους που εκτελούν οι στατικοί αναλυτές.

Σφάλματα δεδομένων	Μεταβλητές που χρησιμοποιούνται χωρίς αρχικοποίηση. Μεταβλητές που έχουν δηλωθεί αλλά δεν χρησιμοποιούνται Μεταβλητές στις οποίες γίνονται δύο διαδοχικές καταχωρίσεις τιμής χωρίς να χρησιμοποιούνται ενδιάμεσα. Πιθανή υπέρβαση ορίων πίνακα. Μεταβλητές που δεν έχουν δηλωθεί.
Σφάλματα ελέγχου	Κώδικας στον οποίο ποτέ δεν φτάνει ο έλεγχος εκτέλεσης. Διακλαδώσεις χωρίς συνθήκη μέσα σε βρόχους.
Σφάλματα εισόδου / εξόδου	Μεταβλητές που γράφονται στην έξοδο διαδοχικές φορές χωρίς να γίνεται ενδιάμεσα καταχώριση τιμής.
Σφάλματα διεπαφών	Ασυμφωνία στον τύπο των παραμέτρων κλήσης υπορουτινών. Ασυμφωνία στον αριθμό των παραμέτρων κλήσης υπορουτινών. Αποτελέσματα συναρτήσεων που δεν χρησιμοποιούνται ποτέ. Υπορουτίνες που δεν καλούνται ποτέ.
Σφάλματα διαχείρισης χώρου αποθήκευσης	Δείκτες που δεν χρησιμοποιούνται. Αριθμητικά λάθη σε δείκτες.
Γενικές πληροφορίες	Συνολικό πλήθος γραμμών κώδικα. Συνολικό πλήθος και θέση γραμμών με σχόλια. Συνολικό πλήθος και θέση γραμμών με εντολές διακλάδωσης. Συνολικό πλήθος και θέση γραμμών με κλήσεις υπορουτινών συστήματος. Συνολικό πλήθος και θέση των σταθερών του προγράμματος.

Πίνακας Α : Κατάλογος ελέγχων που εκτελούν οι στατικοί αναλυτές

Η στατική ανάλυση δεν μπορεί να αντικαταστήσει τις επιθεωρήσεις ή τον έλεγχο του λογισμικού, γιατί υπόκειται σε πολλούς πρακτικούς αλλά και θεωρητικούς περιορισμούς (Fairley[85], Σκορδαλάκης[91]). Στην πρώτη κατηγορία αναφέρουμε ενδεικτικά τους εξής:

• Μόνο πιθανά (και όχι εξακριβωμένα) λάθη χρήσης δεικτών πινάκων ή μεταβλητών τύπου δείκτη μπορούν να ανακαλυφθούν, γιατί αυτές οι μεταβλητές παίρνουν διαφορετικές τιμές κάθε φορά που το πρόγραμμα εκτελείται.

• Δεν είναι δυνατή η ανακάλυψη εντολών λανθασμένης αρχικοποίησης μεταβλητών.

• Δεν είναι δυνατός ο εντοπισμός μιας κλήσης υπορουτίνας όπου περνιέται λανθασμένη παράμετρος του σωστού τύπου.

Ο βασικός θεωρητικός περιορισμός των δυνατοτήτων των στατικών αναλυτών προέρχεται από τη θεωρία υπολογισμού, όπου το θεώρημα δυνατότητας λήψης απόφασης (decidability theorem) δηλώνει ότι «δεδομένου ενός προγράμματος, το οποίο έχει γραφεί σε μια γλώσσα προγραμματισμού ικανή να εξομοιώσει μια Μηχανή Turing[26] , δεν είναι δυνατό να εξεταστεί με αλγοριθμικό τρόπο και να αποφασιστεί εάν μια οποιαδήποτε εντολή του προγράμματος θα εκτελεστεί τελικά, όταν το πρόγραμμα εκτελείται με δεδομένα εισόδου που έχουν επιλεγεί τυχαία». Σε αντίθετη περίπτωση θα ήταν δυνατή η επίλυση και του προβλήματος τερματισμού (halting problem).

Αυτό σημαίνει ότι δεν είναι δυνατό να αναπτυχθεί ένας στατικός αναλυτής, ο οποίος εξετάζοντας τον κώδικα ενός οποιουδήποτε προγράμματος με τυχαία δεδομένα εισόδου, θα μπορεί να αποφανθεί εάν κατά την εκτέλεση του προγράμματος θα εκτελεστεί κάποια συγκεκριμένη εντολή. Το πρόβλημα γίνεται αποφασίσιμο μόνο εάν θέσουμε περιορισμούς στη δομή του προγράμματος (π.χ. ένας στατικός αναλυτής μπορεί να εγγυηθεί την εκτέλεση όλων των εντολών ενός προγράμματος χωρίς διακλαδώσεις).

Η αναθεώρηση (review) είναι μια τεχνική διασφάλισης της ορθότητας και της ποιότητας του λογισμικού που μπορεί να εφαρμοστεί σε διάφορα σημεία του κύκλου ανάπτυξης του συστήματος λογισμικού. Οι στόχοι μιας αναθεώρησης είναι (Pressman[94]):

• να ανακαλύψει λάθη στη λειτουργία, λογική ή υλοποίηση οποιασδήποτε αναπαράστασης του λογισμικού,

• να επαληθεύσει ότι το λογισμικό ικανοποιεί τις προδιαγραφές του,

• να διασφαλίσει ότι το λογισμικό υιοθετεί αναγνωρισμένα πρότυπα,

• να αμβλύνει τις διαφορές στον τρόπο σχεδίασης ή ανάπτυξης του λογισμικού,

• να διευκολύνει τη διαχείριση του έργου,

[26] *Η μηχανή που περιέγραψε το 1937 ο Alan Turing μπορεί να κάνει κάθε συμβολικό υπολογισμό χρησιμοποιώντας μόνο μια απείρου μεγέθους ταινία χαρτιού, μια κεφαλή γραφής/ανάγνωσης από την ταινία και ένα σύνολο κανόνων που μετακινούν την κεφαλή κατά μια θέση πάνω στην ταινία ή την αναγκάζουν να γράψει σε ή να διαβάσει ένα σύμβολο από αυτή. Έχει αποδειχθεί ότι, εάν αγνοηθούν παράμετροι ταχύτητας και διαθεσιμότητας υπολογιστικών πόρων, αυτή η τόσο απλή υποθετική μηχανή, αποτελεί το γενικότερο και ισχυρότερο τυπικό μοντέλο υπολογισμού.*

• να προωθήσει την επικοινωνία ανάμεσα στα μέλη της ομάδας ανάπτυξης του λογισμικού,

• να βοηθήσει στην εκπαίδευση νέων μελών της ομάδας ανάπτυξης του λογισμικού. Στην πραγματικότητα υπάρχουν διάφοροι τύποι αναθεωρήσεων, οι οποίοι παρουσιάζουν αρκετά κοινά χαρακτηριστικά:

• Συνήθως συμμετέχουν λίγα άτομα (3 – 5 συμμετέχοντες είναι αρκετοί).

• Πρέπει να δίδεται σε όλους αρκετός χρόνος (όχι όμως περισσότερο από 2 ώρες) και μέσα για την προετοιμασία της αναθεώρησης.

• Η διάρκεια της αναθεωρητικής συνάντησης δεν πρέπει να ξεπερνά τις 2 ώρες.

• Σε κάθε συνάντηση εξετάζεται ένα ολοκληρωμένο αλλά σχετικά μικρό τμήμα του λογισμικού, ώστε να αυξηθεί η πιθανότητα ανακάλυψης λαθών.

Στο τέλος της αναθεώρησης, όλοι οι συμμετέχοντες πρέπει να αποφασίσουν εάν:

• θα αποδεχθούν το προϊόν χωρίς περαιτέρω τροποποιήσεις,

• θα απορρίψουν το προϊόν εξ’ αιτίας σοβαρών λαθών (αυτό σημαίνει ότι όταν διορθωθούν τα λάθη, θα γίνει νέα αναθεώρηση),

• θα αποδεχθούν προσωρινά το προϊόν επειδή έχουν βρεθεί όχι κρίσιμα λάθη (που σημαίνει ότι τα λάθη πρέπει να διορθωθούν, αλλά δεν θα γίνει νέα αναθεώρηση).

Μπορεί οι διάφοροι τύποι αναθεωρήσεων να μοιάζουν στον τρόπο οργάνωσης και διεξαγωγής , διαφέρουν όμως ως προς το στόχο. Στη συνέχεια θα παρουσιαστούν οι δύο πιο συνηθισμένοι τύποι, οι περιηγήσεις και οι επισκοπήσεις.

Τα στάδια της αναθεώρησης

Οι οδηγίες διεξαγωγής αναθεωρήσεων πρέπει να έχουν εκ των προτέρων καθοριστεί, συμφωνηθεί και διανεμηθεί σε όλους τους συμμετέχοντες, ώστε η αναθεώρηση να μην ξεφύγει από τον έλεγχο. Σε αυτές μπορεί να περιλαμβάνονται οι ακόλουθες δέκα[27] :

1. Όλοι πρέπει να θυμούνται ότι κατά την αναθεώρηση εξετάζεται το προϊόν λογισμικού και όχι η ομάδα ή το άτομο που το ανέπτυξε.
2. Η «ατζέντα» της συνάντησης, ή ο κατάλογος των σημείων που θα εξεταστούν, πρέπει να έχει προσχεδιαστεί και γνωστοποιηθεί στους συμμετέχοντες. Η αναθεώρηση πρέπει να ακολουθεί αυστηρά την προσυμφωνημένη ατζέντα.
3. Μόνο βασικά ζητήματα πρέπει να θίγονται. Δεν πρέπει να γίνεται αναλυτική συζήτηση επί των θεμάτων που προκύπτουν, αλλά είναι αρκετό αυτά να καταγράφονται.
4. Δεν πρέπει να γίνεται προσπάθεια επίλυσης των προβλημάτων που ανακύπτουν, αλλά να δίνεται έμφαση μόνο στην ανακάλυψη των λαθών. Τα προβλήματα λύνονται αργότερα από τον υπεύθυνο του υπό αναθεώρηση λογισμικού, ο οποίος σε κάποια άλλη συνάντηση (ή με ένα υπόμνημα) ενημερώνει τους αναθεωρητές.
5. Κάποιος πρέπει να κρατά σημειώσεις.
6. Ο αριθμός των συμμετεχόντων πρέπει να είναι περιορισμένος, αλλά όλοι πρέπει να έχουν προετοιμαστεί.
7. Κάθε προϊόν πρέπει να εξετάζεται με βάση έναν κατάλογο χαρακτηριστικών.
8. Οι αναθεωρήσεις αποτελούν απαραίτητη δραστηριότητα και πρέπει να προβλέπονται στο χρονοδιάγραμμα του έργου. Έτσι, οι συμμετέχοντες θα τις θεωρούν ως μέρος των υποχρεώσεών τους και όχι σαν κάτι έκτακτο.
9. Οι συμμετέχοντες στην αναθεώρηση (λέγονται και αναθεωρητές) καλό είναι να είναι έμπειροι και να έχουν εκπαιδευτεί τόσο σε τεχνικά θέματα όσο και σε θέματα ψυχολογίας.
10. Η διαδικασία αναθεώρησης πρέπει τακτικά να αναθεωρείται.

Κατά την περιήγηση (walkthrough), ο αναθεωρούμενος (reviewee), ο οποίος έχει κατασκευάσει το υπό αναθεώρηση προϊόν, παρουσιάζει το προϊόν αυτό στους αναθεωρητές (reviewers). Η συνάντηση αρχίζει με συζήτηση επί της ατζέντας και συνεχίζει με μια σύντομη εισαγωγή από τον αναθεωρούμενο. Στη

[27] *R. Pressman (1994), Software Engineering: a practitioner's approach. McGraw – Hill, England*

συνέχεια, αυτός παρουσιάζει στους αναθεωρητές το προϊόν, εξηγώντας τη λειτουργία του και τον τρόπο που αναπτύχθηκε. Η παρουσίαση πρέπει να είναι περιγραφική και μεστή, σαν ο αναθεωρούμενος να «περιηγείται» μέσα στο προϊόν. Κατά την παρουσίαση, οι αναθεωρητές επισημαίνουν και καταγράφουν λάθη ή παρατηρήσεις. Στην περιήγηση συμμετέχουν, εκτός του αναθεωρούμενου και ο υπεύθυνος του έργου, μέλη της ομάδας ανάπτυξης του προϊόντος, ένας εκπρόσωπος της ομάδας ποιότητας, ένας υπεύθυνος για την καταγραφή των συζητήσεων και άλλα άτομα που ενδιαφέρονται για το έργο (π.χ. στις αρχικές φάσεις του έργου καλό είναι να συμμετέχουν και εκπρόσωποι του πελάτη ή των χρηστών). Της περιήγησης προΐσταται ο υπεύθυνος του έργου ή ο υπεύθυνος ποιότητας του λογισμικού, ο οποίος συντονίζει τη συζήτηση και είναι υπεύθυνος για την τήρηση της ατζέντας και τη διατήρηση ενός κλίματος άνεσης και εμπιστοσύνης ανάμεσα στους συμμετέχοντες. Ένας από τους αναθεωρητές είναι υπεύθυνος για την καταγραφή της συζήτησης και τη σύνταξη της «Αναφοράς Περιήγησης», στην οποία περιγράφονται: (α) το προϊόν που εξετάσθηκε, (β) οι αναθεωρητές, (γ) ο κατάλογος με τα ζητήματα που ανέκυψαν και (δ) οι παρατηρήσεις και τα συμπεράσματα.

Κατά την επισκόπηση (inspection), μια ομάδα εποπτών (inspectors) υποβάλλει το προϊόν σε ένα σύνολο προκαθορισμένων ελέγχων με στόχο την ανακάλυψη λαθών. Η εμπειρία έχει δείξει ότι η επισκόπηση είναι αποδοτική όταν ελέγχεται η σχεδίαση ή ο κώδικας ενός τμήματος λογισμικού (το τελευταίο είδος επισκόπησης είναι τόσο αποτελεσματικό που τείνει να αντικαταστήσει τον έλεγχο των μονάδων λογισμικού). Η επισκόπηση είναι μια τυπική διαδικασία και κάθε μέλος της ομάδας εποπτών έχει συγκεκριμένο ρόλο σε αυτή (Πίνακας Β). Υπεύθυνος για τη διεξαγωγή της επισκόπησης είναι ο πρόεδρος, ο οποίος σχεδιάζει τη διαδικασία, συγκροτεί την ομάδα, φροντίζει για τη διανομή του υλικού προετοιμασίας, συνθέτει τον κατάλογο ελέγχων, διασφαλίζει ότι το υπό επισκόπηση προϊόν βρίσκεται σε κατάλληλη κατάσταση, οργανώνει και συντονίζει τη συνάντηση, φροντίζει για την καταγραφή των λαθών και επιβλέπει την κατοπινή επιδιόρθωσή τους. Πραγματικά, μετά την επισκόπηση ακολουθεί ένα στάδιο προσαρμογής ή διόρθωσης του λογισμικού από την ομάδα που το ανέπτυξε, σύμφωνα με τις παρατηρήσεις της ομάδας των ελεγκτών (οι οποίοι πρέπει να αποφύγουν να κάνουν υποδείξεις για τον τρόπο που θα γίνει η προσαρμογή ή η διόρθωση). Μετά από το στάδιο αυτό, ο Πρόεδρος πρέπει να αποφασίσει εάν

απαιτείται νέα επισκόπηση ή τα προβλήματα έχουν αντιμετωπιστεί επιτυχώς.Η επισκόπηση, εάν διεξαχθεί σωστά, μπορεί να αποφέρει πολλά οφέλη στην επιχείρηση που αναπτύσσει το λογισμικό. Κατ' αρχήν, μετά από αρκετές επισκοπήσεις, κάθε επιχείρηση αναπτύσσει το δικό της ιδιαίτερο κατάλογο σημείων προς έλεγχο, τον οποίο συνεχώς εμπλουτίζει και προσαρμόζει στις ανάγκες του κάθε έργου. Αφού τα ευρήματα της επισκόπησης ανατροφοδοτούνται στους προγραμματιστές, αυτοί τείνουν να τα διορθώσουν με αποτέλεσμα τη βελτίωση της ποιότητας των προγραμμάτων που αναπτύσσουν. Τέλος, η επιτυχία μιας επισκόπησης στην ανεύρεση λαθών μπορεί να είναι τέτοια που να μην χρειάζεται έλεγχος των μονάδων του λογισμικού (unit testing), οπότε η επιχείρηση εξοικονομεί πόρους.

Συγγραφέας (αναθεωρούμενος)	Ο υπεύθυνος ανάπτυξης του σχεδίου ή του λογισμικού που βρίσκεται υπό αναθεώρηση. Είναι επίσης υπεύθυνος για τη διόρθωση των λαθών που θα βρεθούν.
Επόπτης	Βρίσκει λάθη, παραλείψεις ή σημεία ασυμβατότητας σε προγράμματα και αναφορές, ενώ μπορεί και να επισημάνει ζητήματα με σημασία ευρύτερη των στόχων της συγκεκριμένης επισκόπησης.
Αναγνώστης	Διαβάζει τον κώδικα ή την αναφορά σχεδίασης στα οποία γίνεται επισκόπηση.
Καταγραφέας	Καταγράφει τις συζητήσεις και τα αποτελέσματα της συνάντησης.
Πρόεδρος ή Συντονιστής	Οργανώνει και συντονίζει τη διαδικασία, διευκολύνοντας την αναθεώρηση. Αναφέρει τα αποτελέσματα στον αρχι – συντονιστή ή στον υπεύθυνο Ε&Ε.
Αρχι – συντονιστής ή Υπεύθυνος Ε&Ε	Υπεύθυνος για την εφαρμογή και βελτίωση των διαδικασιών Ε&Ε.

Πίνακας Β: Ρόλοι των μελών της ομάδας ελεγκτών

Η συμβολική εκτέλεση (symbolic execution) είναι μια δυναμική τεχνική επικύρωσης της ορθότητας ενός προγράμματος λογισμικού. Σύμφωνα με αυτή, στις μεταβλητές εισόδου του λογισμικού «καταχωρίζονται» συμβολικές τιμές και

έπειτα εκτελείται ο κώδικας του προγράμματος. Η ανάλυση του λογισμικού μπορεί να γίνει χειρωνακτικά ή με αυτοματοποιημένα εργαλεία. Σε κάθε περίπτωση, περιλαμβάνει την παρακολούθηση του τρόπου διάδοσης των συμβολικών τιμών στις υπόλοιπες μεταβλητές του προγράμματος που λαμβάνουν μέρος στους υπολογισμούς που γίνονται στο λογισμικό. Με τον τρόπο αυτό, όλοι οι υπολογισμοί, οι καταχωρίσεις και οι συνθήκες του προγράμματος αποκτούν συμβολικές τιμές. Ακολουθούν δύο σχηματικά παραδείγματα χρήσης της συμβολικής εκτέλεσης[28] που χρησιμοποίησε ο R. Fairley.

Το πρώτο αναφέρεται στην ανάλυση των μονοπατιών εκτέλεσης. Ο αλγόριθμος TEST που ακολουθεί διαβάζει δύο μεταβλητές b και c, με βάση τις οποίες υπολογίζει τις a και x. Έπειτα, χρησιμοποιεί διάφορες συνθήκες στις τιμές των μεταβλητών αυτών για να εκτελέσει ένα από τα τρία τμήματα κώδικα που περιλαμβάνει. Δίπλα στις εντολές του αλγορίθμου παρατίθενται οι αντίστοιχες ενέργειες που πρέπει να κάνουμε κατά τη συμβολική εκτέλεσή του. Έτσι, κατά την ανάγνωση των τιμών των μεταβλητών εισόδου B και C, καταχωρίζουμε σ’ αυτές τις συμβολικές τιμές b και c αντίστοιχα. Έπειτα υπολογίζουμε τις συμβολικές τιμές των μεταβλητών A και X (a και x αντίστοιχα) και προχωρούμε στην περιγραφή των συνθηκών των δομών απόφασης με συμβολικές τιμές. Παρατηρήστε ότι από τη στιγμή που σε κάποια μεταβλητή καταχωρισθεί συμβολική τιμή, αυτή χρησιμοποιείται αντί της μεταβλητής σε όλες τις εντολές όπου συμμετέχει η μεταβλητή.

[28] *R. Fairley (1985), Software Engineering Concepts. McGraw – Hill, Singapore.*

```
ΑΛΓΟΡΙΘΜΟΣ TEST
ΑΡΧΗ
   ΔΙΑΒΑΣΕ (B, C) ;                                      B <— b ; C <— c;
   A : = B+C ;                                           A <— b+c;
   X : = A * C ;                                         X <— (b+c)*c
   ΕΑΝ (A < X) ΤΟΤΕ             {Εξωτερικό ΕΑΝ}          (b+c) <= (b+c)*c
      ΕΚΤΕΛΕΣΕ ΤΜΗΜΑ ΚΩΔΙΚΑ 1
   ΑΛΛΙΩΣ
      ΕΑΝ (B >= 1) OR (B <= – 1) ΤΟΤΕ  {Εσωτερικό ΕΑΝ}    (b >= 1) OR (b <= – 1)
         ΕΚΤΕΛΕΣΕ ΤΜΗΜΑ ΚΩΔΙΚΑ 2
      ΑΛΛΙΩΣ
         ΕΚΤΕΛΕΣΕ ΤΜΗΜΑ ΚΩΔΙΚΑ 3
      ΕΑΝ – ΤΕΛΟΣ               {Εσωτερικό ΕΑΝ}
   ΕΑΝ – ΤΕΛΟΣ                  {Εξωτερικό ΕΑΝ}
ΤΕΛΟΣ.
```

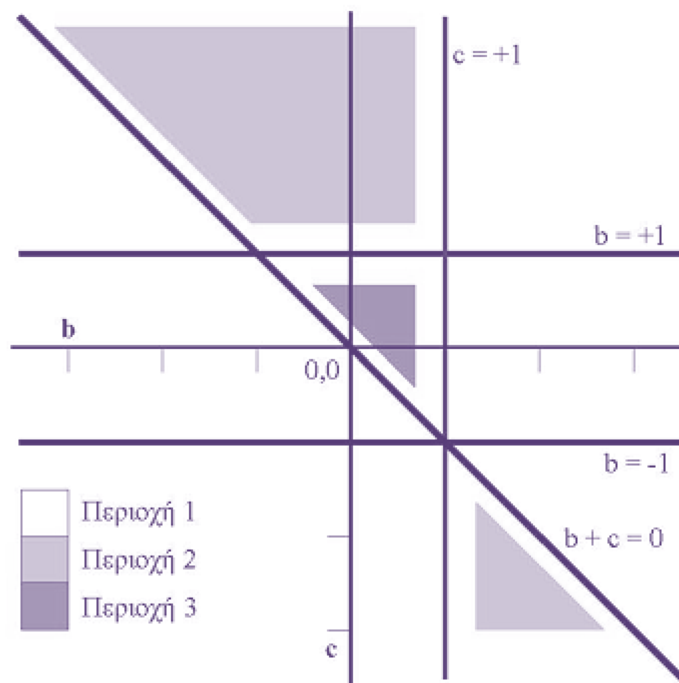
Σύμφωνα με τις αντικαταστάσεις των τιμών των μεταβλητών που έγιναν κατά τη συμβολική εκτέλεση, οι συνθήκες εκτέλεσης των τμημάτων κώδικα είναι οι εξής:

ΤΜΗΜΑ 1: [(b+c) <= (b+c)*c]

ΤΜΗΜΑ 2: [(b+c) > (b+c)*c] AND [(b >= 1) OR (b <= – 1)]

ΤΜΗΜΑ 3: [(b+c) > (b+c)*c] AND [(– 1 < b < 1)]

Στη γραφική παράσταση που ακολουθεί φαίνονται οι περιοχές τιμών των μεταβλητών b και c για τις οποίες εκτελούνται τα τμήματα κώδικα 1, 2 και 3 (περιοχές 1, 2 και 3, αντίστοιχα). Με βάση αυτή τη γραφική παράσταση, μπορούμε να εξάγουμε κατάλληλα σύνολα τιμών για τις μεταβλητές εισόδου, ώστε η εκτέλεση του προγράμματος να ακολουθεί το μονοπάτι που εμείς θέλουμε να ελέγξουμε. Η ανεύρεση δοκιμαστικών δεδομένων (test data) ενός λογισμικού είναι μια αρκετά χρήσιμη εφαρμογή της συμβολικής εκτέλεσης, με την προϋπόθεση ότι οι συνθήκες που καθορίζουν την εκτέλεση του κάθε μονοπατιού περιγράφονται από εξισώσεις πρώτου βαθμού επί των συμβολικών μεταβλητών εισόδου. Επιπλέον, μπορούμε με τον ίδιο τρόπο να ελέγξουμε και την ασφάλεια της εκτέλεσης εντολών διαίρεσης, δεικτοδότησης πινάκων ή αναφοράς σε δείκτες. Για παράδειγμα, στην περιοχή 1 της γραφικής παράστασης μπορεί η διαίρεση με A= b + c να μην είναι ασφαλής, αφού η ευθεία b + c = 0 περιέχεται εκεί.



Περιοχές μεταβλητών b και c του TEST

Το δεύτερο αναφέρεται στην ανάλυση βρόχων, όπου ο αλγόριθμος SUM_ARRAY υπολογίζει το άθροισμα των στοιχείων ενός πίνακα ακεραίων A[1..N]. Χρησιμοποιεί μια δομή επανάληψης (βρόχο) για να διαβάσει κάθε φορά το επόμενο στοιχείο του πίνακα και να το προσθέσει στο μερικό άθροισμα, το οποίο καταχωρεί στη μεταβλητή TOTAL.

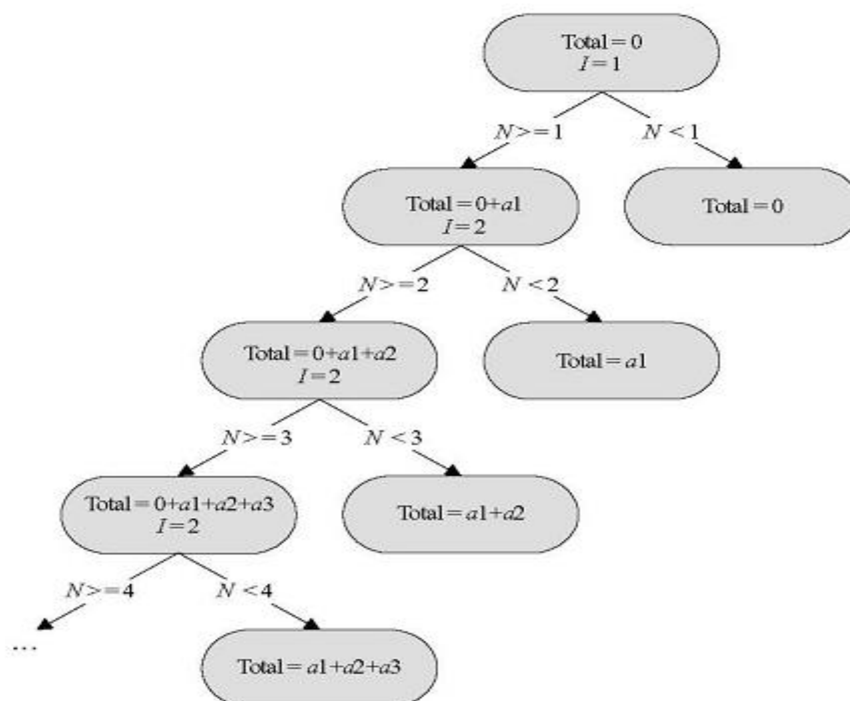
```
ΑΛΓΟΡΙΘΜΟΣ SUM_ARRAY
ΑΡΧΗ
   TOTAL := 0 ;
   ΓΙΑ I := 1 ΕΩΣ N ΚΑΝΕ
      TOTAL := TOTAL + A[I]
   ΕΠΑΝΕΛΑΒΕ
ΤΕΛΟΣ
```

Εάν οι συμβολικές τιμές που δίνουμε στα στοιχεία του πίνακα A είναι
A[1] <— a1, A[2] <— a2, A[3] <— a3, …, A[n] <— an,
τότε στο παρακάτω σχήμα, φαίνεται το δένδρο εκτέλεσης του προγράμματος SUM_ARRAY. Ο συνολικός υπολογισμός που περιγράφεται από το δένδρο συμβολικής εκτέλεσης μπορεί να εκφραστεί σε συμβολική μορφή ως:

$$TOTAL = \sum_{j=1}^{I-1} A[j]$$

Η έκφραση αυτή καλείται αναλλοίωτο του βρόχου (loop invariant) και περιγράφει τη «συμπεριφορά» ενός βρόχου ανεξάρτητα από τις φορές που αυτός επαναλαμβάνεται. Εάν υποθέσουμε ότι αυτό ισχύει στο παράδειγμά μας, τότε μπορούμε να χρησιμοποιήσουμε το αναλλοίωτο του βρόχου για να δείξουμε ότι ο αλγόριθμος SUM_ARRAY επιστρέφει το επιθυμητό αποτέλεσμα: εάν στην έκφραση του αναλλοίωτου αντικαταστήσουμε το I με την τιμή του κατά την έξοδο από το βρόχο (I=N+1), έχουμε :

$$TOTAL = \sum_{j=1}^{I-1} A[j] = \sum_{j=1}^{(N+1)-1} A[j] = \sum_{j=1}^{N} A[j]$$



Αυτή είναι κάθε φορά η τελική τιμή της μεταβλητής TOTAL, η οποία και συμφωνεί με την επιθυμητή τιμή της. Εκτός από γέφυρα προς την τυπική επαλήθευση, η συμβολική εκτέλεση μπορεί να συνδυαστεί με τη δοκιμή ενός λογισμικού, εάν σε κάποιες από τις εισόδους δοθούν αληθινές τιμές ενώ στις υπόλοιπες δοθούν συμβολικές τιμές. Με τον τρόπο αυτό, μπορούμε να πετύχουμε διάφορες διαβαθμίσεις δοκιμών ανάμεσα στην καθαρή συμβολική εκτέλεση (σε όλες τις μεταβλητές εισόδου καταχωρίζονται συμβολικές τιμές) και την καθαρή δοκιμή (σε όλες τις μεταβλητές εισόδου καταχωρίζονται αληθινές τιμές). Για παράδειγμα, εάν θέλουμε να ελέγξουμε την ευαισθησία ενός προγράμματος σε

κάποια συγκεκριμένη μεταβλητή εισόδου, χρησιμοποιούμε συμβολική τιμή για τη μεταβλητή αυτή και αληθινές τιμές για τις υπόλοιπες.

Μια από τις τελευταίες γραμμές άμυνας απέναντι στα σφάλματα είναι η προσομοίωση (simulation) της εκτέλεσης ενός λογισμικού. Η τεχνική αυτή είναι αρκετά αποτελεσματική, αλλά και δαπανηρή, γι' αυτό και χρησιμοποιείται στα τελευταία στάδια της φάσης ανάπτυξης ή ελέγχου του λογισμικού. Βέβαια, οι δοκιμές προσομοίωσης είναι συμφέρουσες μόνο όταν είναι πιο φθηνές από το κόστος επισκευής του, συστήματος, αφού αυτό παραδοθεί (δηλαδή τη συντήρηση του συστήματος). Όταν λέμε προσομοίωση, εννοούμε κάποιου είδους προσέγγιση της πραγματικότητας. Πιο συγκεκριμένα, θεωρούμε ότι η ανάπτυξη του λογισμικού έχει προχωρήσει αρκετά και το σύστημα βρίσκεται σε προχωρημένο στάδιο συγκρότησης, οπότε μπορούμε να δοκιμάσουμε το σύστημα με τρόπο που προσεγγίζει τις πραγματικές συν- θήκες στις οποίες αυτό θα λειτουργήσει όταν παραδοθεί[29] . Σε μια δοκιμή προσομοίωσης πρέπει να προσεγγισθεί τόσο το πραγματικό λογισμικό, όσο και το πραγματικό υλικό όπου θα εκτελείται το λογισμικό, ώστε να δοκιμαστούν διάφορες πραγματικές περιπτώσεις χρήσης του συστήματος. Πολλές φορές, τμήματα του υλικού ή του λογισμικού λείπουν ή δεν έχουν ακόμη καθοριστεί επακριβώς. Πριν αυτά υλοποιηθούν προσομοιώνεται η λειτουργία τους, ώστε να δοκιμαστεί η συμπεριφορά τους αλλά και η συμπεριφορά του συνολικού συστήματος. Πολλές φορές, οι δοκιμές προσομοίωσης ενός συστήματος ξεκινούν από τις αρχικές φάσεις της ανάπτυξης και διαρκούν μέχρι την παράδοση του συστήματος στον πελάτη. Οι αρχικές δοκιμές γίνονται στις εγκαταστάσεις ανάπτυξης του συστήματος. Είναι φυσικό στη φάση αυτή πολλά τμήματα του λογισμικού να μην έχουν ακόμη υλοποιηθεί (για τα τμήματα αυτά χρησιμοποιείται ειδικός κώδικας - λέγεται stub - ο οποίος εξομοιώνει την αναμενόμενη εξωτερική συμπεριφορά του τμήματος χωρίς να έχει υλοποιηθεί η εσωτερική επεξεργασία που προκαλεί αυτή τη συμπεριφορά, ώστε το συνολικό σύστημα να λειτουργεί). Καθώς αναπτύσσονται τα τμήματα που λείπουν, ενσωματώνονται στα υπάρχοντα και αντικαθιστούν τα stubs, με αποτέλεσμα οι προσομοιώσεις να είναι πληρέστερες. Η τελική προσομοίωση γίνεται εγκαθιστώντας το σύστημα στο υλικό του πελάτη, όπου και λειτουργεί δοκιμαστικά για μερικούς μήνες μέχρι να παραδοθεί τελικά.

[29]*M.L. Shooman (1983), Software Engineering. McGraw – Hill, Tokyo.*

Από την άλλη, η ανάλυση ευαισθησίας (sensitivity analysis) περιγράφει ποσοτικά τη συμπεριφορά του συστήματος λογισμικού σε σχέση με την πιθανότητα να υπάρχουν κρυμμένα σφάλματα. Περιλαμβάνει την επαναλαμβανόμενη εκτέλεση του αρχικού κώδικα του προγράμματος, αλλά και παραλλαγών αυτού, χρησιμοποιώντας δύο υποθέσεις (Voas[95]):

• την υπόθεση μοναδικού σφάλματος, η οποία ισχυρίζεται ότι το λογισμικό περιέχει ένα μόνο σφάλμα και όχι πολλά σφάλματα κατανεμημένα σε όλο τον κώδικα, και

• την υπόθεση απλού σφάλματος, η οποία ισχυρίζεται ότι το σφάλμα βρίσκεται σε μια και μοναδική θέση μέσα στον κώδικα, χωρίς να είναι κατανεμημένο σε ολόκληρο τον κώδικα και το σφάλμα αυτό έχει την ίδια πιθανότητα να βρίσκεται σε οποιαδήποτε θέση μέσα στον κώδικα.

Αυτές οι υποθέσεις αποτελούν παραλλαγή της υπόθεσης του ικανού προγραμματιστή, η οποία ισχυρίζεται ότι ένας ικανός προγραμματιστής θα γράψει κώδικα που είναι αρκετά κοντά στο να χαρακτηριστεί «ορθός».

Ο στόχος της ανάλυσης ευαισθησίας είναι να υποδείξει πόσο «μικρά» μπορεί να είναι τα μικρότερα σφάλματα ενός λογισμικού. Έχοντας αυτή την πρόβλεψη, μπορούμε να εφαρμόσουμε στατιστικές τεχνικές για να προσδιορίσουμε πόσες δοκιμές χρειάζονται για να ανακαλυφθούν σφάλματα αυτού του μεγέθους και έτσι να έχουμε ένα κριτήριο τερματισμού των δοκιμών. Το πλεονέκτημα της μεθόδου είναι ότι η πρόβλεψη βασίζεται στην παρατήρηση αποτελεσμάτων που οφείλονται στην ύπαρξη πραγματικών σφαλμάτων. Η αδυναμία της έγκειται στο γεγονός ότι τα σφάλματα αυτά αποτελούν ένα μικρό υποσύνολο όλων των πιθανών λαθών.

Η ανάλυση ευαισθησίας εκτιμά την πιθανότητα αποτυχημένης λειτουργίας του λογισμικού εξαιτίας ενός μοναδικού λάθους που θα μπορούσε να συμβεί σε ένα συγκεκριμένο σημείο του κώδικα, για κάθε σημείο του προγράμματος .

Για να παράσχει αυτή την πρόβλεψη, η ανάλυση ευαισθησίας εισάγει εσκεμμένα σφάλματα μέσα στον κώδικα και εκτιμά την ορατή επίδρασή τους (που είναι η αστοχία – failure – του λογισμικού). Η επιτυχία της εξαρτάται πολύ από τον τρόπο που έγιναν οι δοκιμές για να μετρηθεί η επίδραση: στη χειρότερη περίπτωση πρέπει να ελεγχθούν όλα τα «σημεία» του κώδικα που αλλάζουν τις τιμές κάποιων μεταβλητών, δεδομένων, αρχείων ή ακόμη και του μετρητή εντολών προγράμματος. Για να συμβεί και να παρατηρηθεί μια αστοχία κατά την εκτέλεση του λογισμικού, πρέπει να συμβούν τρία πράγματα: πρέπει να εκτελεστεί το σφάλμα που την προκαλεί, να βρεθεί το λογισμικό σε μια κατάσταση δεδομένων

που δεν είναι ορθή, η οποία να διαδοθεί μέχρι κάποια παρατηρούμενη έξοδο του λογισμικού. Η ανάλυση ευαισθησίας διαχωρίζει τρία αντίστοιχα είδη γεγονότων που μπορεί να προκαλέσουν αστοχία του λογισμικού. Με στόχο να εκτιμηθεί η πιθανότητα εμφάνισης κάθε τέτοιου γεγονότος, χρησιμοποιεί τις εξής διαφορετικές διαδικασίες ανάλυσης για κάθε σημείο του προγράμματος:

• Ανάλυση εκτέλεσης (execution analysis): Εκτιμά την πιθανότητα εκτέλεσης ενός σημείου του κώδικα, εκτελώντας το συνολικό λογισμικό πολλές φορές. Κάθε φορά, χρησιμοποιούνται δεδομένα εισόδου που επιλέγονται από ένα σύνολο δεδομένων που έχει μια συγκεκριμένη κατανομή. Μετά την εκτέλεση του δοκιμαζόμενου σημείου, καταγράφεται η κατάσταση δεδομένων (αυτή είναι η «αρχική κατάσταση δεδομένων»)

• Ανάλυση μόλυνσης (infection analysis): Εάν ένα σημείο κώδικα περιέχει ένα σφάλμα και εάν το σημείο αυτό εκτελεστεί κατά την εκτέλεση του προγράμματος με ένα σύνολο δεδομένων εισόδου, τότε το σφάλμα μπορεί να διαταράξει την ορθότητα της κατάστασης των δεδομένων. Τότε, η κατάσταση δεδομένων για το συγκεκριμένο σύνολο δεδομένων εισόδου θεωρείται «μολυσμένη». Για να εκτιμήσει την πιθανότητα μόλυνσης, ο αλγόριθμος ανάλυσης ευαισθησίας εφαρμόζει μια σειρά συντακτικών παραλλαγών σε κάθε σημείο. Ως συντακτική παραλλαγή ορίζεται η αλλαγή από την αρχική σύνταξη σε μια νέα σύνταξη που είναι γραμματικά ορθή και έχει διαφορετικό νόημα για ένα τουλάχιστον δεδομένο εισόδου. Μετά από κάθε παραλλαγή, το λογισμικό εκτελείται ξανά με τυχαία δεδομένα εισόδου και συγκρίνεται η κατάσταση δεδομένων αμέσως μετά την εκτέλεση του δοκιμαζόμενου σημείου, με την αρχική κατάσταση δεδομένων. Εάν οι δύο καταστάσεις διαφέρουν, τότε έχει συμβεί «μόλυνση».

• Ανάλυση διάδοσης (propagation analysis): Μετά την εκτέλεση του δοκιμαζόμενου σημείου, τροποποιούμε την παραγόμενη κατάσταση δεδομένων δίνοντας μια τυχαία τιμή (από ένα σύνολο τιμών με συγκεκριμένη κατανομή) σε ένα αντικείμενο δεδομένων. Έπειτα συνεχίζεται η εκτέλεση του προγράμματος μέχρι να παραχθεί κάποια έξοδος, την οποία συγκρίνουμε με την έξοδο που θα είχε παραχθεί, εάν δεν είχε γίνει τροποποίηση της κατάστασης δεδομένων. Εάν αυτές διαφέρουν, τότε το σφάλμα έχει διαδοθεί.

Βλέποντας πάλι τον αλγόριθμο TEST που χρησιμοποίησε ο R. Fairley ,το οποίο τροποποίησε ,για τις ανάγκες του εδώ παραδείγματος ,ώστε να εκτελείται μόνο εάν τουλάχιστον ένας από τους B και C είναι θετικός αριθμός.

```
ΑΛΓΟΡΙΘΜΟΣ TEST – 1
ΑΡΧΗ
  ΔΙΑΒΑΣΕ (B, C) ;
  ΕΑΝ (B>0) OR (C>0) ΤΟΤΕ                      {Αρχικό ΕΑΝ}
  ΑΡΧΗ
    A := B+C ;
    X := A * C ;
    ΕΑΝ (A < X) ΤΟΤΕ                            {Εξωτερικό ΕΑΝ}
      ΕΚΤΕΛΕΣΕ ΤΜΗΜΑ ΚΩΔΙΚΑ 1
    ΑΛΛΙΩΣ
      ΕΑΝ (B >= 1) OR (B <= – 1) ΤΟΤΕ           {Εσωτερικό ΕΑΝ}
        ΕΚΤΕΛΕΣΕ ΤΜΗΜΑ ΚΩΔΙΚΑ 2
      ΑΛΛΙΩΣ
        ΕΚΤΕΛΕΣΕ ΤΜΗΜΑ ΚΩΔΙΚΑ 3
      ΕΑΝ – ΤΕΛΟΣ                               {Εσωτερικό ΕΑΝ}
    ΕΑΝ – ΤΕΛΟΣ                                 {Εξωτερικό ΕΑΝ}
  ΤΕΛΟΣ
  ΕΑΝ – ΤΕΛΟΣ                                   {Αρχικό ΕΑΝ}
ΤΕΛΟΣ
```

Ας προσπαθήσουμε να προσδιορίσουμε την πιθανότητα αστοχίας του προγράμματος, εάν συμβεί λάθος στην εντολή Α : = Β + C, εφαρμόζοντας τις τρεις διαδικασίες της ανάλυσης ευαισθησίας. Κατά την ανάλυση εκτέλεσης, παρατηρούμε ότι η εκτέλεση αυτής της γραμμής κώδικα εξαρτάται από την τιμή των Β και C: εάν και οι δύο αριθμοί είναι αρνητικοί, αυτή δεν εκτελείται. Ας υποθέσουμε στο παράδειγμά μας ότι από τα 100 δεδομένα δοκιμής (τα οποία είναι ζεύγη τιμών για τους Β και C) που έχουμε επιλέξει, μόνο σε 10 από αυτά οι δύο αριθμοί έχουν αρνητικές τιμές. Συνεπώς, η εντολή εκτελείται στο 90% των περιπτώσεων δοκιμής, οπότε η πιθανότητα εκτέλεσης είναι Pε=0,9. Για να εφαρμόσουμε τις άλλες δύο διαδικασίες, πρώτα εισάγουμε ένα σφάλμα στο υπό δοκιμή σημείο του κώδικα: τροποποιούμε την εντολή Α := Β+C σε Α := Β – C. Ας εξετάσουμε τη συμπεριφορά του προγράμματος για 2 από τα 100 δεδομένα δοκιμής:

Ζεύγος δεδομένων Α: (B=6, C=2)		Ζεύγος Δεδομένων Β: (B= – 1, C=3)	
Αναμενόμενη κατάσταση	Παρατηρούμενη κατάσταση	Αναμενόμενη κατάσταση	Παρατηρούμενη κατάσταση
$A = 8$ $X = 16$	$A=4$ $X = 8$	$A=2$ $X=6$	$A= -4$ $X= -12$
Τμήμα κώδικα που εκτελείται		Τμήμα κώδικα που εκτελείται	
ΤΜΗΜΑ 1	ΤΜΗΜΑ 1	ΤΜΗΜΑ 1	ΤΜΗΜΑ 2

Παρατηρούμε λοιπόν τα εξής:

• Και στις δύο περιπτώσεις δοκιμής, προκλήθηκε σφάλμα στην κατάσταση δεδομένων του προγράμματος, αφού οι μεταβλητές A και X βρέθηκαν να έχουν διαφορετική τιμή από την αναμενόμενη στην ορθή κατάσταση. Συνεπώς, η κατάσταση δεδομένων είναι και στις δύο περιπτώσεις μολυσμένη.
• Στην πρώτη περίπτωση δοκιμής, η μόλυνση δεν διαδόθηκε, αφού δεν προκλήθηκε αλλαγή στην παρατηρούμενη έξοδο του προγράμματος (και τις δύο φορές εκτελέσθηκε το ΤΜΗΜΑ 1).
• Στη δεύτερη περίπτωση, εξαιτίας της μόλυνσης εκτελέσθηκε το ΤΜΗΜΑ 2 αντί του ΤΜΗΜΑΤΟΣ 1, οπότε συμπεραίνουμε ότι η μόλυνση διαδόθηκε. Κατά την ανάλυση μόλυνσης θα δοκιμάσουμε το «λανθασμένο» πρόγραμμα με όλα τα 100 δεδομένα εισόδου και θα μετρήσουμε την πιθανότητα μόλυνσης Pμ ως το πλήθος των μολυσμένων καταστάσεων που παρατηρήθηκαν.

Κατά την ανάλυση διάδοσης, εκτελούμε το πρόγραμμα με την τροποποιημένη κατάσταση δεδομένων και μετρούμε την πιθανότητα επίδρασης της τροποποίησης στις εξόδους του προγράμματος Pδ ως το πλήθος των μη αναμενόμενων εξόδων που παρατηρήθηκαν. Το αποτέλεσμα της ανάλυσης ευαισθησίας είναι η εκτιμώμενη πιθανότητα αστοχίας του προγράμματος, εάν υπήρχε σφάλμα στη συγκεκριμένη εντολή, και προκύπτει πολλαπλασιάζοντας τους μέσους όρους των πιθανοτήτων Pε, Pμ και Pδ. Εάν πολλαπλασιάσουμε τις ελάχιστες τιμές των τριών πιθανοτήτων, τότε βρίσκουμε το κάτω όριο της πιθανότητας αστοχίας του προγράμματος εξαιτίας ενδεχόμενου λάθους στην εντολή αυτή.

Η ανάλυση ευαισθησίας βοηθά στον προσδιορισμό του αριθμού δοκιμών που απαιτούνται ώστε να πεισθούμε ότι σε ένα σύστημα λογισμικού δεν υπάρχουν κρυμμένα λάθη. Επιπλέον, βοηθά στην ανεύρεση τμημάτων κώδικα που έχουν πολύ μικρή πιθανότητα ελέγχου, ώστε να γίνουν περισσότερο συστηματικές δοκιμές τους. Παρ' όλο που και οι δύο αρχικές υποθέσεις περιορίζουν το πλήθος των κατηγοριών σφαλμάτων στα οποία μπορεί να εφαρμοστεί η ανάλυση ευαισθησίας, η πολυπλοκότητα των τριών διαδικασιών ανάλυσης είναι ανάλογη με το τετράγωνο των σημείων κώδικα που εξετάζονται. Χωρίς τους δύο αρχικούς περιορισμούς, η πολυπλοκότητα αυτή είναι μη ιχνηλατήσιμη (intractable).

Οι δοκιμές των μονάδων του λογισμικού (unit testing) επικεντρώνονται στα μικρότερα τμήματα του λογισμικού και στοχεύουν να δείξουν ότι καθένα από αυτά λειτουργεί σωστά και ικανοποιεί τις προδιαγραφές του. Για τη σχεδίαση των

δοκιμών χρησιμοποιούνται οι προδιαγραφές και ο κώδικας κάθε τμήματος και εφαρμόζονται τεχνικές διαφανούς κουτιού.

Η πολυπλοκότητα της σχεδίασης των περιπτώσεων ελέγχου ενός τμήματος είναι σχετικά μικρή, εξαιτίας του μικρού μεγέθους του κώδικα και της μειωμένης πολυπλοκότητας της επεξεργασίας που εκτελεί ένα (καλά σχεδιασμένο) τμήμα. Η πολυπλοκότητα αυτή μειώνεται ακόμη περισσότερο όταν η σχεδίαση εμφανίζει υψηλή συνοχή (στην ιδανική περίπτωση, όταν κάθε τμήμα εκτελεί μια μόνο λειτουργία). Αντίστοιχα όμως, είναι περιορισμένη και η εμβέλεια των δοκιμών που μπορούμε να εφαρμόσουμε σε ένα τμήμα, ενώ το ίδιο μικρή είναι και η σημασία των λαθών που θα ανακαλύψουμε. Αυτό όμως δεν πρέπει να μας οδηγήσει σε επιφανειακές δοκιμές των μονάδων του λογισμικού, καθώς το κόστος διόρθωσης κάθε ελαττώματος που θα ξεφύγει από αυτή τη φάση γίνεται στη συνέχεια των δοκιμών πολύ μεγάλο έως απαγορευτικό.

Στις δοκιμές μονάδων περιλαμβάνονται:

• η δοκιμή της διεπαφής του τμήματος, ώστε να διασφαλιστεί ότι οι πληροφορίες εισέρχονται στο τμήμα και εξέρχονται από αυτό σωστά,

• η δοκιμή των τοπικών (εσωτερικών) δομών δεδομένων, ώστε να διασφαλιστεί ότι το τμήμα, κατά την εκτέλεση των λειτουργιών του, δεν παραβιάζει την ακεραιότητα των δεδομένων που διατηρεί προσωρινά,

• η δοκιμή των οριακών συνθηκών, ώστε να διασφαλιστεί ότι το τμήμα λειτουργεί σωστά κοντά στα όρια επεξεργασίας που του έχουν τεθεί,

• η δοκιμή βασικών μονοπατιών εκτέλεσης, ώστε η εκτέλεση κάθε εντολής του κώδικα του τμήματος να δοκιμαστεί τουλάχιστον μια φορά,

• η δοκιμή του χειρισμού λαθών που περιλαμβάνεται στο τμήμα, δηλαδή δοκιμαστική εκτέλεση του κώδικα που εκτελείται όταν παρουσιαστούν σφάλματα.

5. Ποιοτικές, μετρήσεις και μετρικές λογισμικού

Σκοπός του κεφαλαίου είναι μια διεξοδική ανάλυση ορισμένων μετρήσεων, που πραγματοποιούνται στην διαδικασία λειτουργίας ενός συστήματος ποιότητας λογισμικού, με τη χρήση μετρικών και η παρουσίαση επιλεγμένων μετρικών καθώς και τρόπων μέτρησης.

Η ποιότητα λογισμικού είναι συνυφασμένη με την έννοια των μετρήσεων. Αναφέραμε ότι αντικείμενο των μετρήσεων δεν είναι η μέτρηση των οντοτήτων, αλλά η μέτρηση των ιδιοτήτων των οντοτήτων. Επίσης, δώσαμε τον ορισμό της μέτρησης (measurement). Στη βιβλιογραφία αναφέρονται οι όροι μετρική (metric) και μέτρο (measure). Αν και πολλά βιβλία θεωρούν τους όρους ισοδύναμους, συνήθως ο όρος μετρική[30] χρησιμοποιείται για να χαρακτηρίσει απλές ιδιότητες, όπως για παράδειγμα το μήκος γραμμών κώδικα, ή η κυκλωματική πολυπλοκότητα (που θα περιγραφεί στα πλαίσια αυτής της ενότητας), ενώ ο όρος μέτρο χρησιμοποιείται για συσχετισμούς ιδιοτήτων και για την πρόβλεψη πιο πολύπλοκων χαρακτηριστικών όπως για παράδειγμα το κόστος, ή η πολυπλοκότητα ενός τμήματος κώδικα. Παραδείγματος χάριν αναφέρουμε ότι: «η μετρική V(g) είναι ένα μέτρο της πολυπλοκότητας». Τιμές σε μετρικές ανατίθενται με τη διαδικασία της μέτρησης, ενώ συσχετισμοί μετρικών και αποτελέσματα που προκύπτουν από μετρήσεις παρέχουν πληροφορίες για κάποια μέτρα.

Η χρήση μέτρων και μετρικών, στην τεχνολογία λογισμικού, έρχεται να λύσει το βασικό πρόβλημα του λογισμικού, που σχετίζεται με την αδυναμία καθορισμού μετρήσιμων ποσοτήτων, κατά τη σχεδίαση και ανάπτυξη ενός έργου. Υποσχόμαστε, για παράδειγμα, πως ένα πρόγραμμα θα είναι «αξιόπιστο», «φιλικό προς το χρήστη» και «συντηρήσιμο», χωρίς να προσδιορίζουμε με μετρήσιμες ποσότητες τι σημαίνει κάθε ένα από αυτά. Έτσι, σε πολλά έργα, η αποτυχία επίτευξης των στόχων τους είναι φυσιολογική συνέπεια. Όπως ορίζει ο Gilb[30]: «το μόνο ξεκάθαρο για έργα που δεν έχουν ξεκάθαρους στόχους, είναι πως δε θα πετύχουν τους στόχους τους». Επίσης, αποτυγχάνουμε να εκτιμήσουμε βασικά ποσοτικά χαρακτηριστικά που σχετίζονται με το έργο, όπως για παράδειγμα το χρόνο που απαιτείται για τη μεταφορά σε κάποιο άλλο σύστημα, ή την αξιοπιστία του συστήματος χρησιμοποιώντας όρους όπως είναι ο «αριθμός αποτυχιών σε δεδομένη χρονική περίοδο». Αντίθετα, αν κανείς δει τις διαφημίσεις εμπορικού

[30] *T. Gilb, «Principles of Software Engineering Management», Addison Wesley, (1987).*

λογισμικού θα διαβάσει πληθώρα διαφημιστικών σλόγκαν που μιλάνε για «μείωση του χρόνου συντήρησης κατά 80%», «αυξημένη αξιοπιστία κατά 75%» κτλ, χωρίς την παραμικρή ένδειξη για το πώς προκύπτουν αυτοί οι αριθμοί και βάσει ποιων μετρικών.

Οι πρώτες μετρικές λογισμικού παρουσιάστηκαν προς το τέλος της δεκαετίας του ‘70. Εκείνη την περίοδο δημοσιεύθηκαν οι πρώτες εργασίες σχετικά με τις μετρικές λογισμικού, όπως αυτή του Halstead[31]και του McCabe[32]. Οι δύο αυτές πρωτοποριακές εργασίες στον τομέα της εξασφάλισης ποιότητας, θεωρούνται ως το έναυσμα για τη θεωρία των μετρήσεων και μετρικών και είναι σίγουρα οι εργασίες με τις περισσότερες αναφορές στην επιστήμη της ποιότητας λογισμικού. Παρόλα αυτά, καμία από τις δύο δεν παρουσίαζε τις μετρικές με στόχο τη διασφάλιση ποιότητας, αφού οι μετρικές του Halstead είχαν ως πρωταρχικό σκοπό την εκτίμηση του μεγέθους μελλοντικών έργων και η εργασία του McCabe την ανάλυση της φάσης ελέγχου των μονοπατιών του κώδικα. Η αξία και των δύο επιβεβαιώθηκε από τις πρώτες συσχετίσεις[33][34] των μετρήσεών τους με στοιχεία όπως οι αναφορές λαθών (defect reports). Τις δύο αυτές μετρικές ακολούθησε ένας μεγάλος αριθμός μετρικών που προτάθηκαν τα επόμενα χρόνια και συνεχίζουν να προτείνονται ακόμα και σήμερα, οι οποίες μετρούν σχεδόν τα πάντα.

Ο βασικότερος στόχος όλων των μετρικών που διεξάγονται σε ένα λογισμικό είναι η διασφάλιση της ποιότητάς του. Για να επιτευχθεί αυτό, όμως, θα πρέπει να υπάρχει ένα «πλάνο ποιότητας» συνολικά σε ολόκληρο το έργο ή – ακόμη καλύτερα – σε ολόκληρη την εταιρεία. Έτσι, η διασφάλιση της ποιότητας, σε συνδυασμό με τις μετρικές που χρησιμοποιούνται, μπορεί να τεθεί από διαφορετικές πλευρές:

- διασφάλιση ποιότητας προϊόντος (product quality assurance): στόχος είναι η ποιότητα του προϊόντος και κατ’ επέκταση η ικανοποίηση του χρήστη.
- διασφάλιση ποιότητας έργου (project quality assurance): που στοχεύει α) στην αποτίμηση της κατάστασης του έργου, β) την επίλυση προβλημάτων πριν αυτά γίνουν «κρίσιμα» και επηρεάσουν την πορεία του έργου γ) τη ρύθμιση και το

[31]*M. H. Halstead, «Elements of Software Science», Elsevier Publications, N – Holland, (1975)*

[32] *T. J. McCabe, «A Complexity Measure», IEEE Transactions in Software Engineering SE – 2(4), (1976)*

[33] *Y. Funami and M. Halstead, «A Software Physics Analysis of Akiyama’sDebugging Data», Symposium onComputer Software Engineering, (1976)*

[34] *A. Fitzsimmons and T. Love, «A Review and Evaluation of Software Science», Computing Surveys, Volume 10, 1, (1978).*

διαχωρισμό των εργασιών και δ) την εκτίμηση της ικανότητας των ομάδων να παράγουν «ποιοτικό» λογισμικό.

- διασφάλιση ποιότητας διαδικασιών (process quality assurance): που βοηθά την εταιρεία να έχει γνώση για την αποτελεσματικότητα των διαδικασιών.

Πιο συγκεκριμένα, η μέτρηση κάποιων χαρακτηριστικών του λογισμικού είναι μια διαδικασία απαραίτητη για την εκτίμηση της κατάστασης των έργων, των προϊόντων, των διαδικασιών και των πόρων παραγωγής λογισμικού. Κάθε μέτρηση, όμως, πρέπει να εξυπηρετεί μια συγκεκριμένη ανάγκη, η οποία έχει σαφώς κατανοηθεί και καθοριστεί.

Στην πλειοψηφία τους οι μετρικές που προτάθηκαν περιορίζονται στη μέτρηση μεμονωμένων εσωτερικών χαρακτηριστικών (internal characteristics) του λογισμικού, χωρίς να προτείνουν τρόπους συσχέτισης με τα εξωτερικά χαρακτηριστικά (external characteristics) του λογισμικού που η μέτρησή τους είναι ο σκοπός του συστήματος ποιότητας λογισμικού. Παράδειγμα εσωτερικού χαρακτηριστικού είναι οι γραμμές κώδικα (LOC), για τις οποίες μιλήσαμε στο 2ο κεφάλαιο. Τα εσωτερικά χαρακτηριστικά μετρούνται εύκολα, αλλά δεν προσφέρουν πληροφορίες υψηλού επιπέδου που σχετίζονται με την ποιότητα του προϊόντος. Παραδείγματα εξωτερικών χαρακτηριστικών είναι η αξιοπιστία και η ευχρηστία. Αυτά τα χαρακτηριστικά είναι δύσκολο έως αδύνατο να μετρηθούν άμεσα. Ακριβώς επειδή η χρήση των μετρικών γινόταν με στόχο τη μέτρηση των εσωτερικών χαρακτηριστικών, χωρίς να υπάρχει σύνδεση των μετρήσεων με τα εξωτερικά χαρακτηριστικά, αυτό είχε ως αποτέλεσμα, κάποιες φορές, η χρήση των μετρικών να γίνεται χωρίς σκοπό και όπως αναφέρει ο Hambling[35] , με το σκεπτικό «να μετρήσουμε οτιδήποτε, με την ελπίδα ότι κάποιες τουλάχιστον από τις μετρήσεις θα είναι χρήσιμες». Έλειπε δηλαδή αυτό που ορίζουμε ως ερμηνεία μετρικής (metric interpretation), δηλαδή η χρήση των αποτελεσμάτων της μέτρησης για την εξαγωγή συμπερασμάτων για κάποια από τα εξωτερικά χαρακτηριστικά του λογισμικού.

[35] *Brian Hambling, «Managing Software Quality», McGraw–Hill, (1996).*

5.1 Κατηγορίες Μετρικών

Υπάρχουν διάφορες κατηγοριοποιήσεις των μετρικών. Δύο από τις πιο διαδεδομένες κατηγοριοποιήσεις είναι:

α) ανάλογα με τα χαρακτηριστικά που μετρούν, οπότε κατηγοριοποιούνται ως εσωτερικές και εξωτερικές μετρικές, και

β) ανάλογα με το στόχο του συστήματος ποιότητας με τον οποίο σχετίζονται, οπότε κατηγοριοποιούνται ως μετρικές διαδικασίας, μετρικές πόρων και μετρικές προϊόντος.

Σύμφωνα με την πρώτη κατηγοριοποίηση, αντίστοιχα με τα εσωτερικά και τα εξωτερικά ποιοτικά χαρακτηριστικά, ορίζουμε τις εσωτερικές μετρικές (internal metrics) και τις εξωτερικές μετρικές (external metrics). Έτσι, οι εσωτερικές μετρικές μετρούν χαρακτηριστικά όπως οι γραμμές κώδικα, το ποσοστό σχολίων στον κώδικα, ο αριθμός των «goto» εντολών σε μία συνάρτηση, το βάθος δέντρου κληρονομικότητας, κτλ. Οι εσωτερικές μετρικές μπορούν να υπολογιστούν εύκολα, αλλά το ζητούμενο δεν είναι τόσο ο τρόπος μέτρησής τους, όσο η σύνδεσή τους με τα εξωτερικά χαρακτηριστικά του λογισμικού τα οποία επιθυμούμε να μετρήσουμε, δηλαδή η ερμηνεία των εσωτερικών μετρικών.

Αντίθετα με τις εσωτερικές μετρικές, οι εξωτερικές μετρικές δεν ικανοποιούν απόλυτα τον ορισμό της μετρικής, αφού εμπεριέχουν –συνήθως– το στοιχείο της υποκειμενικότητας. Αυτές οι μετρικές δεν μπορούν, εκ των πραγμάτων, στην πλειοψηφία των περιπτώσεων να βασιστούν σε μετρήσεις φυσικών ποσοτήτων (όπως στο παράδειγμα των εσωτερικών μετρικών), αλλά βασίζονται σε έρευνες για τη γνώμη των πελατών (συνήθως με χρήση ερωτηματολογίων ή συνεντεύξεων), σε παρατήρηση του χρήστη και μελέτη της συμπεριφοράς του, κτλ. Αν και σύμφωνα με τον ορισμό της μετρικής που προηγήθηκε αυτές δεν είναι τυπικά μετρικές, σε αρκετά βιβλία[36][37][38]τις αναφέρουν και τις χρησιμοποιούν ως μετρικές ποιότητας . Στα πλαίσια της εργασίας αυτής, χρησιμοποιούμε τον όρο μετρικές τόσο για τις εσωτερικές μετρικές, όσο και για τις εξωτερικές μετρικές, όμως δεν πρέπει να αγνοούνται οι ιδιαιτερότητες των εξωτερικών μετρικών.

---

[36] *S. D. Conte et al., «Software Engineering Metrics and Models», Benjamin Cummings, (1986)*
[37] *C. Jones, «Applied Software Measurement: Assuring Productivity and Quality», McGraw Hill, (1991)*
[38] *C. Kaplan et al., «Secrets of Software Quality», McGraw Hill, (1995)*

Σύμφωνα με τη δεύτερη κατηγοριοποίηση, διακρίνουμε τις μετρικές σε μετρικές διαδικασίας (process metrics), μετρικές πόρων (resource metrics) και μετρικές προϊόντος (product metrics). Οι μετρικές διαδικασίας μας δίνουν απαντήσεις σε ερωτήματα σχετικά με το χρόνο που χρειάστηκε μία δραστηριότητα για να ολοκληρωθεί, πόσο θα κοστίσει, πώς συγκρίνεται με εναλλακτικές διαδικασίες που θα μπορούσαν να είχαν επιλεγεί, κτλ. Ένας μικρός μόνο αριθμός των χαρακτηριστικών της διαδικασίας μπορεί να μετρηθεί άμεσα (με χρήση εσωτερικών μετρικών). Τέτοια χαρακτηριστικά για παράδειγμα είναι η διάρκεια μιας διαδικασίας, η προσπάθεια (σε ανθρωπομήνες) για την ολοκλήρωσή της καθώς και ο αριθμός γεγονότων καθορισμένου τύπου που συνέβησαν κατά τη διάρκειά της. Τέτοια γεγονότα μπορεί, για παράδειγμα, να είναι ο αριθμός των λαθών που εντοπίστηκαν κατά τη διάρκεια μίας ώρας ελέγχου μονάδας, οαριθμός των συναντήσεων με τον πελάτη που απαιτήθηκαν για την ολοκλήρωση των προδιαγραφών, ο αριθμός των αναφορών λαθών που ελήφθησαν από Ν επιλεγμένους πελάτες κατά τη διάρκεια Μ εβδομάδων ελέγχου βήτα (beta testing), κτλ.

Σε αντίθεση με τα παραπάνω χαρακτηριστικά, ένας αριθμός χαρακτηριστικών της διαδικασίας μπορεί να μετρηθεί μόνο έμμεσα (είναι λοιπόν εξωτερικά χαρακτηριστικά). Τέτοια χαρακτηριστικά για παράδειγμα είναι η σταθερότητα της διαδικασίας, η δυνατότητα παρακολούθησής της από τον Υπεύθυνο έργου, κτλ. Με τη χρήση μετρικών διαδικασιών σχετίζεται και ο στατιστικός έλεγχος ποιότητας. Τεχνικές στατιστικού ελέγχου στο λογισμικό[39] εφαρμόζουν προκειμένου να μετρήσουν γεγονότα που σχετίζονται με μία διαδικασία και μπορούν να οδηγήσουν σε εκτιμήσεις για το αν αυτή η διαδικασία είναι εντός ή εκτός ελέγχου. Τα πιο τυπικά παραδείγματα τέτοιων γεγονότων είναι τα λάθη (defects) ανά διαδικασία ανάπτυξης, τα προβλήματα ανά εβδομάδα, η συσχέτιση των γραμμών κώδικα ανά ώρα με τις επισκοπήσεις που χρειάστηκαν, κτλ. Οι μετρικές πόρων μας δίνουν απαντήσεις σε ερωτήματα σχετικά με το προσωπικό, τα υλικά (αναλώσιμα, εξοπλισμός γραφείου, κτλ) που χρησιμοποιήθηκαν, τα εργαλεία (τόσο σε επίπεδο υλικού όσο και λογισμικού) και τις μεθόδους ανάπτυξης. Οι περισσότερες μετρικές πόρων μετρούν εσωτερικά χαρακτηριστικά τα οποία είναι σχετικά εύκολο να μετρηθούν και για αυτό δεν θα

---

[39] *E. Weller, «Practical Applications of Statistical Process Control», IEEE Software, Vol. 17, No. 3, (2000)*

επεκταθούμε περισσότερο σε αυτές (για παράδειγμα είναι εύκολο να μετρηθεί χωρίς κάτι τέτοιο να απαιτεί κάποια ιδιαίτερη ανάλυση, ο αριθμός των δισκετών που αγοράστηκαν για ένα έργο και το κόστος τους). Εκτός από τα εσωτερικά χαρακτηριστικά υπάρχουν και εξωτερικά χαρακτηριστικά που σχετίζονται με τους πόρους και είναι δύσκολο να μετρηθούν, όπως για παράδειγμα η παραγωγικότητα του προσωπικού.

Τέλος, οι μετρικές προϊόντος σχετίζονται με οτιδήποτε θα παραδοθεί στον πελάτη. Έμφαση θα δώσουμε φυσικά στις μετρικές λογισμικού. Οι μετρικές λογισμικού μπορεί να είναι εσωτερικές ή εξωτερικές.

Όπως αναφέραμε πιο πάνω, οι εσωτερικές μετρικές χρησιμοποιούνται για τη συλλογή δεδομένων που σχετίζονται με εκείνα τα χαρακτηριστικά του λογισμικού που μπορούν να οριστούν ξεκάθαρα (χωρίς να εισέρχεται το στοιχείο της υποκειμενικότητας) και να μετρηθούν με ένα απλό και σαφώς καθορισμένο τρόπο. Αυτή η μέτρηση πρέπει να οδηγεί πάντα στο ίδιο αποτέλεσμα, ανεξάρτητα με το ποιος εκτελεί τις μετρήσεις. Για παράδειγμα, όταν μετράμε τον αριθμό γραμμών κώδικα πρέπει να είναι ξεκάθαρο τι αποτελεί γραμμή κώδικα και τι όχι, έτσι ώστε η μέτρηση να είναι ανεξάρτητη από παραδοχές που μπορεί να κάνει οποιοσδήποτε αναλαμβάνει τη μέτρηση.

Οι εσωτερικές μετρήσεις, για να έχουν αξία, πρέπει να μπορούν να συσχετίζονται με εξωτερικά ποιοτικά χαρακτηριστικά. Αυτές οι συσχετίσεις προκύπτουν από την ανάλυση των μετρήσεων και την αξιοποίηση των ιστορικών δεδομένων. Για παράδειγμα, μία επιχείρηση που χρησιμοποιεί ως γλώσσα ανάπτυξης λογισμικού την Object Pascal (χρησιμοποιώντας ως περιβάλλον ανάπτυξης το Borland Delphi), μπορεί να συσχετίσει την εσωτερική μέτρηση LOC με τα λάθη που εντοπίζει ο πελάτης. Έτσι, θα προέκυπτε μία ανάλυση που θα συσχέτιζε τα λάθη που εντοπίζει ο πελάτης με τις γραμμές κώδικα, δίνοντας μετρήσεις σε μονάδες «λάθη πελάτη ανά KLOC». Αναλύοντας στατιστικά τα δεδομένα για κάθε τμήμα κώδικα, η επιχείρηση θα αποκτούσε εξειδικευμένη γνώση για κάθε τμήμα, αλλά –το κυριότερο– δεδομένα για το ιδανικό μέγεθος (σε LOC) κάθε τμήματος, ή για το όριο στο οποίο ένα τμήμα πρέπει να κοπεί σε μικρότερα. Αντίστοιχες συσχετίσεις μπορούν να γίνουν με την παραγωγικότητα και τις γραμμές κώδικα, ή άλλες εσωτερικές μετρικές. Παρόμοιες μετρήσεις μπορούν να συγκεντρωθούν για πολλές εσωτερικές μετρικές και αυτές να οδηγήσουν την επιχείρηση σε αποφάσεις. Αυτές οι αποφάσεις πρέπει να αντικατοπτρίζονται στο

εγχειρίδιο ποιότητας. Οι αποφάσεις που βασίζονται σε εσωτερικές μετρικές έχουν ως κύριο σκοπό τους την πρόληψη. Έτσι, για παράδειγμα, αν μία επιχείρηση έχοντας αναλύσει δεδομένα από μετρήσεις, αποφασίσει ότι κάθε τμήμα δεν πρέπει να ξεπερνά τις 500 LOC (για κάποια συγκεκριμένη γλώσσα προγραμματισμού), αυτό δεν σημαίνει ότι ένα τμήμα 600 LOC (για το οποίο απαιτείται προσπάθεια, προκειμένου να διασπαστεί σε 2 μικρότερα τμήματα) απαραίτητα θα δημιουργήσει προβλήματα. Όμως, επειδή η πιθανότητα να δημιουργήσει προβλήματα είναι υψηλή, η επιχείρηση αναλαμβάνει το κόστος των αλλαγών, ελπίζοντας να προλάβει τα λάθη που θα εντοπίσει ο πελάτης. Τέτοια λάθη, κοστίζουν περισσότερο στην επιχείρηση (και πέρα από το υλικό κόστος υπάρχουν και έμμεσες ζημιές που σχετίζονται με την εικόνα της επιχείρησης προς τον πελάτη).

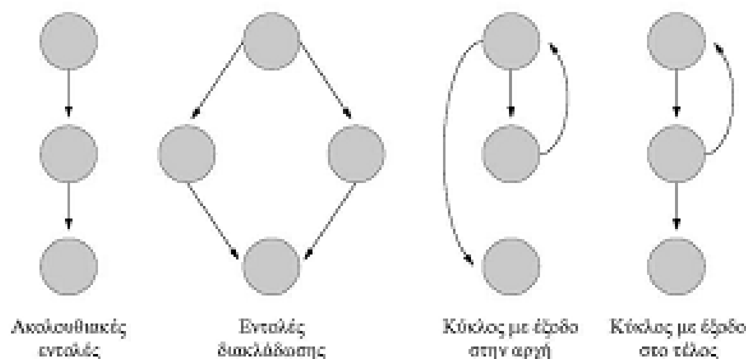
Γενικότερα, μπορούμε να πούμε ότι οι εσωτερικές μετρικές είναι –συνήθως– εύκολο να συγκεντρωθούν και να αναλυθούν, ότι η συλλογή δεδομένων που βασίζονται σε εσωτερικές μετρικές είναι πολύ οικονομική (συγκρινόμενη και με την αντίστοιχη συλλογή δεδομένων για εξωτερικές μετρικές) και τα αποτελέσματα που προκύπτουν είναι εύκολο να αναλυθούν με στατιστικές μεθόδους. Το κυριότερο είναι ότι τα δεδομένα από αυτές μπορούν να συγκεντρωθούν και να οδηγήσουν σε αποφάσεις πολύ πριν την ολοκλήρωση και τελική παράδοση του λογισμικού. Από την άλλη, τα αποτελέσματά τους είναι –συνήθως– δύσκολο να ερμηνευτούν (συγκρινόμενα με αυτά των εξωτερικών μετρήσεων) αφού πολλές φορές δεν είναι ξεκάθαρη η σύνδεσή τους με κάποιο ή κάποια εξωτερικά ποιοτικά χαρακτηριστικά. Έρευνες[40] έχουν αποδείξει ότι η χρήση εσωτερικών μετρικών μπορεί να εντοπίσει σχεδόν όλα τα τμήματα του κώδικα που θα δημιουργήσουν προβλήματα, αλλά μπορεί να υποδείξει και τμήματα κώδικα που τελικά δεν θα δημιουργήσουν προβλήματα. Ακόμα κι έτσι, όμως, το όφελος από την πρώτη περίπτωση (εντοπισμός προβληματικών τμημάτων πριν παραδοθούν) είναι πολύ σημαντικότερο από το κόστος που συνεπάγεται η δεύτερη περίπτωση (επιβολή αλλαγών σε «υγιή» τμήματα). Συνεχίζοντας, θα δώσουμε ένα παράδειγμα μίας ακόμα πολύ γνωστής μετρικής, αυτής της κυκλωματικής πολυπλοκότητας.

---

[40] *M. Xenos et al., «The Correlation Between Developer – oriented and User – oriented Software Quality Measurements (A Case Study)», 5th European Conference on Software Quality, EOQ – SC, (1996) & M. Xenos and D. Christodoulakis, «Measuring Perceived Software Quality», Information Technology Journal, Vol. 39, (1997).*

Η μετρική της κυκλωματικής πολυπλοκότητας (cyclomatic complexity[41]) είναι ένα μέτρο της πολυπλοκότητας ενός τμήματος, που βασίζεται στο γράφο ροής (flow graph) του τμήματος αυτού. Βασικό στοιχείο της μετρικής είναι ο εντοπισμός εκείνων των τμημάτων του λογισμικού, τα οποία θα είναι δύσκολο να κατανοηθούν, να ελεγχθούν και να συντηρηθούν. Η μελέτη του τμήματος λογισμικού, από τη σκοπιά αυτής της μετρικής, σχετίζεται με τους πιθανούς δρόμους (μονοπάτια) που μπορεί να ακολουθήσει η εκτέλεση ενός τμήματος κώδικα. Μια ένδειξη των δυνατών επιλογών κατεύθυνσης ενός προγράμματος παρέχει ο γράφος ροής του προγράμματος που μας δίνει, ίσως, την καλύτερη ένδειξη για τη δομή του προγράμματος. Πάνω στο γράφο ροής βασίζονται οι μετρήσεις. Ο γράφος ροής ενός προγράμματος δείχνει σχηματικά τους πιθανούς «δρόμους» που μπορεί να ακολουθήσει η εκτέλεση του προγράμματος, επιλέγοντας κάθε φορά ένα διαφορετικό μονοπάτι από τις δυνατότητες επιλογής. Οι τέσσερις περιπτώσεις οι οποίες συναντιόνται συχνότερα στο γράφο ροής είναι: ακολουθιακές εντολές, εντολή διακλάδωσης, κύκλος με έξοδο στην αρχή, κύκλος με έξοδο στο τέλος. Ακολουθιακές εντολές είναι ακολουθίες εντολών κατά τις οποίες δεν γίνεται επιλογή. Τέτοιες εντολές θα εκτελεστούν σειριακά, χωρίς να υπάρχει επιλογή διαφορετικού τρόπου (μονοπατιού) εκτέλεσής τους. Εντολές διακλάδωσης (όπως η εντολή *«if … then … else …»* στην C) είναι αυτές που καθορίζουν δύο μονοπάτια επιλογής. Μετά την ολοκλήρωση των μονοπατιών και τα δύο καταλήγουν στο κοινό σημείο του προγράμματος, από το οποίο αρχίζει η επόμενη εντολή. Οι εντολές κύκλου με έξοδο στην αρχή (χαρακτηριστικό παράδειγμα η εντολή *«while … do»)* και οι εντολές κύκλου με έξοδο στο τέλος (χαρακτηριστικό παράδειγμα η εντολή «*do … while*») αφορούν τις τυπικές δομές επανάληψης που έχετε διδαχθεί. Όλες αυτές οι εντολές μπορούν να παρασταθούν γραφικά στο γράφο ροής του προγράμματος, όπως φαίνεται στο σχήμα που ακολουθεί.

[41] *T. J. McCabe, «A Complexity Measure», IEEE Transactions in Software Engineering SE – 2(4), (1976)*



Αντίστοιχα με τις παραπάνω περιπτώσεις, υπάρχουν πιο πολύπλοκες καταστάσεις, οι οποίες μπορούν να δημιουργηθούν με τη χρήση εντολών «*goto*», που αν εμπεριέχονται σε εντολές επανάληψης ή οδηγούν μέσα σε αυτές, περιπλέκουν σε μεγάλο βαθμό τη μορφή του γράφου. Ο McCabe όρισε τη μετρική της κυκλωματικής πολυπλοκότητας V(g) για ένα γράφο ροής ***g***, όπως ορίζεται στη σχέση A, όπου ***e*** είναι ο αριθμός των ακμών του γράφου, ***n*** ο αριθμός των κόμβων του γράφου, και ***p*** ο αριθμός των συνεκτικών συνιστωσών[42] του γράφου.

$$V(g) = e - n + 2p$$

Σχέση A

Η μετρική της κυκλωματικής πολυπλοκότητας είναι από τις πιο διαδεδομένες μετρικές και έχει ενταχθεί στα προγράμματα μετρήσεων των περισσοτέρων εταιριών. Αρκετές εταιρίες έχουν καθορίσει στα πρότυπα ποιότητάς τους, όπως περιγράφονται στα εγχειρίδια ποιότητάς τους, ότι τμήματα κώδικα με πολυπλοκότητα *V(g)>10* δεν θα γίνονται αποδεκτά, αφού τα ιστορικά τους δεδομένα έδειχναν υποβάθμιση της ποιότητας για τμήματα με τέτοια χαρακτηριστικά (κυκλωματική πολυπλοκότητα μεγαλύτερη του 10). Βασικό πλεονέκτημα της μετρικής είναι ότι είναι τελείως ανεξάρτητη από τη γλώσσα προγραμματισμού.

Πριν προχωρήσουμε στις εξωτερικές μετρήσεις, πρέπει να τονιστεί ότι, αν και οι εξωτερικές μετρήσεις συνήθως σχετίζονται με το χρήστη, δεν περιορίζονται μόνο στα ποιοτικά χαρακτηριστικά της λειτουργικότητας, της ευχρηστίας, της αποδοτικότητας και της αξιοπιστίας. Εξωτερικές μετρήσεις μπορούν να γίνουν και

[42] *Ως συνεκτική συνιστώσα ενός γράφου ορίζεται το τμήμα του γράφου το οποίο περιέχει όλους τους κόμβους και τις ακμές μεταξύ αυτών των κόμβων, για τους οποίους υπάρχει διαδρομή που τους ενώνει μεταξύ τους. Στην πράξη, αν μιλάμε για αυτόνομα τμήματα κώδικα, το p θα είναι πάντα ίσο με 1.*

για τα ποιοτικά χαρακτηριστικά της μεταφερσιμότητας και της συντηρησιμότητας. Το τμήμα συντήρησης, εξάλλου, μπορεί να θεωρηθεί ως χρήστης του κώδικα με σκοπό την υλοποίηση αλλαγών. Μετρήσεις εξωτερικές για τη συντηρησιμότητα θα μπορούσαν να είναι: η εκτίμηση για το μέσο χρόνο εντοπισμού και υλοποίησης μίας διόρθωσης, το ποσοστό επιτυχίας της διόρθωσης, το μέσο μέγεθος του κώδικα που χρειάζεται να τροποποιηθεί για μία διόρθωση, το μέσο μέγεθος του κώδικα που χρειάστηκε να διαβαστεί (εξεταστεί) μέχρι να εντοπιστεί το τμήμα στο οποίο υπάρχει το λάθος, κτλ. Όπως βλέπετε, οι εξωτερικές μετρικές εμπεριέχουν το στοιχείο της υποκειμενικότητας. Βέβαια, στα περισσότερα εγχειρίδια ποιότητας, η περιγραφή τους γίνεται με αυστηρούς τυπικούς περιορισμούς, έτσι ώστε να μειώνονται τα περιθώρια της υποκειμενικότητας. Εξάλλου, οι μετρήσεις που προκύπτουν από την χρήση τους είναι άμεσα αξιοποιήσιμες για εξαγωγή συμπερασμάτων σχετικά με τη συντηρησιμότητα και δεν χρειάζεται η αναζήτηση συσχετίσεων με εξωτερικά ποιοτικά χαρακτηριστικά (όπως στην περίπτωση των εσωτερικών μετρικών).

Οι εξωτερικές μετρήσεις για ποιοτικά χαρακτηριστικά που αφορούν τον τελικό χρήστη μπορούν να βασιστούν σε τρεις κατηγορίες μεθόδων[43]:

α) Αναλυτικές μέθοδοι (analytic methods)

β) Πειραματικέςμέθοδοι (experimental methods)

γ) Διερευνητικές μέθοδοι (inquiry methods)

Οι αναλυτικές μέθοδοι εκτελούνται στο εργαστήριο και δεν συμμετέχουν οι τελικοί χρήστες. Ως ενδεικτικά παραδείγματα τέτοιων μεθόδων, αναφέρουμε την ανάλυση πληκτρολογήσεων (όπου χρησιμοποιείται ένα μοντέλο ανάλυσης εργασιών και με υπολογισμό των χρόνων που προβλέπονται για τυπικές ενέργειες του χρήστη, εκτιμάται ο χρόνος ολοκλήρωσης ενός έργου με τη χρήση του προγράμματος), την ευρετική αξιολόγηση (όπου εξωτερικοί κριτές εξετάζουν την τήρηση κανόνων με βάση κάποια κριτήρια σπουδαιότητας, εξειδικευμένα για κάθε εφαρμογή) και τον έλεγχο συμβατότητας με κανόνες σχεδιασμού και πρότυπα (όπου οι κανόνες δίδονται υπό μορφή καταλόγων ελέγχου και οι ειδικοί αξιολογούν την ικανοποίησή τους ή όχι από το σύστημα). Οι πειραματικές μέθοδοι γίνονται στο εργαστήριο αλλά –σε αντίθεση με τις αναλυτικές μεθόδους– υπάρχει συμμετοχή των τελικών χρηστών. Σε αυτές τις μεθόδους οι χρήστες

[43] *Νικόλαος Αβούρης, «Εισαγωγή στην Επικοινωνία Ανθρώπου – Υπολογιστή», Εκδόσεις ΔΙΑΥΛΟΣ, (2000)*

χρησιμοποιούν το πρόγραμμα μέσα σε ένα εργαστήριο, ενώ οι ειδικοί (συνήθως αθέατα και με χρήση εξειδικευμένου εξοπλισμού, όπως βίντεο, ή σύστημα καταγραφής συμβάντων) τους παρακολουθούν και σημειώνουν (μετρούν) τις αντιδράσεις τους. Τέλος, οι διερευνητικές μέθοδοι εκτελούνται εκτός εργαστηρίου και προϋποθέτουν ενεργή συμμετοχή χρηστών. Ως ενδεικτικά παραδείγματα τέτοιων μεθόδων αναφέρουμε τις συνεντεύξεις των χρηστών (όπου κάποιος αξιολογητής καταγράφει τις απόψεις του χρήστη) και τη συμπλήρωση ερωτηματολογίων (όπου οι χρήστες καλούνται να εκφέρουν άποψη για τα ποιοτικά χαρακτηριστικά του λογισμικού).

Από τις μεθόδους που αναφέραμε, η πιο διαδεδομένη στην πράξη είναι η χρήση ερωτηματολογίου. Τα πλεονεκτήματα της μεθόδου αυτής είναι ότι μία επιχείρηση μπορεί να ζητήσει την άποψη των χρηστών για ποιοτικά χαρακτηριστικά του λογισμικού με μικρό κόστος (τα ερωτηματολόγια μπορεί να αποστέλλονται στους χρήστες ηλεκτρονικά, ή να βρίσκονται στον κόμβο της επιχείρησης στο διαδίκτυο και οι χρήστες να τα συμπληρώνουν εκεί). Μετρήσεις τέτοιου τύπου είναι απόλυτα συμβατές με τον ορισμό της ποιότητας (που σχετίζεται άμεσα με την άποψη των χρηστών) και οδηγούν σε άμεσα αναλύσιμα δεδομένα. Τα μειονεκτήματα αυτής της μεθόδου είναι η υποκειμενικότητα των απαντήσεων και τα συχνά λάθη. Όσον αφορά το πρώτο μειονέκτημα, ένας καλός τρόπος αντιμετώπισής του από το σχεδιαστή του ερωτηματολογίου είναι η ορθή δόμηση του ερωτηματολογίου, η σαφήνεια και ο έλεγχος των απαντήσεων (με χρήση διπλών ερωτήσεων, ερωτήσεων ασφαλείας, κτλ). Για τον εντοπισμό των χρηστών που έχουν πολλά λάθη στις απαντήσεις τους –και κατά συνέπεια δεν θα πρέπει να συμπεριληφθούν στην ανάλυση των αποτελεσμάτων– υπάρχουν τεχνικές που βασίζονται σε ερωτήσεις ασφαλείας (ερωτήσεις που δεν έχουν στόχο να μετρήσουν την άποψη του χρήστη για κάποιο ποιοτικό χαρακτηριστικό, αλλά να ελέγξουν την εγκυρότητα των απαντήσεων του χρήστη). Συνοψίζοντας, πρέπει να τονιστεί ότι οι εξωτερικές μετρήσεις βασίζονται στον ορισμό της ποιότητας (ικανοποίηση των τελικών χρηστών) και μετρούν άμεσα τα επιθυμητά εξωτερικά χαρακτηριστικά, χωρίς να χρειάζεται περαιτέρω ανάλυση, ή ερμηνεία. Όμως, αρκετές από τις μεθόδους μέτρησης δεν είναι καθόλου οικονομικές (ιδιαίτερα συγκρινόμενες με τις εσωτερικές μετρήσεις) και τα αποτελέσματα βασίζονται συχνά σε υποκειμενικές κρίσεις ή απόψεις. Παρά τα προβλήματα αυτά, οι

εξωτερικές μετρήσεις είναι πολύτιμο εργαλείο για το τμήμα ποιότητας κάθε επιχείρησης.

Η εύρεση εσωτερικών μετρικών, οι οποίες ενσωματωμένες σε μία μέθοδο εσωτερικών μετρήσεων θα είχαν απόλυτη συσχέτιση με τις εξωτερικές μετρήσεις είναι πολύ δύσκολη. Πρακτικά, εάν υπήρχε ένα τέτοιο σύνολο μετρικών, τότε δε θα υπήρχε ανάγκη για εξωτερικές μετρήσεις, αφού η τήρηση των ορίων τα οποία τίθενται από αυτές τις εσωτερικές μετρικές θα υποκαθιστούσε το πρόγραμμα ποιότητας. Για να το θέσουμε πιο απλά: η τήρηση των ορίων στις εσωτερικές μετρικές θα συνεπαγόταν αυτόματα υψηλή ποιότητα του λογισμικού, οπότε δεν θα υπήρχε λόγος επιβεβαίωσης με τη χρήση εξωτερικών μετρικών. Ο λόγος, όμως, που δεν μπορεί να βρεθεί ένα τέτοιο σύνολο μετρικών είναι ότι η «ποιότητα» συντίθεται από παράγοντες με υψηλό επίπεδο αφαίρεσης, όπως για παράδειγμα η φιλικότητα προς το χρήστη, οι οποίοι δεν μπορούν να μετρηθούν εσωτερικά και που δε σχετίζονται με χαρακτηριστικά τα οποία μετρούνται εσωτερικά βασισμένα σε εσωτερικές μετρικές. Το ζητούμενο είναι κατά πόσο μία μέθοδος εσωτερικών μετρήσεων μπορεί να προβλέψει αρνητικές καταστάσεις, δηλαδή κατά πόσο οι εσωτερικές μετρήσεις μπορούν να εντοπίσουν τμήματα του λογισμικού τα οποία ενδέχεται να δημιουργήσουν προβλήματα, τόσο κατά τις εσωτερικές στην επιχείρηση φάσεις αξιολόγησης (έλεγχο και συντήρηση), όσο και κατά την τελική αξιολόγηση από τους χρήστες. Η χρήση εσωτερικών μετρήσεων και μετρικών, για τις οποίες μιλήσαμε πιο πάνω σε αυτό το κεφάλαιο, είναι μία μέθοδος η οποία μπορεί να εντοπίσει τέτοια προβληματικά τμήματα του λογισμικού. Ζητούμενο είναι κατά πόσο συσχετίζονται οι εσωτερικές μετρήσεις που γίνονται με αυτή τη μέθοδο με τις εξωτερικές μετρήσεις που πραγματοποιούνται με τη μέθοδο που περιγράψαμε επίσης πιο πριν. Αυτό το ζητούμενο μπορεί να αναλυθεί στα δύο ερωτήματα που ακολουθούν:

α) Ένα έργο το οποίο ικανοποιεί ένα πρόγραμμα ποιότητας βασισμένο σε εσωτερικές μετρήσεις, θα έχει αντίστοιχη επιτυχία στις μετρήσεις της άποψης των πελατών για την ποιότητά του;

β) Ένα έργο το οποίο απέτυχε να ικανοποιήσει ένα πρόγραμμα ποιότητας βασισμένο σε εσωτερικές μετρήσεις, θα γνωρίσει επίσης αποτυχία στις μετρήσεις της άποψης των πελατών για την ποιότητά του;

Η απάντηση στο πρώτο ερώτημα είναι: «δεν υπάρχει εγγύηση ότι ένα έργο το οποίο ικανοποίησε το πρόγραμμα εσωτερικών μετρήσεων θα έχει αντίστοιχα

ικανοποιητικές εξωτερικές μετρήσεις». Αυτό οφείλεται στο γεγονός ότι οι εσωτερικές μετρήσεις δεν μπορούν να καλύψουν όλους τους εξωτερικούς ποιοτικούς παράγοντες. Αντίθετα για το δεύτερο ερώτημα η απάντηση είναι: «αποτυχία στο πρόγραμμα εσωτερικών μετρήσεων κατά κανόνα σημαίνει και αποτυχία στις εξωτερικές μετρήσεις» . Αυτό οφείλεται στο γεγονός ότι αποτυχία στις εσωτερικές μετρήσεις σημαίνει αποτυχία τήρησης κανόνων και αρχών στην παραγωγή του λογισμικού, αδυναμία ελέγχου του κώδικα, αδυναμία εντοπισμού εσωτερικών στόχων κτλ. Όλα αυτά, σχεδόν πάντα, αντικατοπτρίζονται στην ποιότητα του τελικού προϊόντος όπως την αντιλαμβάνεται ο χρήστης (πελάτης). Είναι σχεδόν αδύνατο για ένα έργο να μην ικανοποιεί εσωτερικούς ποιοτικούς περιορισμούς και παρόλα αυτά να κρίνεται ικανοποιητικό από τους πελάτες. Ένα έργο που αποτυγχάνει στις εσωτερικές μετρήσεις πρακτικά σημαίνει προχειροφτιαγμένος κώδικας. Κατά συνέπεια αποτυχία σε εσωτερικές μετρικές θα σημαίνει αποτυχία και στην άποψη των πελατών για την ποιότητα.

Οι μετρικές της επιστήμης λογισμικού του Halstead[44] μετρούν αριθμήσιμα στοιχεία του λογισμικού σχετιζόμενα με το μέγεθος του πηγαίου κώδικα. Οι τέσσερις βασικές μετρήσιμες ποσότητες που προτείνει είναι οι ακόλουθες:

n_1	ο αριθμός των διακριτών τελεστών που εμφανίζονται στο πρόγραμμα.
n_2	ο αριθμός των διακριτών εντέλων που εμφανίζονται στο πρόγραμμα.
N_1	ο αριθμός των συνολικών εμφανίσεων τελεστών στο πρόγραμμα.
N_2	ο αριθμός των συνολικών εμφανίσεων εντέλων στο πρόγραμμα.

Από τα τέσσερα παραπάνω προκύπτουν και οι υπόλοιπες μετρικές της επιστήμης λογισμικού που είναι οι εξής:

Λεξιλόγιο προγράμματος (n)	$n = n_1 + n_2$
Μήκος προγράμματος (N)	$N = N_1 + N_2$
Εκτιμητής μήκους προγράμματος (N_{est})	$N_{est} = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2$
Όγκος προγράμματος (V)	$V = N \cdot \log_2 n$
Δυναμικός όγκος προγράμματος (V^*) που αναφέρεται στην πιο συνοπτική θεωρητική υλοποίηση του προγράμματος	
Επίπεδο προγράμματος (L)	$L = V^*/V$
Εκτιμητής επιπέδου προγράμματος (L_{est})	$L_{est} = 2 \cdot n_2 / n_1 \cdot N_2$
Επίπεδο γλώσσας στο πρόγραμμα (λ)	$\lambda = L^2 \cdot V$
Δυσκολία προγράμματος (D)	$D = 1/L$
Προσπάθεια υλοποίησης προγράμματος (E)	$E = DV$
Χρόνος υλοποίησης προγράμματος (T)	$T = E/S$, (S ο αριθμός Stroud)
Εκτίμηση αριθμού λαθών στο πρόγραμμα (B)	$B = E^{2/3}/E_0$, ($E_0 \in [3000,3200]$)

[44] *Halstead M.H., "Elements of Software Science", Elsevier Publications, N-Holland, 1975.*

Με τη χρήση των μετρικών του Halstead μπορούν εύκολα να διαγνωσθούν τυχόν προγραμματιστικές ατέλειες σε ένα πρόγραμμα λογισμικού. Συγκεκριμένα, ρουτίνες στις οποίες θα εμφανίζονταν συμπληρωματικοί τελεστές, διφορούμενα ή συνώνυμα εκτελεί, αχρείαστες καταχωρήσεις, όμοιες υποεκφράσεις και μη παραγοντοποιημένες εκφράσεις θα έδιναν μικρά αποτελέσματα στις παραπάνω μετρικές .

Επίσης, η μετρική κυκλωματικής πολυπλοκότητας του McCabe[45] είναι μία μετρική δομής και μετράει πόσο πολύπλοκος είναι ο γράφος ροής του προγράμματος. Επικεντρώνει το ενδιαφέρον της στα σημεία του προγράμματος που μπορεί να ληφθεί απόφαση και σχετίζεται με τα πιθανά μονοπάτια που μπορεί να ακολουθήσει η ροή του προγράμματος. Η μετρική αυτή ονομάζεται και κυκλωματικός αριθμός V(G), όπου G ο γράφος του προγράμματος, και δίνεται από την παρακάτω σχέση:

$V(G) = e - n + 2 \cdot p$, όπου e ο αριθμός των ακμών του γράφου,
n ο αριθμός των κόμβων του γράφου και
p ο αριθμός των συνεκτικών συνιστωσών του γράφου.

Στον παραπάνω γράφο, που είναι ένας απλοποιημένος γράφος ελέγχου, οι κόμβοι είναι τα σημεία απόφασης και οι ακμές είναι οι εναλλακτικές δράσεις. Προφανώς, όσο μεγαλύτερος είναι ο κυκλωματικός αριθμός μίας ρουτίνας, τόσο πιο πολύπλοκη είναι αυτή. Σε περιπτώσεις ρουτινών με πολύ μεγάλο κυκλωματικό αριθμό συνιστάται να γίνεται διάσπασή τους σε περισσότερες.

Τέλος, η μετρική της πολυπλοκότητας δομών δεδομένων του Tsai[46] εφαρμόζεται στα δεδομένα του προγράμματος, επομένως το μόνο που χρειάζεται είναι να υπάρχουν οι λεπτομερείς κατάλογοι των δεδομένων που θα επεξεργαστεί το πρόγραμμα και όχι η τελική έκδοση του κώδικα του προγράμματος. Σύμφωνα με τον αλγόριθμο εφαρμογής αυτής της μετρικής, κάθε δομή δεδομένων D είναι της μορφής: D::F(D1,…,Dn), με n≥1, όπου F ο τελεστής σχηματισμού της δομής δεδομένων, Di οι αναφορές σε δομές δεδομένων που χρησιμοποιήθηκαν για το σχηματισμό της D και n ο αριθμός των Di. Τα επόμενα βήματα είναι α) η κατασκευή ενός κατευθυνόμενου γράφου, στον οποίο η κάθε δομή δεδομένων αναπαρίσταται με έναν κόμβο, β) η μείωση του γράφου, κατά την οποία αφαιρούνται κόμβοι με μοναδική είσοδο και μοναδική έξοδο και γ) ο εντοπισμός

---

[45] *McCabe T.J., "A complexity measure", IEEE Transactions in Software Engineering SE-2 (4), pp. 308-320, 1976*

[46] *Tsai W.T., Lopez M.A., Rodriguez V. and Volovik D., "An Approach to Measuring Data Structure Complexity", Compsac86, pp. 240-246, 1986.*

των ισχυρά συνεκτικών συνιστωσών. Στη συνέχεια υπολογίζονται τα μονώνυμα S(K) των συνεκτικών συνιστωσών βάσει του τύπου S(K) = (V + E)xL, όπου K η συνιστώσα, V ο αριθμός των κόμβων της, E ο αριθμός των ακμών της και L ο αριθμός των απλών κυκλικών μονοπατιών μέσα σε αυτήν. Τέλος, υπολογίζονται τα πολυώνυμα πολυπλοκότητας C(K) βάσει του τύπου :

$$C(K) = \sum_{\forall K_i \leftarrow K} [C(K_i)] + S(K)$$

Λέγοντας ότι Km←Kn με Km≠Kn αν και μόνο αν υπάρχει μονοπάτι από κάποιον κόμβο της Kn σε κάποιον κόμβο της Km. Προφανώς, όσο μεγαλύτερου βαθμού είναι τα πολυώνυμα που προκύπτουν, τόσο πιο πολύπλοκες είναι οι δομές δεδομένων του προγράμματος. Με τη χρήση αυτής της μετρικής είναι δυνατή η μείωση της πολυπλοκότητας των προγραμμάτων λογισμικού αρκετά πριν την ολοκλήρωσή τους. Με άλλα λόγια, μπορούν να προληφθούν τα προβλήματα πριν δημιουργηθούν, αντί να εντοπίζονται ήδη υπάρχοντα προβλήματα και να αναζητείται η μέθοδος διόρθωσής τους.

Ανακεφαλαιώνοντας λοιπόν τα περί μετρικών λογισμικού πρέπει να αναφέρουμε ότι ο Roger Pressman[47] χωρίζει τις μετρικές λογισμικού σε κατηγορίες, ανάλογα με τη φάση ανάπτυξης του λογισμικού που εφαρμόζονται:

- Μετρικές για το μοντέλο της ανάλυσης απαιτήσεων, που εξετάζουν την ορθότητα και την πληρότητα της ανάλυσης των απαιτήσεων, προκειμένου να γίνει ο σχεδιασμός του τελικού συστήματος. Τέτοιες μετρικές αριθμούνται ελάχιστες στη βιβλιογραφία.

- Μετρικές για το μοντέλο του σχεδιασμού, που αναπτύσσονται για τον έλεγχο του σχεδιασμού του τελικού συστήματος. Εστιάζουν τόσο στο τμήματα (components) του συστήματος, όσο και στη διεπαφή (interface).

- Μετρικές για τον πηγαίο κώδικα, που αφορούν μετρήσεις πάνω στον κώδικα και αναλύονται στη συνέχεια της ενότητας.

- Μετρικές για τον έλεγχο (testing), που αφορούν στον έλεγχο του λογισμικού. Οι περισσότερες μετρικές εστιάζουν στη διαδικασία του ελέγχου και όχι στα τεχνικά χαρακτηριστικά. Οι μετρικές του ελέγχου προκύπτουν ουσιαστικά από τις μετρικές των τριών προηγούμενων φάσεων.

- Μετρικές για τη συντήρηση, που μετρούν το βαθμό στον οποίο είναι συντηρήσιμο το λογισμικό.

[47] *Roger S. Pressman, “Software Engineering: A Practitioner’s Approach”, 4th ed.,McGraw Hill 1997*

Ακολουθεί ένας πίνακας όπου παρουσιάζονται ορισμένες από τις πιο γνωστές μετρικές προϊόντος που εφαρμόζονται στον τελικό κώδικα του λογισμικού.

Average Module Length Πρόκειται για μία μετρική της τμηματοποίησης ενός προγράμματος. Ορίζεται ως το μέσο μήκος των τμημάτων (modules) του.
Chen Metric Εξετάζει την εντροπία ενός προγράμματος.
Decision Count Μετρά τη λογική δομή ενός προγράμματος. Ορίζεται από τον αριθμό των δηλώσεων ελέγχου (υπό-συνθήκη δηλώσεις και ανακυκλώσεις).
Executable Statements Μετρική μεγέθους. Ορίζεται ως τον αριθμό των εκτελέσιμων εντολών (executable statements) ενός προγράμματος. Μετρά τις διαφορετικές εντολές στην ίδια γραμμή ως διαφορετικές και αγνοεί τα σχόλια, τις δηλώσεις δεδομένων και τα headings.
Extent Of Reuse Μετρά το ποσοστό επαναχρησιμοποίησης κώδικα σε ένα υπάρχον πρόγραμμα. Ορίζονται τέσσερα διαφορετικά επίπεδα: reused verbatim , slightly modified , extensively modified και new
Function Count Ορίζεται από τον αριθμό των συναρτήσεων (functions) ενός προγράμματος.
Hausen Metric On Modularity Μετρική της τμηματοποίησης ενός προγράμματος που περιγράφει την ολική τμηματοποίηση σε σχέση με ειδικές της θεωρήσεις. Για παράδειγμα: M1=modules/procedures, M2=modules/variables.
Live Variables Στόχος της μετρικής αυτής είναι να χαρακτηρίσει τη ροή των δεδομένων μέσα σε ένα τμήμα (module).
Lines Of Code Μετρά τον αριθμό των γραμμών κώδικα.
McCabe's Essential Complexity Measure Μετράται το ποσό της δομής σε ένα πρόγραμμα. Για ένα πρόγραμμα με γράφο ροής G, η βασική πολυπλοκότητα ορίζεται ως $ev(G) = v(G) - m$, όπου $v(G) = e - n + 2$ η κυκλωματική πολυπλοκότητα του γράφου ροής G και m ο αριθμός των υπογράφων ροής του γράφου G που είναι D-structured primes Μετρά πόσο πρέπει να 'μειωθεί' ο γράφος ροής.
Reach Ability Μετρά τη λογική δομή του προγράμματος. Ορίζει την reachability ως R = πλήθος διαφορετικών τρόπων για να φτάσει κάποιος ένα κόμβο και R' = (συνολικό πλήθος μονοπατιών) / (αριθμός κόμβων) Η μετρική αυτή σχετίζεται με τον αριθμό των λαθών και τον χρόνο εντοπισμού τους.
Tree Impurity Η μετρική αυτή ορίζει τον αριθμό m(G) από το γράφο G του συστήματος. O m(G) ισούται με το 'πόσο απέχει' από ένα δένδρο ο γράφος αυτός. Όσο το m τείνει στο μηδέν, τόσο καλύτερος είναι ο σχεδιασμός του συστήματος.

Στον αντικειμενοστραφή προγραμματισμό τον κύριο ρόλο έχουν οι κλάσεις και τα αντικείμενα. Κάθε αντικείμενο έχει μια συμπεριφορά, η οποία καθορίζεται από τις μεθόδους (συναρτήσεις) που αυτό διαθέτει. Μεταξύ των αντικειμένων και των κλάσεων δεν υπάρχουν απλώς σχέσεις καλούντος-καλούμενου, αλλά πλήθος άλλων σχέσεων, όπως ιεραρχίας, συνδέσμου κλπ. Συνεπώς, ο αντικειμενοστραφής προγραμματισμός έχει τα δικά του χαρακτηριστικά που επηρεάζουν την ποιότητα του λογισμικού. Δηλαδή, χρησιμοποιώντας την ορολογία του μοντέλου του McCall[48], η ποιότητα διασπάται στους ποιοτικούς παράγοντες και αυτοί στα κριτήρια ποιότητας, τα οποία όμως εξαρτώνται κυρίως από την ύπαρξη ή όχι κάποιων χαρακτηριστικών αντικειμενοστραφούς προγραμματισμού. Αντίστροφα, οι μετρικές που χρησιμοποιούνται πρέπει να μετρούν κατά πόσο υπάρχουν τα χαρακτηριστικά αυτά, τα οποία και βελτιώνουν την ποιότητα των αντικειμενοστραφών προγραμμάτων.

Ο RogerPressman[49] χωρίζει τις μετρικές λογισμικού σε κατηγορίες, ανάλογα με τη φάση ανάπτυξης του λογισμικού που εφαρμόζονται: στη φάση της ανάλυσης απαιτήσεων, του σχεδιασμού, της ανάπτυξης (μετρικές για τον πηγαίο κώδικα), του ελέγχου και της συντήρησης.

Ακολουθούν πίνακες όπου παρουσιάζονται ορισμένες από τις πιο γνωστές μετρικές αντικειμενοστραφούς λογισμικού.

[48] *Ζηκούλη Κ., "Αντικειμενοστραφείς μετρικές λογισμικού", Διπλωματική εργασία, 2000*
[49] *Roger S. Pressman, "Software Engineering: A Practitioner's Approach", 4th ed.,McGraw Hill 1997*

Average Method Size

Υπολογίζει το μέσο μήκος των μεθόδων ενός συστήματος.

Class Size

Μετρά το μέγεθος μίας κλάσης.

Class Cohesion

Μετρά τη συνοχή μιας κλάσης, κάτι που καθορίζεται από τις σχέσεις μεταξύ των μεθόδων της.

Class Coupling

Μετρά τον αριθμό των κλάσεων που συνδέονται και το ποσό της σύνδεσης μιας κλάσης με τις άλλες.

Direct Class Coupling

Μετρά το πλήθος των κλάσεων με τις οποίες μια κλάση είναι συνδεδεμένη, συμπεριλαμβανομένων και των κλάσεων που συνδέονται άμεσα με δηλώσεις κοινών μεταβλητών και μεταβίβαση μηνυμάτων.

Depth of the Inheritance Tree

Ορίζεται ως το μέγιστο μήκος από τη ρίζα σε ένα κόμβο του δένδρου. Όσο μεγαλύτερο είναι το νούμερο αυτό, τόσο μεγαλύτερη είναι και η πολυπλοκότητα του συστήματος.

System Size in Classes

Μετρά το συνολικό αριθμό των κλάσεων στο σύστημα.

Internal Privacy

Αναφέρεται στη χρήση συναρτήσεων που ενεργούν πάνω στα στιγμιότυπα μιας κλάσης.

Lack of Cohesion in Methods

Η συνοχή μιας κλάσης ορίζεται από το πόσο στενά σχετίζονται οι τοπικές μέθοδοι της κλάσης με τα τοπικά στιγμιότυπα των ιδιοτήτων της.

Για μια κλάση η μετρική αυτή ορίζεται ως το πλήθος των μη επικαλυπτόμενων τοπικών μεθόδων της κλάσης.

Length of OO Program

Η μέτρηση του μήκους του προγράμματος βασίζεται στον αριθμός των κλάσεων και των μεθόδων που αυτό διαθέτει.

Number of Independent Classes

Μετρά τον αριθμό των κλάσεων που δεν ανήκουν σε κάποια ιεραρχία και δε κληρονομούνται από καμία κλάση στο σύστημα.

Number of Multiple Inheritance

Μετρά τον αριθμό των στιγμιότυπων που υπάρχει πολλαπλή κληρονομικότητα.

Number of Ancestors

Μετρά τον αριθμό των διακριτών κλάσεων τις οποίες μια κλάση κληρονομεί.

Number of Children

Ορίζεται ως ο αριθμός των υποκλάσεων από τις οποίες αποτελείται μία κλάση.

Όσο μεγαλύτερος είναι ο αριθμός αυτός, τόσο εξασθενεί η αφαιρετικότητα της κλάσης-γονέα, ενώ αποδεικνύει ότι κάποιες υποκλάσεις δε θα πρέπει να θεωρούνται παιδιά της συγκεκριμένης κλάσης-γονέα.

Number of Hierarchies

Μετρά τον αριθμό των διακριτών ιεραρχιών κλάσεων στο σύστημα.

Number of Inline Methods

Μετρά τον αριθμό των inline μεθόδων μιας κλάσης.

A Class Total Number of All Methods

Μετρά το συνολικό αριθμό μεθόδων σε μια κλάση.

Percent of Potential Method uses actually reused

Ορίζεται ως

PP = (αριθμός πραγματικών συνολικών χρήσεων μεθόδου) /
(αριθμός πιθανών χρήσεων της μεθόδου)

Ratio of Methods per Class

Μετρά τον αριθμό των μεθόδων ανά κλάση.

Total Method Size

Μετρά το συνολικό μέγεθος μιας κλάσης, αθροίζοντας το μέγεθος των μεθόδων της.

Weighted Class Size

Ορίζεται ως

weighted class size = Number Of Ancestors + Total Method Size

Weighted Methods per Class

Ορίζεται ως το άθροισμα της πολυπλοκότητας των μεθόδων μίας κλάσης και υπολογίζεται από τον τύπο:

$WMC = \Sigma c_i$

όπου c_i η πολυπλοκότητα της μεθόδου i.

Response for a Class

Ορίζεται ως ο αριθμός των μεθόδων σε ένα 'response set', το οποίο είναι το σύνολο των μεθόδων που μπορεί να εκτελεστεί σε απάντηση ενός μηνύματος που έχει σταλεί από ένα αντικείμενο της κλάσης.

Όσο αυξάνει ο αριθμός των μεθόδων σε ένα 'response set', τόσο αυξάνεται και η πολυπλοκότητα της κλάσης.

Average Operation Size

Ορίζονται ως ο αριθμός των μηνυμάτων που στέλνονται από μία λειτουργία.

Όταν ο αριθμός αυτός αυξάνεται, σημαίνει ότι δεν έχει γίνει καλή κατανομή των 'υποχρεώσεων' εσωτερικά στην κλάση.

Operation Complexity

Μετρά την πολυπλοκότητα μία λειτουργίας – το νούμερο αυτό πρέπει να κρατάται χαμηλό.

6. Μοντέλα αξιοπιστίας λογισμικού

Η αξιοπιστία του λογισμικού (software reliability) είναι ένας από τους σημαντικότερους παράγοντες ποιότητας. Επιπλέον, η αξιοπιστία μπορεί να μετρηθεί άμεσα και να εκτιμηθεί χρησιμοποιώντας δεδομένα που έχουν συλλεχθεί κατά την ανάπτυξη του λογισμικού. Στατιστικά, η αξιοπιστία ορίζεται ως «η πιθανότητα της χωρίς σφάλματα λειτουργίας του λογισμικού σε ένα καθορισμένο περιβάλλον για ένα καθορισμένο χρονικό διάστημα».

Συνεπώς, μέτρο της αξιοπιστίας αποτελεί ο «μέσος χρόνος ανάμεσα στην εμφάνιση σφαλμάτων» (mean time between failure – MTBF), ο οποίος ορίζεται ως

MTBF = MTTF + MTTR

• MTTF (mean time to failure) είναι ο μέσος χρόνος μέχρι την εμφάνιση σφάλματος,

• MTTR (mean time to repair) είναι ο μέσος χρόνος που απαιτείται για τη διόρθωση του σφάλματος.

Η μέτρηση της αξιοπιστίας με τον τρόπο αυτό είναι πολύ πιο χρήσιμη από τη μέτρηση των ελαττωμάτων ανά 1000 γραμμές κώδικα (KLOC): Επειδή κάθε ελάττωμα που περιέχεται στο πρόγραμμα δεν προκαλεί τον ίδιο ρυθμό εμφάνισης σφαλμάτων, η απλή μέτρηση των λαθών μπορεί να οδηγήσει σε λανθασμένη εκτίμηση της αξιοπιστίας. Με βάση αυτό το μέτρο αξιοπιστίας μπορεί να οριστεί και η διαθεσιμότητα του συστήματος, ως εξής:

Διαθεσιμότητα = 100% * MTTF / (MTTF + MTTR)

Τα μοντέλα αξιοπιστίας λογισμικού εφαρμόζονται για τρεις κυρίως λόγους: πρόβλεψη, συγκριτική ανάλυση και έλεγχο της διαδικασίας ανάπτυξης. Η πρόβλεψη της αξιοπιστίας του λογισμικού πρέπει να γίνει κατά τη διάρκεια της καταγραφής των απαιτήσεων, καθώς πολλές φορές, ο μηχανικός λογισμικού πρέπει να εγγυηθεί στους χρήστες έναν ελάχιστο χρόνο λειτουργίας του λογισμικού χωρίς αστοχία. Τα μοντέλα αξιοπιστίας εφαρμόζονται και κατά τη διάρκεια της σχεδίασης του λογισμικού, καθώς η αξιοπιστία του τελικού προϊόντος (θα πρέπει να) είναι ένας σημαντικός παράγοντας για την επιλογή ανάμεσα σε εναλλακτικά σχέδια. Τέλος, κατά τη διαδικασία ανάπτυξης του λογισμικού, η σύγκριση των αποτελεσμάτων που έδωσαν οι έλεγχοι του λογισμικού με το

μοντέλο αξιοπιστίας αποτελεί σοβαρή ένδειξη του σταδίου ανάπτυξης. Από την ίδια σύγκριση είναι δυνατή και η εκτίμηση του απαιτούμενου χρόνου εκσφαλμάτωσης μέχρι να φτάσει το λογισμικό σε ένα αποδεκτό επίπεδο αξιοπιστίας.

Τα τελευταία 30 χρόνια έχει αναπτυχθεί ένας σημαντικός αριθμός μοντέλων αξιοπιστίας, τα οποία χρησιμοποιούν στοιχεία από τη θεωρία των πιθανοτήτων και χρησιμοποιούν δεδομένα που προέρχονται από τον έλεγχο παρόμοιων συστημάτων λογισμικού[50]. Στα μοντέλα αυτά, ορίζεται πρώτα η τυχαία μεταβλητή που θα μελετηθεί. Ανάλογα με τη μεταβλητή που θα οριστεί, έχουμε:

• Τα μοντέλα εμφάνισης σφαλμάτων, όπου η τυχαία μεταβλητή είναι ο αριθμός των ελαττωμάτων σε ένα πρόγραμμα,

• Τα μοντέλα αξιοπιστίας, όπου η τυχαία μεταβλητή είναι ο χρόνος μέχρι την αστοχία του λογισμικού.

Τα μοντέλα εμφάνισης σφαλμάτων επιχειρούν να εκτιμήσουν τον αριθμό των σφαλμάτων που απομένουν σε ένα πρόγραμμα, καθώς μια τέτοια εκτίμηση αποτελεί μέτρο της προόδου της διαδικασίας ανάπτυξης. Βασίζονται στις εξής δύο υποθέσεις:

• Ο κανονικοποιημένος αριθμός των ελαττωμάτων που βρίσκονται σε παρόμοια συστήματα λογισμικού είναι κατά προσέγγιση σταθερός. Ο κανονικοποιημένος αριθμός των ελαττωμάτων προκύπτει διαιρώντας το συνολικό αριθμό ελαττωμάτων $\mathbf{E_T}$με το πλήθος των εντολών του προγράμματος $\mathbf{I_T}$, με την προϋπόθεση ότι οι γλώσσες προγραμματισμού είναι συγκρίσιμες. Λέγοντας «κατά προσέγγιση σταθερός» εννοούμε ότι αρχικά μας είναι αρκετή μια «χονδρική» εκτίμηση του αριθμού των ελαττωμάτων (π.χ. με ακρίβεια 50%), η οποία θα γίνεται περισσότερο ακριβής κατά τη σταδιακή εφαρμογή του μοντέλου (έχετε υπόψη σας ότι ο αριθμός των ελαττωμάτων αποτελεί μια παράμετρο ενός στοχαστικού μοντέλου, όχι μια φυσική σταθερή). Για να ορίσουμε την «ομοιότητα» δύο προγραμμάτων πρέπει να θεωρήσουμε ότι οι δύο ομάδες ανάπτυξης έφεραν συγκρίσιμο «μείγμα ικανοτήτων», ότι χρησιμοποιήθηκαν παρόμοιες τεχνικές ελέγχου του λογισμικού, ότι κατά την ανάπτυξη των δύο προγραμμάτων υιοθετήθηκαν παρόμοιες μεθοδολογίες, ενώ η ανάπτυξη βρίσκεται στο ίδιο στάδιο, κ.ά.

---

[50]*M.L. Shooman (1983), Software Engineering. McGraw – Hill, Tokyo.*

• Ο κανονικοποιημένος ρυθμός εκκαθάρισης λαθών σε παρόμοια συστήματα λογισμικού είναι κατά προσέγγιση σταθερός. Ο κανονικοποιημένος ρυθμός εκκαθάρισης λαθών προκύπτει διαιρώντας το ρυθμό εκκαθάρισης λαθών r_T με το πλήθος των εντολών του προγράμματος I_T. Στην περίπτωση αυτή πρέπει να προσθέσουμε δύο προϋποθέσεις ομοιότητας δύο συστημάτων: (α) η σύνθεση, οι ικανότητες και ο χρόνος απασχόλησης των ομάδων ανάπτυξης και ελέγχου είναι παρόμοιες για τα δύο συστήματα και (β) η ένταση και ο χρόνος ελέγχου στη μονάδα χρόνου είναι συγκρίσιμοι για τα δύο συστήματα.

Χρησιμοποιώντας δεδομένα που έχουν συλλεχθεί κατά την ανάπτυξη παρόμοιων προγραμμάτων, μπορούμε, με βάση τις δύο αυτές υποθέσεις, να εκτιμήσουμε τον αριθμό των σφαλμάτων που απομένουν και στο πρόγραμμα που ελέγχουμε. Η βασική ποσότητα, η οποία εμφανίζεται σε όλους τους υπολογισμούς, είναι ο συνολικός αριθμός ελαττωμάτων E_T. Για την εκτίμηση της ποσότητας αυτής κατά τη διάρκεια του ελέγχου του λογισμικού έχουν αναπτυχθεί ειδικά μοντέλα, όπως τα ακόλουθα:

• Μοντέλα εισαγωγής ελαττωμάτων, σύμφωνα με τα οποία, ένας αριθμός K γνωστών ελαττωμάτων εισάγεται στο λογισμικό. Κατά τον έλεγχο, η πιθανότητα εντοπισμού m άγνωστων (πραγματικών) ελαττωμάτων από τα συνολικά M πραγματικά ελαττώματα μπορεί να συσχετιστεί με την πιθανότητα ανακάλυψης k από τα K γνωστά ελαττώματα που έχουν εισαχθεί στον κώδικα,

• Μοντέλα παλινδρόμησης, τα οποία σχεδιάζουν μια γραφική παράσταση των έως τώρα δεδομένων του ελέγχου, την οποία ανάγουν σε μια από τις γνωστές καμπύλες, με βάση την οποία επιχειρούν να προβλέψουν την μελλοντική πορεία της εκσφαλμάτωσης.

Τέλος, για τα μοντέλα αξιοπιστίας χρησιμοποιούνται δύο προσεγγίσεις για τον ορισμό ενός μοντέλου αξιοπιστίας: η μακροσκοπική και η μικροσκοπική. Τα μοντέλα της πρώτης κατηγορίας δε λαμβάνουν υπόψη τους τα ιδιαίτερα χαρακτηριστικά κάθε λογισμικού, αλλά εξετάζουν τον αριθμό των εντολών, τον αριθμό των ελαττωμάτων που έχουν διορθωθεί κ.λπ. και βασίζονται σε δεδομένα που προέρχονται από παρόμοια συστήματα, τα οποία είχαν αναπτυχθεί στο παρελθόν. Τα μικροσκοπικά μοντέλα βασίζονται σε λεπτομερειακή ανάλυση της δομής ελέγχου και των εντολών του λογισμικού. Τα μοντέλα αξιοπιστίας λογισμικού διακρίνονται σε αυτά που προβλέπουν την αξιοπιστία ως συνάρτηση ημερολογιακού χρόνου και σε αυτά που την εκτιμούν ως συνάρτηση χρόνου

επεξεργασίας. Η εμπειρία έχει δείξει ότι τα μοντέλα της δεύτερης κατηγορίας δίνουν καλύτερα συνολικά αποτελέσματα.

7. Πρότυπα ανάπτυξης και αξιολόγησης λογισμικού

Το πρώτο πρότυπο ISO αναφέρθηκε στην προηγούμενη ενότητα, όπου παρουσιάστηκε το πρότυπο ISO 9126, το οποίο προσδιορίζει τα χαρακτηριστικά που συνθέτουν αυτό που ονομάζουμε ποιότητα λογισμικού. Παρουσιάζοντας το σύστημα ποιότητας μίας επιχείρησης, αναφέραμε ότι αυτό βασίζεται σε κάποιο πρότυπο. Αυτό το πρότυπο μπορεί να είναι κάποιο διεθνές πρότυπο (από κάποιον οργανισμό τυποποίησης) ή ακόμα και μία εσωτερική για την επιχείρηση τεκμηριωμένη σύμβαση που καθορίζει ένα επίπεδο τυποποίησης (standardisation). Συνήθως, όμως, είναι κάποιο διεθνές πρότυπο με πιο πιθανό το ISO 9001. Επειδή αναφέρθηκε αρκετές φορές ο οργανισμός ISO, πριν συνεχίσουμε, ακολουθεί μία συνοπτική παρουσίασή του.

Το 1946, στο Λονδίνο πραγματοποιήθηκε μία συνάντηση εκπροσώπων από 25 χώρες, οι οποίοι αποφάσισαν να δημιουργήσουν ένα νέο διεθνή οργανισμό, με αντικείμενο το διεθνή συντονισμό και την ομοιογένεια των προτύπων σχεδόν κάθε εταιρείας και οποιουδήποτε τύπου βιομηχανίας. Το αποτέλεσμα αυτής της απόφασης ήταν η δημιουργία του διεθνούς οργανισμού για τυποποίηση (International Organisation for Standardisation) ο οποίος ξεκίνησε να λειτουργεί επίσημα στις 23 Φεβρουαρίου 1947. Η συντομογραφία ISO, που προηγείται στην κωδικοποίηση κάθε προτύπου αυτού του διεθνούς οργανισμού, παραπέμπει στα αρχικά του ονόματός του (International Organisation for Standardisation), αλλά η επιλογή της έγινε από το ελληνικό «ίσος», το οποίο είναι η ρίζα του προθέματος «ίσο» που εμφανίζεται σε ένα πλήθος όρων όπως «ισομετρία, ισονομία». Το πρώτο ISO πρότυπο δημοσιεύτηκε το 1951 με τίτλο «Standard Reference Temperature For Industrial Length Measurement».

Στις μέρες μας, πάνω από 130 χώρες έχουν υιοθετήσει και χρησιμοποιούν τα πρότυπα του οργανισμού ISO. Στην Ευρωπαϊκή Κοινότητα μόνο, πάνω από 75.000 επιχειρήσεις βασίζουν την τυποποίησή τους στα πρότυπα του ISO, στις Ηνωμένες Πολιτείες, πάνω από 10.000 επιχειρήσεις (ο αριθμός αυτός αυξάνει ραγδαία) και πάνω από 1.500 επιχειρήσεις στον Καναδά.

Ο οργανισμός ISO εκδίδει πρότυπα (standards) και οδηγίες (guidelines) για την εφαρμογή των προτύπων. Μερικά από τα πρότυπα του οργανισμού ISO που σχετίζονται με την ποιότητα και αναφέρθηκαν, ή θα αναφερθούν παρακάτω, είναι τα:

ISO 8402: Περιγράφει το λεξιλόγιο που χρησιμοποιείται όταν μιλάμε για ποιότητα, δηλαδή όλους τους ορισμούς που χρησιμοποιήσαμε (εγχειρίδια ποιότητας, αναφορές, διαδικασίες), ώστε να υπάρχει κοινό λεξιλόγιο σε όλους.
ISO 9126: Πρότυπο που περιγράφει πώς η ποιότητα λογισμικού μπορεί να διασπαστεί σε παράγοντες ποιότητας και κάθε παράγοντας σε επιμέρους χαρακτηριστικά, χωρίς να υπάρχει επικάλυψη χαρακτηριστικών ανάμεσα στους παράγοντες.
ISO 9001: Πρότυπο για διασφάλιση ποιότητας στη σχεδίαση, ανάπτυξη, εγκατάσταση ή παροχή υπηρεσιών (όχι μόνο λογισμικού, αλλά κάθε είδους προϊόντων).
ISO 9000 – 3: Οδηγίες για την εφαρμογή του προτύπου ISO 9001 στη σχεδία- ση, ανάπτυξη, προμήθεια (εγκατάσταση, πώληση) και συντήρηση του λογισμικού.
ISO 9002: Πρότυπο για διασφάλιση ποιότητας στην ανάπτυξη (που βασίζεται σε καθορισμένο σχέδιο), εγκατάσταση ή παροχή υπηρεσιών (όχι μόνο λογισμικού, αλλά κάθε είδους προϊόντων).
ISO 9003: Πρότυπο για διασφάλιση ποιότητας στην τελική επιθεώρηση και έλεγχο του προϊόντος (όπου προϊόν μπορεί να είναι όχι μόνο το λογισμικό, αλλά κάθε είδους προϊόν).
ISO 9004: Οδηγίες για τη διοίκηση ενός συστήματος ποιότητας, δηλαδή για το πώς μπορεί να αναπτυχθεί και να εφαρμοστεί ένα σύστημαποιότητας.

Από το πρότυπο ISO 8402 χρησιμοποιήσαμε αρκετούς ορισμούς σε αυτό το βιβλίο, ενώ για το ISO 9126 μιλήσαμε στο κεφάλαιο 4. Ακολούθως, θα αναφερθούμε στη σειρά ISO 900x (όπου x = 1,2,3) και στην οδηγία ISO 9000 – 3 που αφορά στο λογισμικό. Πολλές φορές όλα τα πρότυπα αυτής της σειράς αναφέρονται και ως ISO 9000 (κακώς κατά την άποψή μου). Η οδηγία ISO 9004 μπορεί και πρέπει να χρησιμοποιείται συνοδευτικά με κάθε πρότυπο ISO 900x, ώστε να βοηθά στη δημιουργία και διαχείριση του συστήματος ποιότητας. Από τους παραπάνω ορισμούς των προτύπων είναι φανερό ότι το ISO 9001 είναι το πιο περιεκτικό και αυστηρό πρότυπο της σειράς. Κατά κάποιο τρόπο καλύπτει όλα τα απαιτούμενα στοιχεία που αναφέρονται στα ISO 9002 και ISO 9003. Η διαφορά του ISO 9001 με το ISO 9002 είναι ως προς την επέκταση του συστήματος ποιότητας και στη σχεδίαση. Αντίθετα, το ISO 9002 προϋποθέτει έτοιμα σχέδια στα οποία θα βασιστεί η ανάπτυξη. Το ISO 9002 είναι με τη σειρά του πολύ πιο εκτεταμένο από το ISO 9003, με βασικότερη διαφορά ότι το ISO 9003 περιορίζει

τις απαιτήσεις ποιότητας στις επιθεωρήσεις και στις δοκιμές και όχι στις δραστηριότητες που επηρεάζουν τηνποιότητα.

Όπως αναφέραμε, το πρότυπο ISO 9001 είναι γενικό και όχι εξειδικευμένο για το λογισμικό. Η πιστοποίηση μιας επιχείρησης ή βιομηχανίας με το πρότυπο ISO 9001 αποδεικνύει τη δέσμευσή της στην ποιότητα. Η ανάπτυξη (ή παραγωγή), σύμφωνα με τους κανόνες του προτύπου, αφορά σε ολόκληρη την εταιρεία και απαιτεί προσπάθεια από την αρχή της διαχείρισης του έργου μέχρι την ολοκλήρωση του προϊόντος. Κάθε επιχείρηση που πιστοποιείται με ISO 9001 ελέγχεται τόσο την πρώτη φορά, ώστε να πιστοποιηθεί, όσο και περιοδικά (κάθε έξι ή δώδεκα μήνες) από έναν τελείως ανεξάρτητο με την εταιρεία ελεγκτή, ο οποίος ανήκει σε κάποιο οργανισμό εξουσιοδοτημένο να πιστοποιεί με ISO (στην Ελλάδα υπάρχουν περίπου 10 τέτοιοι οργανισμοί: ο ΕΛΟΤ που είναι ο δημόσιος «Ελληνικός Οργανισμός Τυποποίησης», και οι υπόλοιποι που είναι ιδιωτικοί). Ο ελεγκτής κατευθύνεται από αυστηρούς διεθνείς κώδικες και συμφωνίες που υπαγορεύουν τις μεθόδους ελέγχου και τους περιορισμούς ποιότητας, έτσι ώστε να επιβεβαιώνεται η διασφάλιση της ποιότητας σε όλους τους τομείς της εταιρείας. Εάν ο ελεγκτής διαπιστώσει παρέκκλιση, τότε αφαιρείται η πιστοποίηση που έχει δοθεί στην εταιρεία. Επιπλέον, το πρότυπο ISO απαιτεί την ύπαρξη προγράμματος αυτοαξιολόγησης στις πιστοποιημένες εταιρείες, το οποίο θα είναι υπεύθυνο για τον έλεγχο των ποιοτικών απαιτήσεων και του επιπέδου εφαρμογής τους. Σε αυτό το σημείο πρέπει να τονιστεί ότι, γενικά, τα πρότυπα (και κατά συνέπεια και το ISO 9001) παρέχουν ένα σκελετό για το σύστημα ποιότητας της επιχείρησης και καθορίζουν το «τι» πρέπει να γίνει. Δεν καθορίζουν το «πώς» πρέπει να γίνει κάτι. Ο τρόπος ενέργειας αποτελεί έργο της επιχείρησης (όσον αφορά στο σχεδιασμό και την εφαρμογή κάποιας αποτελεσματικής διεργασίας ανάπτυξης, μέσα στο πλαίσιοτου προτύπου ISO 9001). Το ISO 9001 καθορίζει είκοσι τυποποιημένα στοιχεία, που λειτουργούν ως απαιτήσεις του συστήματος ποιότητας και παρατίθενται στη συνέχεια. Οι απαιτήσεις που θα οριστούν για το σύστημα ποιότητας στοχεύουν στην αποφυγή ανωμαλιών σε όλα τα στάδια της ανάπτυξης, από τον αρχικό σχεδιασμό του προϊόντος (του λογισμικού για την περίπτωσή μας) μέχρι την εγκατάστασή του στο χώρο του πελάτη και την παροχή των τελικών υπηρεσιών (όπως είναι η εκπαίδευσηγια τη χρήση του λογισμικού). Οι απαιτήσεις του ISO 9001 δίνονται συνοπτικά:

1. Ευθύνη διοίκησης (management responsibility), που περιλαμβάνει την πολιτική ποιότητας (quality policy) την οργάνωση, την ευθύνη και εξουσιοδότηση, τον έλεγχο των πηγών (πρώτων υλών για τη βιομηχανία) και του προσωπικού.

2. Σύστημα ποιότητας (quality system).

3. Ανασκόπηση συμβάσεων (contract review).

4. Έλεγχος σχεδιασμού (design control), που περιλαμβάνει τον αρχικό σχεδιασμό και ανάπτυξη των σχεδίων, τις οργανωτικές και τεχνικές αλληλοσυνδέσεις (organizational and technical interfaces), τα στοιχεία εισόδου και εξόδου και τονέλεγχο αλλαγών.

5. Έλεγχος εγγράφων (document control), που περιλαμβάνει την έγκριση και έκδοση εγγράφων (document approval and issue) και τις αλλαγές ή μετατροπές εγγράφων.

6. Αγορές (purchasing), που περιλαμβάνει την αξιολόγηση των υπεργολάβων(assessment of sub – contractor) και τα δεδομένα σχετικά με την αγοραστική ικανότητα.

7. Έλεγχος του προϊόντος που παρέχεται από τον προμηθευτή (purchaser supplied product).

8. Αναγνώριση ταυτότητας και ανιχνευσιμότητα του προϊόντος (product identification and traceability).

9. Έλεγχοςτηςδιεργασίαςανάπτυξης (process control).

10. Επισκόπηση και πειραματικός έλεγχος (inspection and testing), που περιλαμβάνει την επισκόπηση και τον πειραματικό έλεγχο κατά την ανάπτυξη (in – process inspection and testing), την τελική επισκόπηση και τον τελικό έλεγχο (final inspection and testing) και την τήρηση αρχείων για την επισκόπηση και τον πειραματικό έλεγχο (inspection and test records).

11. Έλεγχος εξοπλισμού επισκόπησης, υπολογισμών και μετρήσεων (inspection, measuring and test equipment).

12. Κατάσταση ελέγχων και επισκοπήσεων (inspection and test status).

13. Έλεγχος μη συμμορφούμενων προϊόντων (control of nonconforming product).

14. Διορθωτικές και προληπτικές ενέργειες (corrective and preventive actions).

15. Διακίνηση, αποθήκευση, συσκευασία και διανομή (handling, storage, packaging and delivery).

16. Αρχείαγιατηνποιότητα (quality records).
17. Εσωτερικές περιοδικές επιθεωρήσεις ποιότητας (internal quality audits).
18. Εκπαίδευση (training).
19. Εξυπηρέτηση (servicing).
20. Στατιστικές τεχνικές (statistical techniques).

Σε αυτό το σημείο πρέπει να τονιστεί ότι το πρότυπο ISO 9001 δεν περιέχει τεχνικές που θα μπορούσαν να ενταχθούν στο σύστημα ποιότητας κάποιας επιχείρησης. Κάτι τέτοιο δεν θα ήταν εφικτό, αφού το πρότυπο σκόπιμα είναι τόσο γενικό, ώστε να απευθύνεται και σε μία βιομηχανία που, για παράδειγμα, παράγει χρώματα, αλλά και σε μία επιχείρηση που αναπτύσσει λογισμικό, ή σε μία βιοτεχνία που σχεδιάζεικαι παράγει υποδήματα. Το πρότυπο παρέχει οδηγίες προς την επιχείρηση για το πού πρέπει να χρησιμοποιεί τεχνικές ελέγχου και ποιες απαιτήσεις πρέπει να εκπληρώνει, ώστε να είναι κατάλληλη να πιστοποιηθεί με το ISO 9001.

Ακριβώς επειδή το πρότυπο ISO 9001 είναι πολύ γενικό, πολλές επιχειρήσεις και κυρίως οι επιχειρήσεις που αναπτύσσουν λογισμικό, αντιμετωπίζουν πρόβλημα στην προσπάθειά τους να το εφαρμόσουν. Αυτό συμβαίνει γιατί το λογισμικό παρουσιάζει μια ιδιαιτερότητα που οφείλεται σε τρεις αιτίες:

• Η φύση των διαδικασιών ανάπτυξης στο λογισμικό διαφέρει, στην ουσία της, από αυτή στη βιομηχανική παραγωγή. Αυτό που στο λογισμικό είναι η βάση της ανάπτυξης, στη βιομηχανική παραγωγή θα έμοιαζε ως διαδικασία σχεδίασης. Αντίστοιχα, αυτό που στη βιομηχανική παραγωγή ορίζεται ως ανάπτυξη (δηλαδή η αναπαραγωγή σε πολλά αντίγραφα ενός τυποποιημένου προϊόντος), στο λογισμικό είναι ασήμαντη (αφού το λογισμικό «αναπαράγεται» σε αντίτυπα με σχεδόν μηδενικόκόστος και προσπάθεια και η βιομηχανική ανάπτυξη περιορίζεται μόνο στην αναπαραγωγή των εγχειριδίων και των μέσων στα οποία αποθηκεύεται το λογισμικό).

• Ο κύκλος ζωής του λογισμικού, δηλαδή το πλαίσιο ανάπτυξης της διαδικασίας, παίζει έναν ιδιαίτερα βασικό ρόλο. Για αυτό πρέπει να δοθεί σε αυτό η απαιτούμενη έμφαση, δηλαδή να καθοριστεί ο κύκλος ζωής, ως περιγραφή της διαδικασίας ανάλυσης – σχεδιασμού – ανάπτυξης – ελέγχου – συντήρησης, και να καταχωρηθεί σε ένα κατευθυντήριο έγγραφο που εντάσσεται στις απαιτήσεις του προτύπου.

• Κάποιες δραστηριότητες της παραγωγής λογισμικού, όπως η διαχείριση εκδόσεων (που δεν είναι σημαντική, ή λείπει από την παραγωγή υλικών αγαθών), παίζουν έναν ιδιαίτερα βοηθητικό ρόλο στην ανάπτυξη και για αυτό πρέπει να τονιστεί η σημασία τους. Επίσης, σε αυτές τις δραστηριότητες πρέπει να αναφέρονται όροι με τους οποίους οι προγραμματιστές να είναι εξοικειωμένοι, αφού οι όροι του ISO9001 είναι προσανατολισμένοι στη βιομηχανική παραγωγή υλικών αγαθών.

Για τους παραπάνω λόγους δημιουργήθηκε η οδηγία ISO 9000 – 3, που χρησιμοποιείται για την εξειδίκευση του ISO 9001 στην ανάπτυξη λογισμικού. Ολοκληρώνοντας για το ISO 9001 και την οδηγία ISO 9000 – 3, θα αναφέρουμε πλεονεκτήματα αλλά και μειονεκτήματα από την εφαρμογή του σε κάποια επιχείρηση. Το ISO 9001 αποτελεί πρότυπο ποιότητας για αποτελεσματική διαχείριση των διεργασιών ανάπτυξης. Με άλλα λόγια, εμπίπτει στην κατηγορία των μεθοδολογιών βελτίωσης διεργασίας (process improvement methodologies). Πιο συγκεκριμένα, καθορίζει τα απαραίτητα χαρακτηριστικά για τη σωστή ανάπτυξη του προϊόντος, θέτοντας την αναγκαιότητα εκπλήρωσης κάποιων καθορισμένων απαιτήσεων ποιότητας.

Στόχος του είναι η βελτίωση της διεργασίας ανάπτυξης. Όπως αναφέραμε, το ISO 9001 καθορίζει το «τι» πρέπει να εκτελέσει ένας οργανισμός και όχι το «πώς» να τοπραγματοποιήσει, συνεπώς αποτελεί ένα πλαίσιο, μια κατευθυντήρια γραμμή μοντελοποίησης. Το γεγονός αυτό δίνει το πλεονέκτημα σε μία επιχείρηση να έχει έναμεγάλο φάσμα επιλογών για το πώς να χειριστεί την κάθε απαίτηση ποιότητας (πουθέτει το συγκεκριμένο πρότυπο), σύμφωνα με τις ανάγκες και ιδιαιτερότητες της επιχείρησης. Βέβαια, αυτό το χαρακτηριστικό του προτύπου αποτελεί, ανάλογα με τις συνθήκες, και μειονέκτημα, αφού δεν περιλαμβάνει όλες εκείνες τις λεπτομέρειες που είναι απαραίτητο να ληφθούν υπόψη για την ολοκλήρωση με επιτυχία μιας λειτουργίας. Παρόλα αυτά, οι περισσότερες επιχειρήσεις προτιμούν να έχουν πολλές επιλογές στον καθορισμό του συστήματος ποιότητάς τους.

Το ISO 9001 αποτελείται από ένα σύνολο απαιτήσεων ποιότητας, είναι δομημένο δηλαδή με βάση μία συνεχή αρχιτεκτονική (continuous architecture), γεγονός που σημαίνει ότι το μοντέλο δεν είναι δομημένο σε επίπεδα. Η συνεχής αρχιτεκτονική προσφέρει μεγάλη ευελιξία, όσον αφορά στην εκτέλεση των δραστηριοτήτων, αφούεπιτρέπει στον κάθε οργανισμό να αποφασίσει για την

προτεραιότητα και τη διάταξη των διεργασιών. Επιπλέον, είναι δυνατή η προσθήκη νέων απαιτήσεων στοσύστημα ποιότητας της επιχείρησης, αν αυτό κρίνεται απαραίτητο, χωρίς να αλλάζει η πιστοποίηση κατά ISO.

Τέλος, το βασικότερο πλεονέκτημα της πιστοποίησης με ISO 9001 είναι ότι η επιχείρηση βεβαιώνει με αυτή την πιστοποίηση την ικανότητά της να τηρεί διαδικασίες ποιότητας που της εξασφαλίζουν την αποδοχή σε συμβάσεις έργων, ή στην αγορά των προϊόντων που παράγει. Αναλύοντας τα μειονεκτήματα, πρέπει να αναφέρουμε ότι το βασικότερο μειονέκτημα που προκύπτει από την εφαρμογή του προτύπου ISO 9001 είναι ότι αυτόκαθορίζει τις ελάχιστες απαιτήσεις ενός συστήματος ποιότητας, δηλαδή δεν αποτελεί ένα πλήρες σύστημα διασφάλισης ποιότητας για έναν οργανισμό. Επίσης, λόγω της γενικότητάς του, δεν λαμβάνει υπόψη του τις ιδιαιτερότητες του κάθε προϊόντος,ώστε να είναι ικανό να προβλέψει κάθε πιθανό πρόβλημα και να θέσει μια πορεία αντιμετώπισής του. Επίσης, το ISO 9001 δεν δίνει την απαραίτητη έμφαση στη συνεχή βελτίωση των διεργασιών ανάπτυξης της επιχείρησης. Αυτό, ίσως, εμπεριέχεται στην απαίτηση ποιότητας «Διορθωτικές και προληπτικές ενέργειες (corrective andpreventive actions), αλλά επειδή ο ανταγωνισμός που επικρατεί σήμερα θέτει τη βελτίωση της ποιότητας ως τον πιο σημαντικό στόχο μιας επιχείρησης, θεωρείται ότι ένα πρότυπο ποιότητας πρέπει να εστιάζει σε αυτό το σημείο πιο δυναμικά. Τέλος, το ISO 9001 διακρίνεται από μία ασάφεια σχετικά με το ρόλο της μέτρησης σε ένα σύστημα διαχείρισης ποιότητας. Με άλλα λόγια, θέτει μεν την απαίτηση ένας οργανισμός να τεκμηριώνει τα αντικείμενα ποιότητας, αλλά δεν θεωρεί ότι είναι αναγκαία η ποσοτικοποίηση τους.

Το CMM είναι ένα μοντέλο αξιολόγησης που εξελίχθηκε σταδιακά σε πρότυπο όχι μόνο αξιολόγησης, αλλά και πρότυπο ποιότητας λογισμικού ακολουθώνταςτις οδηγίες για κάθε βασική περιοχή της διαδικασίας (key process area) που θα συζητήσουμε . Η βασική ιδέα που οδήγησε στο CMM είναι η βελτίωση της διαδικασίας ανάπτυξηςλογισμικού και η συνεπέστερη εφαρμογή της στα πλαίσια μίας επιχείρησης. Το πρόβλημα, όμως, είναι ότι η θέσπιση στόχων για τη βελτίωση αυτής της διαδικασίαςσχετίζεται άμεσα με την ωριμότητα (maturity) της κάθε επιχείρησης. Πιο απλά,μόνο μία ώριμη επιχείρηση ανάπτυξης λογισμικού έχει την ικανότητα (capability)διαχείρισης της ανάπτυξης του λογισμικού, καθώς και βελτίωσης των σχετικών διαδικασιών ανάπτυξης, με σκοπό την ικανοποίηση του πελάτη. Αντίθετα, σε μία ανώριμη επιχείρηση, η διαδικασία ανάπτυξης

λογισμικού βασίζεται στον αυτοσχεδιασμό των μηχανικών κατά τη διάρκεια του έργου. Τα παραπάνω δείχνουν ότι είναι επιτακτική η ανάγκη ανάπτυξης ενός πλαισίου ωρίμανσης, το οποίο θα προσδιορίζει την ικανότητα της επιχείρησης να καθορίζει και να διαχειρίζεται τη διαδικασία ανάπτυξης λογισμικού. Το πλαίσιο αυτό περιγράφει ένα εξελικτικό μονοπάτι από τυχαίες και χαοτικές διαδικασίες ανάπτυξης σε ώριμες και πειθαρχημένες διαδικασίες ανάπτυξης λογισμικού.

Με δεδομένη την ύπαρξη της κατάστασης αυτής το Νοέμβριο του 1986, το Ινστιτούτο Τεχνολογίας Λογισμικού (Software Engineering Institute) του Carnegie Mellon University ξεκίνησε την ανάπτυξη πλαισίου ωρίμανσης διαδικασιών, με στόχο την προσφορά βοήθειας στους οργανισμούς προς την κατεύθυνση της βελτίωσης της ικανότητάς τους να αναπτύσσουν λογισμικό. Έπειτα από τέσσερα χρόνια έρευνας σχετικά με το πλαίσιο ωριμότητας της διαδικασίας λογισμικού, το Ινστιτούτο Τεχνολογίας Λογισμικού αναπτύσσει και εξελίσσει το πλαίσιο αυτό στο μοντέλο ωριμότητας ικανότητας (CMM) για λογισμικό. Η έκδοση 1.1 του CMM[51], στην οποία αναφερόμαστε δημοσιεύτηκε το 1993.

Συνοπτικά, το CMM είναι ένα μοντέλο που παρέχει συστηματική δόμηση ενός συνόλου εργαλείων (συμπεριλαμβανομένου ενός ερωτηματολογίου αξιολόγησης της ωριμότητας μίας επιχείρησης), με αντικειμενικό σκοπό τη βελτίωση της ικανότητας παραγωγής λογισμικού. Το ίδιο το μοντέλο αποτελεί τη βάση για τη βελτίωση της διαδικασίας ανάπτυξης λογισμικού. Αν θέλαμε να το ορίσουμε απλά, θα λέγαμε ότι τοCMM είναι ένα μοντέλο με το οποίο μπορεί μία επιχείρηση να αξιολογήσει πόσοώριμη είναι σχετικά με την ικανότητά της να αναπτύσσει λογισμικό. Η αξιολόγηση βασίζεται σε κάποια κριτήρια (ένα αναλυτικό ερωτηματολόγιο), κάτι αντίστοιχο δηλαδή με τις 20 απαιτήσεις του ISO 9001. Πέρα από αυτό, όμως, το CMM παρέχει και μία σειρά από βασικές περιοχές της διαδικασίας (key process areas) όπως τις ονομάζει, οι οποίες δίνουν οδηγίες για το τι πρέπει να έχει εντάξει μία επιχείρησηστη διαδικασία ανάπτυξης ώστε να βρίσκεται σε κάποιο επίπεδο ωριμότητας. Αυτέςοι βασικές περιοχές της διαδικασίας, ουσιαστικά, αποτελούν ένα κλιμακωτό πρότυπο ποιότητας που βοηθά την κάθε επιχείρηση να βελτιώνεται βήμα με βήμα. Αυτή η δυνατότητα του CMM να βοηθά την επιχείρηση να εξελίσσεται σταδιακά, σεσυνδυασμό με την εξειδίκευσή του αποκλειστικά για την ανάπτυξη λογισμικού, τοέχουν καταστήσει

---

[51] *M. Paulk et al., «Capability Maturity Model for Software (Version 1.1)», Carnegie Mellon University – Software Engineering Institute, Technical Report, CMU/SEI – 93 – TR – 024, (1993).*

ως το κυρίαρχο πρότυπο στα περισσότερα συστήματα ποιότηταςεπιχειρήσεων που αναπτύσσουν λογισμικό παγκοσμίως. Έτσι πέρα από αυτό που ήτανόταν αρχικά ξεκίνησε (ένα μοντέλο αξιολόγησης της ωριμότητας των επιχειρήσεων),το CMM σήμερα καθορίζει τις διαδικασίες ποιότητας των επιχειρήσεων.

Το ISO 9001 με το CMM έχουν σημαντικές ομοιότητες, αλλά και διαφορές[52]. Βασική τους ομοιότητα είναι ότι και τα δύο αποτελούν πρότυπα ποιότητας για αποτελεσματική διαχείριση των διαδικασιών ανάπτυξης λογισμικού. Και τα δύο παρέχουν μεθοδολογίες βελτίωσης της διαδικασίας ανάπτυξης (process improvementmethodologies), δηλαδή, καθορίζουν τα απαραίτητα χαρακτηριστικά για τη «σωστή» ανάπτυξη του προϊόντος, προς την κατεύθυνση της εκπλήρωσης κάποιων απαιτήσεων ποιότητας.

Άλλη ομοιότητά τους έγκειται στον τρόπο εφαρμογής τους. Μία επιχείρηση πιστοποιείται με ISO 9001 από κάποιον εξωτερικό ελεγκτή και η πιστοποίηση αυτή επιβεβαιώνεται από περιοδικούς ελέγχους. Μία επιχείρηση αξιολογείται, ως προς το επίπεδο ωριμότητάς της στο CMM, πάλι από κάποιον εξωτερικό ελεγκτή και αυτή η αξιολόγηση επίσης επαναλαμβάνεται περιοδικά. Σημαντική ομοιότητα των ISO 9001 και CMM είναι το γεγονός ότι και τα δύο δεν καθορίζουν μία συγκεκριμένη διαδικασία ανάπτυξης. Αντίθετα προσφέρουν καθοδήγηση στους υπεύθυνους για την ανάπτυξη διαδικασιών, για τον καθορισμό των μεθόδων τους και την εφαρμογή των κατάλληλων διαδικασιών για τη διαχείριση της ανάπτυξης. Πιο απλά, τόσο το CMM όσο και το ISO 9001 καθορίζουν «τι» πρέπει να κάνει μία επιχείρηση και όχι «πώς» να το κάνει. Στο σημείο αυτό, πρέπει να τονιστεί ότι το CMM είναι πολύ πιο αναλυτικό και παρέχει πολύ πιο συγκεκριμένες κατευθύνσεις από το ISO 9001 που είναι πιο γενικό. Χαρακτηριστικό των ομοιοτήτων είναι ότι, ανάμεσα σε πολλές από τις απαιτήσεις του ISO 9001 και τις βασικές περιοχές του CMM, υπάρχει άμεση αντιστοίχηση. Βέβαια, δεν υπάρχει ένα προς ένα αντιστοιχία, αφού τότε τα δύο μοντέλα θα ταυτίζονταν.

Βασική διαφορά των δύο προτύπων είναι ότι, ενώ το CMM είναι εξειδικευμένο αποκλειστικά για το λογισμικό, το ISO 9001 είναι ένα γενικό πρότυπο που περιλαμβάνει κάθε είδους προϊόν (συμπεριλαμβανομένου φυσικά και του λογισμικού) καθώς και την παροχή υπηρεσιών προς τους πελάτες. Για αυτό το λόγο το CMM, είναι πολύ πιο ειδικό και πιο εύκολα εφαρμόσιμο στο

[52]*M. Paulk, «A Comparison of ISO 9001 and the Capability Maturity Model for Software», Carnegie Mellon University – Software Engineering Institute, Technical Report, CMU/SEI – 94 – TR – 012, (1994).*

λογισμικό. Άλλη διαφορά τους έγκειται στη φιλοσοφία βελτίωσης. Πιο συγκεκριμένα, το CMM δίνει έμφαση στην ανάγκη για συνεχή βελτίωση των διαδικασιών ανάπτυξης. Ενώ το ISO 9001 καθορίζει τις ελάχιστες απαιτήσεις ενός συστήματος ποιότητας, το CMM αντιμετωπίζει το ζήτημα της συνεχούς βελτίωσης με πολύ μεγαλύτερη σαφήνεια και θέτει, ως ανώτερο επίπεδο ωριμότητας μιας επιχείρησης, αυτό στο οποίο το πρόγραμμα ποιότητας της επιχείρησης βελτιώνεται συστηματικά.

Διαφορά, επίσης, αποτελεί το γεγονός ότι, στο CMM, μία επιχείρηση μπορεί να βρίσκεται σε ένα από πέντε διακριτά επίπεδα ωριμότητας και κάθε επίπεδο δημιουργεί τις προϋποθέσεις για τα επόμενα επίπεδα, με στόχο την εφαρμογή των διαδικασιών ανάπτυξης αποτελεσματικά και αποδοτικά. Αντίθετα, το ISO 9001 καθορίζει τις απαιτήσεις του συστήματος ποιότητας με τη μορφή μιας σειράς απαιτήσεων, τις οποίες η επιχείρηση είτε ικανοποιεί, είτε όχι.

Πέρα από τις παραπάνω διαφορές, υπάρχουν και μικροδιαφορές ανάμεσα στις απαιτήσεις του ISO 9001 και στις βασικές περιοχές του CMM, που συνήθως αναφέρονται σε ελλείψεις του ISO 9001 (όπως για παράδειγμα στον καθορισμό διαδικασιώνμέτρησης), αλλά δεν είναι σημαντικές. Μία επιχείρηση μπορεί να είναι πιστοποιημένη με ISO 9001 και ταυτόχρονα να έχει αξιολογηθεί σε κάποιο επίπεδο του CMM.Συνήθως, μία επιχείρηση πρέπει να φτάσει στο 2ο επίπεδο του CMM και να διαθέτει κάποιες βασικές περιοχές και του 3ου επίπεδου (θα μιλήσουμε αργότερα για ταεπίπεδα του CMM), ώστε να μπορέσει να πιστοποιηθεί με ISO 9001. Στην Ευρώπη,οι περισσότερες επιχειρήσεις που αναπτύσσουν λογισμικό είναι αξιολογημένες μεCMM και πιστοποιημένες με ISO 9001.

Μία επιχείρηση που αξιολογείται με CMM μπορεί να βρίσκεται σε ένα από πέντε επίπεδα. Η αξιολόγηση με CMM βοηθά την επιχείρηση να «ξέρει πού βρίσκεται», έτσι ώστε να θέτει ρεαλιστικούς στόχους για τη βελτίωσή της. Αν μια επιχείρηση δενγνωρίζει πού βρίσκεται, έτσι ώστε να προδιαγράψει πού θέλει να φτάσει, τότε θα βλέπει παντού επιτυχίες, χωρίς στην πραγματικότητα να πετυχαίνει τίποτε. Τα πέντε επίπεδα ωριμότητας του CMM είναι τα:

1. Αρχικό (Initial)
2. Επαναλαμβανόμενο (Repeatable)
3. Καθορισμένο (Defined)
4. Διοικούμενο (Managed)
5. Βελτιστοποίησης (Optimising)

Με εξαίρεση το επίπεδο 1, κάθε επίπεδο αποτελείται από ένα σύνολο βασικών περιοχών μιας διαδικασίας (key process areas), στις οποίες μία επιχείρηση πρέπεινα εστιάσει για τη βελτίωση των διαδικασιών ανάπτυξης λογισμικού. Αναλυτικά,μία βασική περιοχή μιας διαδικασίας περιέχει μία δέσμη συσχετιζόμενων δραστηριοτήτων, οι οποίες, όταν εκτελεστούν συλλογικά, οδηγούν στην επίτευξη ενός συνόλου από στόχους σημαντικούς για την απόκτηση ικανότητας διαχείρισης της διαδικασίας ανάπτυξης. Κάθε βασική περιοχή περιλαμβάνει ένα σύνολο από βασικέςπρακτικές (key practices), οι οποίες πιστοποιούν αν η διαδικασία και η βασικήπεριοχή είναι αποτελεσματική, επαναλαμβανόμενη και διαρκής.Έτσι, το μοντέλο του CMM παρέχει στην επιχείρηση 5 επίπεδα, βασικές περιοχές (αντίστοιχες με τις απαιτήσεις του ISO 9001) για κάθε επίπεδο και βασικές πρακτικές (που βοηθούν στην ανάλυση των βασικών περιοχών σε μεγάλο βαθμό λεπτομέρειας, πράγμα που έλειπε από το ISO 9001) για κάθε περιοχή. Σήμερα, στον κόσμο, πολύ λίγες επιχειρήσεις λογισμικού βρίσκονται στο επίπεδο 5. Στην Ελλάδα,οι περισσότερες επιχειρήσεις είναι στο επίπεδο 1 με λίγες στο επίπεδο 2, ελάχιστεςστο 3 και καμία στο 4 και 5.

8. Συμπεράσματα

Οι συστηματικές ανασκοπήσεις συνεισφέρουν σε μια πιο αντικειμενική προσέγγιση της βιβλιογραφίας συγκριτικά με τις συμβατικές, αφηγηματικές ανασκοπήσεις. Με τον τρόπο αυτό, συμβάλλουν στην αποσαφήνιση θεμάτων όπου υφίσταται αβεβαιότητα, αλλά και στην αποκάλυψη πεδίων όπου η έρευνα ενδέχεται να είναι ελλιπής. Η μετα-ανάλυση, σε περιπτώσεις όπου αρμόζει, αυξάνει την ακρίβεια του υπολογιζόμενου μεγέθους του αποτελέσματος, περιορίζοντας ταυτόχρονα την πιθανότητα ψευδώς αρνητικών αποτελεσμάτων.

Ανατρέχοντας στη βιβλιογραφία ο ερευνητής θα διαπιστώσει ότι κατά καιρούς έχουν δοθεί διάφοροι ορισμοί για την ποιότητα. Μάλιστα, εύκολα μπορεί να παρατηρήσει ότι οι δύο φράσεις-κλειδιά που πρέπει να χαρακτηρίζουν τον ορισμό της ποιότητας είναι α) τα ποιοτικά χαρακτηριστικά και β) οι απαιτήσεις του πελάτη-χρήστη. Η εξασφάλιση της ποιότητας επιπλέον ήταν ανέκαθεν συνυφασμένη με τις μετρήσεις. Ως μέτρηση ορίζεται η διαδικασία με την οποία αριθμοί ή σύμβολα αντιστοιχίζονται σε ιδιότητες οντοτήτων του πραγματικού κόσμου, έτσι ώστε να τις περιγράφουν σύμφωνα με καθορισμένους κανόνες. Ο DeMacro μάλιστα αναφέρει ότι είναι αδύνατο να ελέγξεις ό,τι δεν μπορείς να μετρήσεις. Πόσο μάλλον αδύνατον είναι να στοχεύεις να εξασφαλίζεις την ποιότητα για κάτι το οποίο δεν μπορείς να μετρήσεις.

Όπως είναι γνωστό, η έννοια της ποιότητας λογισμικού είναι άρρηκτα συνδεδεμένη με τα ποιοτικά χαρακτηριστικά, τις απαιτήσεις των πελατών-χρηστών και τις προδιαγραφές που θα πρέπει να ικανοποιεί το λογισμικό. Βασικοί στόχοι της μεθοδολογίας είναι α) η έγκαιρη εκτίμηση της γνώμης των χρηστών για την ποιότητα λογισμικού, ώστε να διορθωθούν νωρίς τα πιθανά λάθη και να γίνουν βελτιώσεις σε πρώιμες φάσεις του κύκλου ζωής λογισμικού, β) η ενεργή συμμετοχή του τελικού χρήστη σε όλες τις φάσεις του κύκλου ζωής λογισμικού, αφού κάθε διαδικασία κατά την παραγωγή λογισμικού θα πρέπει να ακολουθεί τους κανόνες εξασφάλισης ποιότητας που έχουν θεσπιστεί στο πλάνο ποιότητας ενός έργου, γ) η μείωση της προσπάθειας που θα απαιτηθεί στη φάση της συντήρησης, αφού πιθανά προβλήματα θα έχουν διαφανεί σε προηγούμενες φάσεις, δ) η μείωση του κόστους για την εύρεσητης γνώμης του χρήστη για την ποιότητα ενός προϊόντος λογισμικού, αφού δεν είναι απαραίτητη η διεξαγωγή επαναληπτικών εξωτερικών μετρήσεων, ε) ο καθορισμός μετρήσιμων στόχων

κατά την παραγωγή και η παρακολούθηση των βελτιώσεων από έκδοση σε έκδοσή του και στ) η δυνατότητα εστίασης σε συγκεκριμένα χαρακτηριστικά της ποιότητας και ο καθορισμός απαιτούμενων ενεργειών.

Ιδιαίτερη βαρύτητα πρέπει να δίνεται στον καθορισμό των περιορισμών ανάμεσα στις τιμές των διαφόρων ποιοτικών χαρακτηριστικών, όπως αυτά τα αντιλαμβάνονταιοι χρήστες και ιδιαίτερα οι χρήστες με χαμηλότερο βαθμό εμπειρίας. Επιπλέον, οι εταιρείες παραγωγής λογισμικού που θα εντάξουν την προτεινόμενη μεθοδολογία στο πρόγραμμα εξασφάλισης ποιότητάς τους έχουν τη δυνατότητα να την προσαρμόσουν σύμφωνα με τις δικές τους ανάγκες.

Θεμελιώδη στοιχεία μιας συστηματικής ανασκόπησης είναι η διατύπωση του ερευνητικού ερωτήματος, ο προσδιορισμός κριτηρίων εισόδου, η ενδελεχής αναζήτηση της βιβλιογραφίας και η αποτίμηση της μεθοδολογικής ποιότητας των πρωτογενών εργασιών. Μεθοδολογικά χαρακτηριστικά που πρέπει να ελέγχονται στις δοκιμές .

Ωστόσο, κάποια συστηματικά σφάλματα μπορεί να προκύψουν στις ανασκοπήσεις και στις μετα-αναλύσεις των δοκιμών. Η πτωχή μεθοδολογική ποιότητα των πρωτογενών εργασιών μπορεί να υπονομεύσει την αξιοπιστία μιας συστηματικής ανασκόπησης.

Τα συστηματικά σφάλματα μπορούν επίσης να παραποιήσουν τα ευρήματά της, λόγω του ότι εργασίες με στατιστικώς σημαντικά αποτελέσματα είναι πιο πιθανό, όχι μόνο να δημοσιευθούν αλλά και να δημοσιευθούν χωρίς καθυστέρηση συγκριτικά με εκείνες που δεν καταλήγουν σε θετικά αποτελέσματα.

Με τη μεθοδολογία που περιγράφηκε στην παρούσα πτυχιακή, γίνεται φανερό ότι η ποιότητα του λογισμικού είναι δυνατό να εκφραστεί με συγκεκριμένα χαρακτηριστικά του πηγαίου κώδικα και με τον τρόπο αυτό να ποσοτικοποιηθεί και να μετρηθεί. Τα αποτελέσματα των εκάστοτε μετρήσεων ποιότητας δίνουν τη δυνατότητα στους μηχανικούς Η/Υ και τους προγραμματιστές να αναπτύσσουν λογισμικό που θα πληροί τα κριτήρια ποιότητας.

9. Βιβλιογραφία

Νικόλαος Αβούρης, «Εισαγωγή στην Επικοινωνία Ανθρώπου – Υπολογιστή», Εκδόσεις ΔΙΑΥΛΟΣ, (2000).

Γ. Βαρουφάκης, «Αρχαία Ελλάδα & Ποιότητα. Η ιστορία και ο έλεγχος των υλικών που σημάδεψαν τον ελληνικό πολιτισμό», Εκδόσεις Αίολος, (1996).

Ν. Ψύχας, «Διασφάλιση Ποιότητας: Διοίκηση της Ποιότητας», Ελληνικό Ανοικτό Πανεπιστήμιο, ΔΙΠ51/2, (2000).

Σ. Στεφανάτος, «Διασφάλιση Ποιότητας: Ολική Ποιότητα», Ελληνικό Ανοικτό Πανεπιστήμιο, ΔΙΠ51/3, (2000).

Βασίλειος Βεσκούκης, «Τεχνολογία Λογισμικού Ι: Αρχές Τεχνολογίας Λογισμικού», Ελληνικό Ανοικτό Πανεπιστήμιο, ΠΛΗ20/1, (2000).

Μιχάλης Ξένος και Δημήτρης Χριστοδουλάκης, «Τεχνολογία Λογισμικού:Αρχές και Μεθοδολογίες», Εκδόσεις Πανεπιστημίου Πατρών, (1994).

Μιχάλης Ξένος, «Μεθοδολογία ελέγχου και εξασφάλισης της ποιότητας λογισμικού βασισμένη στις μετρικές προϊόντος και στα εξωτερικά ποιοτικά χαρακτηριστικά του λογισμικού», Διδακτορική Διατριβή, Πανεπιστήμιο Πατρών, (1996).

Σταυρινούδης Δ., "Πειραματική μελέτη του βαθμού συσχέτισης εσωτερικών και εξωτερικών μετρικών ποιότητας λογισμικού",Διπλωματική Εργασία, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Πατρών, 1995

A. J. Albrecht and J. E. Gaffney, «Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation», IEEE Transactions on Software Engineering, (1983).

American Society for Quality Control, «Standard A3», (1978).

K. Arnold and M. Holler, «Quality Assurance. Methods and Technologies», McGraw – Hill, New York, (1995).

C. Aubrey, «Quality Management in Financial Service», Wheaton IL,Hitchcock Publishing, (1985).

B. Boehm et al., «Characteristics of Software Quality», North Holland,(1978).

B. Boehm, «Software Engineering Economics», Prentice–Hall, (1981).

B. Boehm, «Software Risk Management», IEEE Computer Society Press, (1989).

Frederick P. Brooks, «The Mythical Man–Month. Essays on Software Engineering», Anniversary edition, , Addison–Wesley, (1995).

S. D. Conte et al., «Software Engineering Metrics and Models», Benjamin Cummings, (1986).

Philip Crosby, «Quality is Free», New York, McGraw – Hill, (1979).

Philip Crosby, «Quality is Still Free», New York, McGraw – Hill, (1996).

B. Curtis et al., «A Field Study of the Software Design Process for Large Systems»,

IEEE (1986), Software Verification and Validation Plans. ANSI – IEEE Standard 1012 – 1986, IEEE Press, New York.

IEEE Transactions of Software Engineering, vol. 31, no. 11, pp. 1268–1287, (1988).

IEEE (1990), Standard Glossary of Software Engineering Terminology.ANSI – IEEE Standard 610.12 – 1990, IEEE Press, New York.

Edwards Deming, «Out of the Crisis», Massachusetts Institute of Technology, Cambridge University Press, (1988).

Tomas DeMarco, «Controlling Software Projects», Prentice Hall, New – York, (1982).

J. Edgemon, «Right Stuff: How to recognize it when selecting a Project Manager», Application Development Trends, vol. 2, no. 5, (1995).

V. A. Feigenbaum, «Total Quality Control», 3rd ed. New York, McGraw – Hill, (1983).

N. Fenton and S. Pfleeger, «Software Metrics A Rigorous & Practical Approach», Thomson Computer Press, (1997).

A. Fitzsimmons and T. Love, «A Review and Evaluation of Software Science», Computing Surveys, Volume 10, 1, (1978).

Y. Funami and M. Halstead, «A Software Physics Analysis of Akiyama's Debugging Data», Symposium on Computer Software Engineering, (1976).

David Garvin, «What Does Product Quality Really Mean?», Sloan Management Review, Fall, (1984).

J. McCall, P. Richards and G. Walters (1977), Factors in Software Quality. NTIS AD – A049 – 014, 015, 055 (3 τόμοι).

R. Pressman (1994), Software Engineering: a practitioner's approach. McGraw–Hill, England.

M.L. Shooman (1983), Software Engineering. McGraw – Hill, Tokyo.

I. Sommerville (1996), Software Engineering. Addison – Wesley, USA

T. Gilb, «Principles of Software Engineering Management», Addison Wesley, (1987).

M. H. Halstead, «Elements of Software Science», Elsevier Publications, N – Holland, (1975).

Brian Hambling, «Managing Software Quality», McGraw–Hill, (1996).

J. Harrington and D. Mathers, «ISO 9000 and Beyond», McGraw–Hill, (1997).

R. P. Higuera, «Team Risk Management», CrossTalk, U.S. Dept. of Defence,(1995).

R. C. Linger (1994), «Cleanroom process model». IEEE Software, 11(2), pp 50 – 58.

Darrel Ince, «ISO 9001 and Software Quality Assurance», McGraw–Hill,(1994).

Darrel Ince, «Software Quality Assurance: A Student Introduction»,McGraw–Hill, (1995).

ISO, «Information technology – Evaluation of software – Quality characteristics and guides for their use», International Standard, ISO/IEC 9126: (1991).

C. Jones, «Applied Software Measurement: Assuring Productivity and Quality», McGraw Hill, (1991).

Joseph Juran and F. Gryna, «Quality Planning and Analysis», 2nd ed. New York, McGraw – Hill, (1980).

C. Kaplan et al., «Secrets of Software Quality», McGraw Hill, (1995).

B. Kitchenham and S. Pfleeger, «Software Quality: The Elusive Target», IEEE Software, January, (1996).

R. Kraul and L. Streeter, «Coordination in Software Development», CASM, vol. 38, no. 3, (1995).

M. Mantei, «The Effect of Programming Team Structures on Programming Tasks», CACM, vol. 24, no. 3, (1981).

T. J. McCabe, «A Complexity Measure», IEEE Transactions in Software Engineering SE – 2(4), (1976).

J. A. McCall et al., «Factors in Software Quality, Vols I, II, III», US Rome Air Development Center Reports NTIS AD/A – 049 014, 015, 055, (1977).

P. W. Metzger, «Managing a Programming Project», Engelwood Cliffs, Prentice–Hall, (1973).

D. C. Montgomery, «Introduction to Statistical Quality Control», second edition, John Wiley & Sons, (1991).

M. Paulk et al., «Capability Maturity Model for Software (Version 1.1)», Carnegie Mellon University – Software Engineering Institute, Technical Report, CMU/SEI – 93 – TR – 024, (1993).

M. Paulk, «A Comparison of ISO 9001 and the Capability Maturity Model for Software», Carnegie Mellon University – Software Engineering Institute, Technical Report, CMU/SEI – 94 – TR – 012, (1994).

Roger S. Pressman, «Software Engineering: A Practitioner's Approach», Forth edition, European Adaptation, McGraw–Hill, (1997).

Martin L. Shooman, «Software Engineering: Design, Reliability and Management», McGraw–Hill, (1993).

Ian Sommerville, «Software Engineering», Third edition, Addison–Wesley, (1989).

Software Quality Management, «A Review of the Managing Software Quality in the 90's», Conference by Elliott Marley, Issue 18, Spring, pp. 24 – 28, (1993).

E. Weller, «Practical Applications of Statistical Process Control», IEEE Software, Vol. 17, No. 3, (2000).

DeMacro T., "Management Can Make Quality (Im)Possible", 6th European Conference on Software Quality, Vienna, 1999.

M. Xenos et al., «The Correlation Between Developer – oriented and User – oriented Software Quality Measurements (A Case Study)», 5th European Conference on Software Quality, EOQ – SC, (1996).

M. Xenos and D. Christodoulakis, «Measuring Perceived Software Quality», Information Technology Journal, Vol. 39, (1997).

Litvak B., Tyszberowicz S., Yehudai A. Behavioral Consistency Validation of UML diagrams, in Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03), ISBN:0-7695-1949-0/03 $17.00 © 2003 IEEE

Matula M. ,NetBeans Metadata Repository., March 2003, URL:http://mdr.netbeans.org/MDR-whitepaper.pdf

What is AndroMDA?,URL:http://galaxy.andromda.org/docs/whatisit.html

Engels G., Hücking G., Sauer S. , Wagner A.: UML Collaboration Diagrams and Their Transformation to Java. In France R, Rumbe B. (eds) : Proc. UML '99-Beyond the Standard, Fort Collins,CO, USA, LNCS 1723, Springer, 1999 pp 473-488

Mathupayas Thongmak , Pornsiri Muenchaisri, Design of Rules for Transforming UML Sequence Diagrams into Java code, in Proceedings of the Ninth Asia-Pacific Software Engineering Conference, p.485, December 04-06, 2002 .

Object Management Group, Inc., UML Superstructure Specification, version 2.1.2, November 2007.

Object Management Group, Inc.,Meta Object Facility (MOF) Specification version 1.4.1, May 2005

Object Management Group, Inc., Meta Object Facility (MOF) Core Specification version 2.0, January 2006.

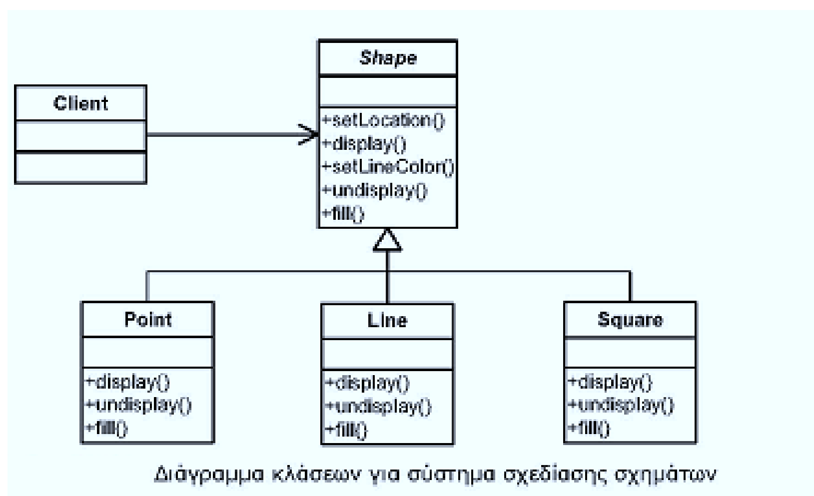
Atlas Group, LINA & INRIA, Atlas Transformation Language, ATL User Manual, version 0.7, February2006

IBM Corporation, IBM Model Transformation Framework User's Guide, version 1.0.2, 2004

10. Παράρτημα

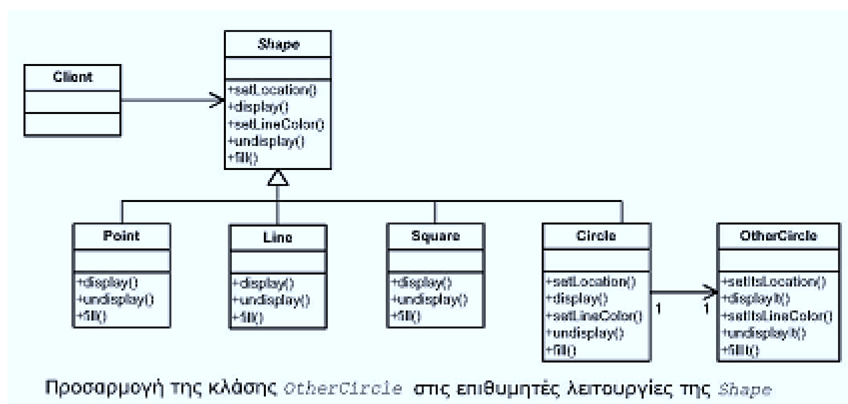
10.1 Προσαρμογέας – Παράδειγμα

Θεωρούμε μια εφαρμογή η οποία χειρίζεται διάφορα σχήματα (σημεία, γραμμές, τετράγωνα) με ενιαίο τρόπο, δηλαδή καλεί λειτουργίες επί των σχημάτων, αδιαφορώντας για το συγκεκριμένο είδος σχήματος. Τέτοιες λειτουργίες είναι η εμφάνιση του σχήματος, ο καθορισμός του χρώματος, η διαγραφή από την οθόνη, ο καθορισμός θέσης, το γέμισμα με χρώμα κλπ. Με άλλα λόγια, όλα τα σχήματα θα πρέπει να ενσωματωθούν σε μια αφηρημένη έννοια σχήματος που θα παρέχει όλες αυτές τις λειτουργίες. Η προφανής αντιμετώπιση του προβλήματος είναι με τη χρήση πολυμορφισμού: Μια αφηρημένη κλάση ορίζει όλες τις λειτουργίες της διασύνδεσης, ενώ η υλοποίηση των λειτουργιών είναι διαφορετική για κάθε μία από τις παράγωγες κλάσεις. Το πρόγραμμα πελάτης είναι σε θέση να χειρίζεται αντικείμενα της αφηρημένης κλάσης διατηρώντας ένα δείκτη προς αυτή, ενώ κατά την εκτέλεση του προγράμματος μεταβιβάζονται ως τιμές στον δείκτη οι διευθύνσεις συγκεκριμένων αντικειμένων των παράγωγων κλάσεων.Το πρόγραμμα πελάτης (αναθέτοντας διευθύνσεις υποκείμενων κλάσεων σε δείκτη της βασικής κλάσης) είναι σε θέση να καλέσει τις λειτουργίες των αντικειμένων – σχημάτων προς τα οποία "δείχνει" ο δείκτης κατά τη διάρκεια της εκτέλεσης (= πολυμορφισμός). Το διάγραμμα κλάσεων σε αυτή την περίπτωση παρουσιάζεται στο Σχήμα παρακάτω Ορισμένες βέβαια λειτουργίες, που έχουν κοινή υλοποίηση σε όλες τις παράγωγες κλάσεις, υλοποιούνται στην αφηρημένη βασική κλάση (όπως για παράδειγμα οι setLocation() και setLineColor()).



Διάγραμμα κλάσεων για σύστημα σχεδίασης σχημάτων

Σε περίπτωση που ζητηθεί η προσθήκη δυνατότητας σχεδίασης κύκλων από την εφαρμογή, θα πρέπει να προστεθεί μία νέα κλάση η οποία θα κληρονομεί από την αφηρημένη κλάση Shape υλοποιώντας την πολυμορφική συμπεριφορά των μεθόδων που δηλώνονται στη διασύνδεση. Στο σημείο αυτό, θα πρέπει να γραφεί ο κώδικας για τις μεθόδους display(), undisplay() και fill(). Επειδή η συγγραφή των μεθόδων αυτών είναι αρκετά περίπλοκη, αρχικά πραγματοποιείται αναζήτηση εναλλακτικής λύσης, υπό τη μορφή κάποιας κλάσης που ήδη υλοποιεί κύκλους και είναι διαθέσιμη. Έστω ότι εντοπίζεται μια τέτοια κλάση με το όνομα OtherCircle, η οποία διαθέτει τις ίδιες λειτουργίες με διαφορετικά ωστόσο ονόματα μεθόδων. Η κλάση OtherCircle δεν μπορεί να ενταχθεί απευθείας στην ιεραρχία των κλάσεων αφενός λόγω της ασυμβατότητας των μεθόδων (ονομάτων και ενδεχομένως παραμέτρων), καθώς και διότι στον ορισμό της δεν κληρονομεί από την κλάση Shape. Η λύση στο σημείο αυτό είναι η δημιουργία μιας νέας κλάσης Circle η οποία όντως κληρονομεί από την Shape (και κατά συνέπεια είναι συμβατή με τις λειτουργίες της διασύνδεσης), και η οποία περιέχει ένα αντικείμενο της κλάσης OtherCircle. Κατά αυτόν τον τρόπο, η κλάση Circle μπορεί να μεταβιβάζει τις αιτήσεις που φθάνουν σε αυτή, στο αντικείμενο OtherCircle που μπορεί να τις ικανοποιήσει. Η υπάρχουσα κλάση OtherCircle επομένως δεν τροποποιείται, αλλά προσαρμόζεται. Η λύση στο ανωτέρω πρόβλημα με χρήση του προτύπου "Προσαρμογέας" παρουσιάζεται στο διάγραμμα κλάσεων του σχήματος που ακολουθεί . Η εφαρμογή του προτύπου επιτρέπει τη συνέχιση της πολυμορφικής συμπεριφοράς των σχημάτων: το πρόγραμμα πελάτης συνεχίζει να χειρίζεται μόνο αντικείμενα τύπου Shape, αδιαφορώντας για τον διαφορετικό τρόπο υλοποίησης των λειτουργιών σε κάθε σχήμα.



Προσαρμογή της κλάσης `OtherCircle` στις επιθυμητές λειτουργίες της *Shape*

Η αφηρημένη κλάση Shape ορίζει την αφαίρεση ενός σχήματος ώστε να μπορεί να χρησιμοποιηθεί από προγράμματα-πελάτες. Δύο από τις μεθόδους (setLocation() και setLineColor()) είναι υλοποιημένες ενώ οι υπόλοιπες αφήνουν την υλοποίηση στις παράγωγες κλάσεις. (Η κλάση κληρονομεί την JComponent της Java έτσι ώστε να μπορεί να σχεδιαστεί σε ένα παράθυρο).

```
//Shape.java
import javax.swing.*;
import java.awt.*;
public abstract class Shape extends JComponent {
  public void setLocation(int x, int y) {
    xTopLeft = x;
    yTopLeft = y;
  }
  public void setLineColor(Color c) {
    lineColor = c;
  }
  public abstract void display();
  public abstract void undisplay();
  public abstract void fill(Color c);

  protected int xTopLeft = 0;
  protected int yTopLeft = 0;
  protected Color lineColor;
}
```

Κάθε πραγματικό σχήμα υλοποιείται ως παράγωγη κλάση της Shape παρέχοντας υλοποίηση στις αφηρημένες μεθόδους. Στη συνέχεια παρουσιάζεται ο κώδικας για τα τετράγωνα. Η μέθοδος paint() καλείται κατά την τοποθέτηση του αντικειμένου σε μια γραφική διασύνδεση.

```java
//Square.java

import javax.swing.*;
import java.awt.*;

public class Square extends Shape {

  public Square() {
    rect = new Rectangle(xTopLeft, yTopLeft, 100, 100);
  }

  public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setColor(lineColor);
    rect.setLocation(xTopLeft, yTopLeft);
    g2.draw(rect);
    if(fillColor != null) {
      g2.setColor(fillColor);
      g2.fillRect(xTopLeft, yTopLeft, 100, 100);
    }
  }

  public void fill(Color c) {
    fillColor = c;
  }

  public void display() {
    this.setVisible(true);
  }

  public void undisplay() {
    this.setVisible(false);
  }

  private Rectangle rect;
  private Color fillColor = null;
}
```

Ένα πρόγραμμα πελάτης προσομοιώνεται από τον ακόλουθο κώδικα. Στο αντικείμενο S1 της κλάσης Square μπορούν να κληθούν όλες οι μέθοδοι που δηλώνονται στη διασύνδεση της Shape.

```java
//Client.java

import javax.swing.*;
import java.awt.*;
import java.util.*;

public class Client extends JFrame {

  public void draw(Shape componentToDraw)
  {
    getContentPane().add(componentToDraw);
  }

  public static void main(String[] args) {
    Client frame = new Client();

    Square S1 = new Square();
    S1.setLineColor(Color.BLUE);
    S1.setLocation(50, 70);
    frame.draw(S1);
    S1.fill(Color.ORANGE);
    S1.undisplay();
    S1.display();

    frame.setSize(400, 400);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

}
```

Όπως αναφέρθηκε, υποθέτουμε ότι υπάρχει ήδη μια κλάση OtherCircle η οποία διαθέτει υλοποιημένες όλες τις αντίστοιχες μεθόδους, με διαφορετικές όμως υπογραφές, αποκλείοντας την απευθείας χρήση από έναν πελάτη ο οποίος "γνωρίζει" τη διασύνδεση της Shape. Για παράδειγμα, ένα πρόγραμμα πελάτης, μπορεί να σαρώνει μια λίστα από σχήματα και να τα σχεδιάζει, καλώντας τις ίδιες μεθόδους (αυτές που δηλώνονται στηShape) για όλα τα σχήματα. Ο κώδικας της OtherCircle φαίνεται παρακάτω:

```
//OtherCircle.java

import javax.swing.*;
import java.awt.*;

public class OtherCircle extends JComponent {

  public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D)g;
//Ο κατασκευαστής δημιουργεί στιγμιότυπο
//της προσαρμοζόμενης κλάσης
  public Circle() {
    adaptedCircle = new OtherCircle();
  }

  public void paint(Graphics g) {
    adaptedCircle.paint(g);
  }

//Επικάλυψη μεθόδων υπερκλάσης και αποστολή μηνυμάτων
  public void setLocation(int x, int y) {
    adaptedCircle.setItsLocation(y, x);
  }

  public void setLineColor(Color c) {
    adaptedCircle.setItsLineColor(c);
  }


//Υλοποίηση αφηρημένων μεθόδων

  public void fill(Color c) {
    adaptedCircle.fillIt(c);
  }

  public void display() {
    adaptedCircle.displayIt();
  }

  public void undisplay() {
    adaptedCircle.undisplayIt();
  }

  private OtherCircle adaptedCircle;
}
```

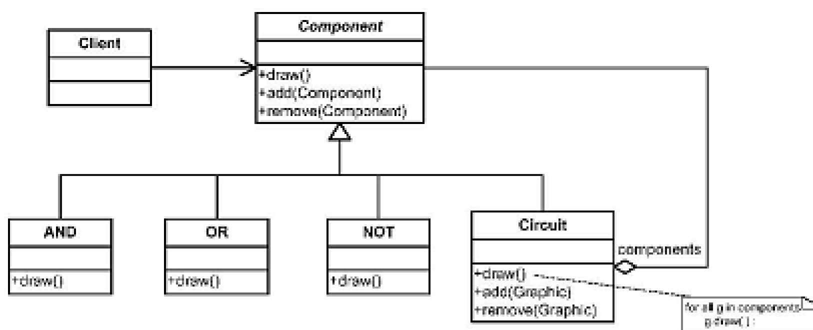
Κατά συνέπεια, το πρόγραμμα πελάτης χειρίζεται αντικείμενα Circle με τον ίδιο ακριβώς τρόπο, αγνοώντας την ύπαρξη των προσαρμοσμένων αντικειμένων OtherCircleπρος τα οποία αποστέλλονται τα πραγματικά μηνύματα:

```
import javax.swing.*;
import java.awt.*;
import java.util.*;

public class Client extends JFrame {

  public void draw(Shape componentToDraw)
  {
    getContentPane().add(componentToDraw);
  }

 public static void main(String[] args) {
    Client frame = new Client();

    Circle C1 = new Circle();
    C1.setLineColor(Color.RED);
    C1.setLocation(200, 200);
    frame.draw(C1);
    C1.fill(Color.RED);
    S1.undisplay();
    S1.display();

    frame.setSize(400, 400);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

}
```

10.2 Σύνθετο – Παράδειγμα

Οι εφαρμογές που σχεδιάζουν πολύπλοκα ψηφιακά κυκλώματα VLSI (όπως για παράδειγμα το κύκλωμα ενός επεξεργαστή), αντιμετωπίζουν οποιαδήποτε οντότητα ως αντικείμενο. Κάθε αντικείμενο έχει ορισμένες λειτουργίες, μεταξύ αυτών η σχεδίαση του. Ορισμένα από τα αντικείμενα είναι πρωταρχικά και δεν αναλύονται περαιτέρω (π.χ. λογικές πύλες AND, OR, NOT), ενώ άλλα αντικείμενα είναι σύνθετα και αποτελούνται από πρωταρχικές πύλες (π.χ. ένας πλήρης αθροιστής – FullAdder). Ο χρήστης είναι σε θέση να δημιουργήσει οποιαδήποτε σύνθετη οντότητα και να την προσθέσει στην εφαρμογή. Η σχεδίαση ενός σύνθετου αντικειμένου ουσιαστικά συνίσταται στη σχεδίαση των επίμερους τμημάτων του. Για το λόγο αυτό, είναι επιθυμητή η ενιαία αντιμετώπιση όλων των αντικειμένων. Το πρότυπο σχεδίασης "Σύνθετο", επιτρέπει τον αναδρομικό ορισμό περιεκτικότητας, ώστε οι πελάτες να μην αντιλαμβάνονται τη διαφορά μεταξύ πρωταρχικών και σύνθετων αντικειμένων. Το σημείο κλειδί στο πρότυπο σχεδίασης "Σύνθετο" είναι η ύπαρξη μιας αφηρημένης κλάσης που αναπαριστά τόσο πρωταρχικές κλάσεις όσο και περικλείουσες κλάσεις. Για τη συγκεκριμένη εφαρμογή, η αφηρημένη κλάση είναι η κλάση

Εξάρτημα . Η κλάση Εξάρτημα (Component) δηλώνει λειτουργίες όπως η σχεδίαση που είναι διαφορετική για κάθε αντικείμενο, καθώς και λειτουργίες που είναι κοινές σε όλα τα σύνθετα αντικείμενα, όπως οι λειτουργίες προσθήκης και αφαίρεσης αντικειμένων. Οι λειτουργίες αυτές, δεν υλοποιούνται στις πρωταρχικές κλάσεις, καθώς οι πρωταρχικές πύλες δεν περιέχουν αντικείμενα. Θα πρέπει να σημειωθεί, ότι η κλάση Εξάρτημα που εισάγεται από το πρότυπο, δεν έχει αντιστοιχία με οποιαδήποτε φυσική έννοια του πραγματικού κόσμου. Η κλάση Κύκλωμα (Circuit), ορίζει μια συσσωμάτωση αντικειμένων τύπου Εξάρτημα. Η λειτουργία της σχεδίασης υλοποιείται καλώντας τη λειτουργία της σχεδίασης όλων των αντικειμένων (πρωταρχικών ή σύνθετων) που εμπεριέχονται. Η κλάση Κύκλωμα, συμμορφώνεται με τη διασύνδεση που δηλώνεται από την κλάση Εξάρτημα και για το λόγο αυτό, αντικείμενα τύπου Κύκλωμα είναι δυνατόν να περιλαμβάνουν αναδρομικά άλλα αντικείμενα τύπου Κύκλωμα.



Εφαρμογή του προτύπου "Σύνθετο" σε πρόγραμμα σχεδίασης κυκλωμάτων

Θέτοντας τις λειτουργίες χειρισμού των αντικειμένων που εμπεριέχονται σε μια περικλείουσα κλάση (add(), remove()) στην κλάση Component, όλα τα αντικείμενα αντιμε-τωπίζονται με τον ίδιο ακριβώς τρόπο, εξασφαλίζοντας ομοιομορφία. Ωστόσο δημιουργείται το εξής πρόβλημα: Για τις κλάσεις φύλλα (όπως οι AND, OR, NOT) είναι δυνατόν να κληθούν οι λειτουργίες add(), remove() οι οποίες δεν έχουν νόημα και για τον λόγο αυτό ο σχεδιαστής θα πρέπει να παράσχει μια εκφυλισμένη υλοποίηση. Εναλλακτικά, οι λειτουργίες αυτές είναι δυνατόν να δηλωθούν στην κλάση Circuit, ώστε οποιαδήποτε απόπειρα να προστεθούν ή να διαγραφούν αντικείμενα σε αντικείμενα τύπου AND, OR ή NOT να ανιχνευθεί κατά τη μεταγλώττιση, εξασφαλίζοντας ασφάλεια.Ωστόσο, στην περίπτωση αυτή, οι πρωταρχικές κλάσεις και οι σύνθετες κλάσεις έχουνδιαφορετική διασύνδεση.

Η αφηρημένη κλάση Component ορίζει τη διασύνδεση κάθε οντότητας που μπορεί νασχεδιαστεί, ώστε να μπορεί να χρησιμοποιηθεί από προγράμματα-πελάτες:

```
//C++

class Component {
public:
  Component(string text);
  virtual void draw() = 0;

  virtual void add(Component*) {};
  virtual void remove(Component*) {};
protected:
  string identifier;
};

Component::Component(string text)
{
  identifier = text;
}
```

Μια πρωταρχική κλάση, όπως μια κλάση πύλης AND, δηλώνεται ως παράγωγη της κλάσης Component:

```
class AND : public Component {
public:
  AND(string text);
  virtual void draw();
};
```

Η κλάση Circuit είναι μια περικλείουσα κλάση η οποία είναι δυνατόν να περιέχει αντικείμενα είτε πρωταρχικών κλάσεων (π.χ. ένα κύκλωμα που αποτελείται από πύλες), είτεάλλων σύνθετων κλάσεων (π.χ. ένα κύκλωμα που αποτελείται από άλλα κυκλώματα), είτεαντικείμενα και των δύο κατηγοριών (ένα κύκλωμα που αποτελείται από πύλες και άλλακυκλώματα). Για το λόγο αυτό, μια από τις ιδιότητες υλοποιεί έναν περιέχοντα(container), όπως ένα διάνυσμα (vector), ουρά (queue), διπλά συνδεδεμένη λίστα (list),στοίβα (stack) κ.ο.κ, που περιλαμβάνονται στην πρότυπη βιβλιοθήκη της C++ (STL):

```
class Circuit : public Component {
public:
  Circuit(string text);
  virtual void draw();

  virtual void add(Component*);
  virtual void remove(Component*);
private:
  vector<Component*> components;  //διάνυσμα περιεχόμενων
                                  //συστατικών
};

Circuit::Circuit(string text) : Component(text) {};
```

10.3 Μοναδιαίο – Παράδειγμα

Έστω ότι επιχειρείται η αντικειμενοστρεφής μοντελοποίηση μιας κεντρικής μονάδας επεξεργασίας (ΚΜΕ) ενός υπολογιστή. Παρόλο που είναι δυνατόν να υπάρχουν πολλοί καταχωρητές γενικής χρήσης, πρέπει να υπάρχει (σε συμβατική αρχιτεκτονική), μόνο ένας καταχωρητής (συσσωρευτής - Accumulator) στον οποίο καταλήγουν τα αποτελέσματα των αριθμητικών και λογικών πράξεων. Η εφαρμογή πρέπει να εξασφαλίσει ότι δεν θα δημιουργηθούν πέραν του ενός τέτοιοι συσσωρευτές και ότι οι λειτουργίες του καταχωρητή θα είναι καθολικά προσπελάσιμες. Οι ανωτέρω απαιτήσεις ικανοποιούνται εφαρμόζοντας το πρότυπο «Μοναδιαίο» και ορίζοντας στην κλάση Accumulator μια στατική λειτουργία getInstance() η οποία δημιουργεί ένα αντικείμενο μόνο όταν αυτό δεν υφίσταται. Η λειτουργία είναι δηλωμένη στατική, έτσι ώστε να μπορεί να κληθεί και πριν από την κατασκευή του μοναδικού αντικειμένου. Η κλάση περιλαμβάνει ένα στατικό μέλος instance, δείκτη προς το μοναδικοαντικείμενο (όσο αυτό δεν υφίσταται, η τιμή του δείκτη είναι NULL). Η λειτουργίαgetInstance() σε περίπτωση που η τιμή του δείκτηinstance είναι NULL δημιουργεί ένα αντικείμενο της κλάσης. Σε περίπτωση που ο δείκτης έχει ήδη μια τιμή, η λειτουργία απλώς επιστρέφει την τιμή του.

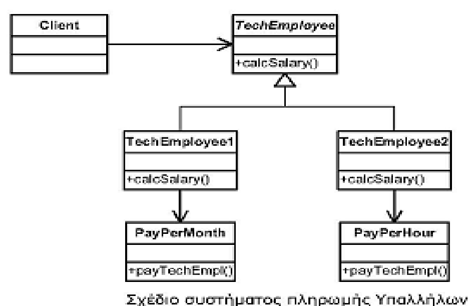
```
class Accumulator {
public:
  static Accumulator* getInstance();
protected:
  Accumulator();
private:
  static Accumulator* instance;
};

Accumulator* Accumulator::instance = NULL;

Accumulator::Accumulator() { }

Accumulator* Accumulator::getInstance()
{
  if(instance == NULL)
    instance = new Accumulator;

  return instance;
}
```

Τα προγράμματα πελάτες μπορούν να προσπελάσουν το συσσωρευτή μόνο μέσω της μεθόδου getInstance() καθώς ο κατασκευαστής είναι δηλωμένος ως προστατευμένοςκαι κατά συνέπεια η δημιουργία ενός αντικειμένου δεν μπορεί να πραγματοποιηθεί παρακάμπτοντας τη μέθοδο getInstance(). Οποιαδήποτε

προσπάθεια κλήσης του κατασκευαστή θα οδηγήσει σε σφάλμα κατά τη μεταγλώττιση. Η τιμή που επιστρέφει η μέθοδος, δεν δημιουργείται μέχρι την πρώτη κλήση της (lazyinitialization). Αν επομένως ένα μοναδιαίο αντικείμενο δεν χρησιμοποιηθεί, δεν δημιουργείται.

10.4 Γέφυρα – Παράδειγμα

Θεωρούμε μια εφαρμογή η οποία χειρίζεται τις πληρωμές τεχνικών υπαλλήλων (TechEmployees).Η πληρωμή των υπαλλήλων είναι δυνατόν να πραγματοποιείται είτε με βάση το μηνιαίο μισθό, είτε με βάση τον αριθμό ωρών εργασίας ανά μήνα και την αντίστοιχη αμοιβή ανά ώρα. Ο υπολογισμός του μισθού, κρατήσεων κλπ. πραγματοποιείται από δύο ανεξάρτητα προγράμματα, που αντιστοιχούν στις κλάσεις PayPerMonth και PayPerHour. Η εφαρμογή θα πρέπει να είναι σε θέση να λειτουργήσει με υπάλληλο οποιασδήποτε κατηγορίας πληρωμής, αλλά δεν οφείλει να γνωρίζει εκ των προτέρων τον τρόπο πληρωμής κάθε υπαλλήλου. Με άλλα λόγια, ο τρόπος πληρωμής (που υλοποιείται προγραμματιστικά ως μια σύνδεση με μία από τις δύο κλάσεις πληρωμών), μπορεί να τροποποιηθεί κατά τη διάρκεια της εκτέλεσης του προγράμματος. Οι υπάλληλοι θα σχετίζονται με συγκεκριμένο τρόπο πληρωμής κατά την υλοποίηση των αντικειμένων τους, και κατά συνέπεια είναι λογικό η εφαρμογή να διατηρεί έναν δείκτη προς μια αφηρημένη κλάση Τεχνικός Υπάλληλος (TechEmployee) από την οποία θα κληρονομούν οι δύο συγκεκριμένες κατηγορίες με βάση τον τρόπο πληρωμής. Η κάθε κατηγορία διατηρεί δείκτη προς την αντίστοιχη κλάση υπολογισμού του μισθού. Το διάγραμμα κλάσεων για την περίπτωση αυτή παρουσιάζεται στο σχήμα.



Σχέδιο συστήματος πληρωμής Υπαλλήλων

Η αφηρημένη κλάση Employee ορίζει την αφαίρεση ενός Υπαλλήλου ώστε να μπορεί να χρησιμοποιηθεί από προγράμματα-πελάτες (C++, η υλοποίηση των μεθόδων συμπεριλαμβάνεται στη δήλωση της κλάσης για λόγους συντομίας.

```
class Employee {
public:
 Employee(Pay* payImpl) { payImplementation = payImpl; }

 virtual void calcSalary()=0;

 void setMonthlySalary(double amount) { monthlySalary = amount; }
 double getMonthlySalary() { return monthlySalary; }
 void setHoursWorked(int hours) { hoursWorked = hours; }
 void setHourlySalary(double amount) { hourlySalary = amount; }
 int getHoursWorked() {return hoursWorked; }
 double getHourlySalary() { return hourlySalary; }
 Pay* getPayImplementation() { return payImplementation; }
 void setPayImplementation(Pay* impl) { payImplementation =
                                         impl;}

private:
 Pay* payImplementation;

 int hoursWorked;
 float monthlySalary;
 float hourlySalary;
};
```

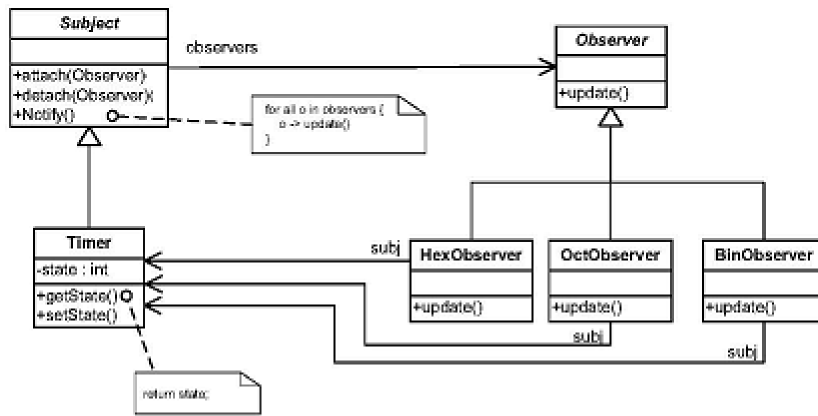
Η κλάση Employee διατηρεί έναν δείκτη προς την κλάση Pay, υλοποιώντας τη γέφυραμεταξύ αφαίρεσης και υλοποίησης. Οι ιδιότητεςhoursWorked, monthlySalary και hourlySalary, κληρονομούνται από όλες τις παράγωγες κλάσεις που αφορούν σε ειδικότερες κατηγορίες υπαλλήλων, δεν λαμβάνουν όμως τιμές σε όλες τις περιπτώσεις. Μια παράγωγος κλάση είναι για παράδειγμα η κατηγορία των Τεχνικών Υπαλλήλων.

```
class TechEmployee : public Employee {
public:
  TechEmployee(Pay* payImpl);
  virtual void calcSalary();
};

TechEmployee::TechEmployee(Pay* payImpl) : Employee(payImpl) {}

void TechEmployee::calcSalary()
{
  getPayImplementation()->payTechEmpl(this);
}
```

10.5 Παρατηρητής – Παράδειγμα

Θεωρούμε μια κλάση Timer η οποία υλοποιεί ένα χρονόμετρο (ο φυσικός μηχανισμόςτης μέτρησης δεν μας ενδιαφέρει, θεωρούμε ότι η κατάσταση του

χρονομέτρου τίθεται από εξωτερική πηγή – στο συγκεκριμένο παράδειγμα από το χρήστη). Την τιμή του χρονομέτρου εμφανίζουν στην οθόνη, υπό διαφορετικές μορφές (στο δεκαεξαδικό, οκταδικόκαι δυαδικό σύστημα) διάφοροι παρατηρητές. Σκοπός είναι η αυτόματη ενημέρωση όλων των "οθονών" οποτεδήποτε αλλάζει η κατάσταση του χρονομέτρου. Το πρώτο ζητούμενο είναι όλοι οι παρατηρητές να έχουν την ίδια διασύνδεση, ώστε να μην απαιτείται η τροποποίηση του υποκειμένου (χρονομέτρου) κάθε φορά που προστίθεται ένας νέος παρατηρητής. Αυτό επιτυγχάνεται επιβάλλοντας σε όλες τις κλάσεις παρατηρητών να κληρονομούν μια αφηρημένη βασική κλάση ή μια διασύνδεση (αφηρημένη κλάση Observer). Επειδή είναι επιθυμητό η ευθύνη της παρακολούθησης του υποκειμένου (χρονομέτρου) να ανήκει στους παρατηρητές (οθόνες), κάθε αντικείμενο Παρατηρητής θα πρέπει να μπορεί να καταχωρεί τον εαυτό του σε μια λίστα παρατηρητών που διαθέτει το χρονόμετρο. Καθώς όλοι οι παρατηρητές είναι ιδίου τύπου (ως απόγονοι της κλάσης Observer) αρκεί η προσθήκη μεθόδων προσθήκης (attach()) και διαγραφής (detach()) παρατηρητών σε μια δομή δεδομένων του υποκειμένου. Πλέον η ανακοίνωση ενός συμβάντος (π.χ. αλλαγή τιμής του χρονομέτρου) στους παρατηρητές γίνεται με τη σάρωση της λίστας των παρατηρητών και την κλήση μιας κοινής μεθόδου ενημέρωσης (update()). Η μέθοδος update() της κάθε οθόνης υλοποιεί τονκώδικα που χειρίζεται το κάθε συμβάν (όπως για παράδειγμα την εμφάνιση της τιμής του χρονομέτρου στο κατάλληλο σύστημα αρίθμησης). Συνήθως η ανακοίνωση του συμβάντος στους παρατηρητές δεν αρκεί. Ο παρατηρητής μπορεί να απαιτεί τη γνώση περισσότερης πληροφορίας για το συμβάν. Η κάθε οθόνη για παράδειγμα δεν αρκεί να γνωρίζει μόνο ότι άλλαξε η τιμή αλλά θα πρέπει να μπορεί να πληροφορηθεί την ίδια την τιμή του χρονομέτρου. Για το λόγο αυτό, παρέχονται μέθοδοι στο υποκείμενο που "εξάγουν" στοιχεία πληροφορίας τα οποία μπορούν να χρησιμοποιούν οι παρατηρητές (π.χ. μια μέθοδος getState() η οποία επιστρέφει την τιμή του χρονομέτρου). Το διάγραμμα κλάσεων για την υλοποίηση του προτύπου Παρατηρητής στο ανωτέρω παράδειγμα φαίνεται στο σχήμα.



Υλοποίηση προτύπου "Παρατηρητής"
στο παράδειγμα του Χρονομέτρου

Η υλοποίηση του συστήματος σε Java παρουσιάζεται στη συνέχεια. Αξίζει να σημειωθείότι η σχεδίαση συμμορφώνεται με την αρχή της αντιστροφής των εξαρτήσεων και για το λόγο αυτό η κλάση Subject αλλά και η κλάση Observer δηλώνεται ως αφηρημένη βασική κλάση. Κατά συνέπεια, η σύζευξη μεταξύ παρατηρητών και υποκειμένων είναι χαλαρή και κάθε έννοια μπορεί να επαναχρησιμοποιηθεί χωρίς να εξαρτάται από την άλλη. Η κλάση Subject (και κάθε απόγονός της) δεν χρειάζεται να γνωρίζει ποιο συγκεκριμένο είδος παρατηρητή πρέπει να ενημερώνεται κάθε φορά.

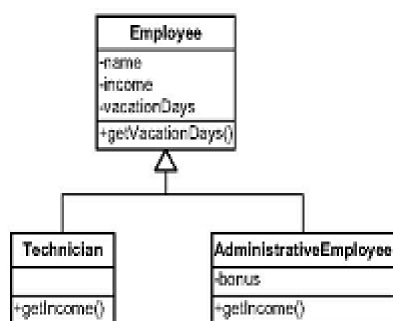
```
import javax.swing.*;
import java.util.*;
class Subject {
   private ArrayList observers = new ArrayList();

   public void attach( Observer o )
   {
     observers.add( o );       //καταχώρηση παρατηρητή
   }

   public void detach( Observer o)
   {
     observers.remove( o );    //διαγραφή παρατηρητή
   }

   public void Notify()
   {
     for (int i=0; i < observers.size(); i++)
       ((Observer)observers.get(i)).update();
                //ανακοίνωση αλλαγών στους παρατηρητές
   }
}

 class Timer extends Subject {

   private int state;

   public int getState() { return state; }
   public void setState( int in ) {
     state = in;
     Notify();      //κάθε φορά που αλλάζει η κατάσταση
   }                //καλείται η Notify()
}

 abstract class Observer {
   public abstract void update();
}

 class HexObserver extends Observer {

   private Timer subj;

   public HexObserver( Timer s ) {
     subj = s;
     subj.attach( this );
                //Οι παρατηρητές καταχωρούν τον εαυτό τους
   }
   public void update() {
   System.out.print( " " + Integer.toHexString(subj.getState()) );
   }            //Οι παρατηρητές αντλούν πληροφορία
}
```

```
class OctObserver extends Observer {

  private Timer subj;

  public OctObserver( Timer s ) {
    subj = s;
    subj.attach( this );
               //Οι παρατηρητές καταχωρούν τον εαυτό τους
  }
  public void update() {
  System.out.print(" " + Integer.toOctalString(subj.getState()));
  }            //Οι παρατηρητές αντλούν πληροφορία
}

class BinObserver extends Observer {

  private Timer subj;

  public BinObserver( Timer s ) {
    subj = s;
    subj.attach( this );
               //Οι παρατηρητές καταχωρούν τον εαυτό τους
  }
  public void update() {
  System.out.print(" " +Integer.toBinaryString(subj.getState()));
  }            //Οι παρατηρητές αντλούν πληροφορία
}

public class ObserverDemo {
  public static void main( String[] args ) {
    Timer timerSubject = new Timer();

    new HexObserver( timerSubject );
    new OctObserver( timerSubject );
    BinObserver binObs = new BinObserver( timerSubject );
    for(int i=0; i<2; i++) {
      System.out.print( "\nEnter a number: " );
      String text = JOptionPane.showInputDialog(null,
                       "Current time? (integer)");
      int n = Integer.parseInt(text);
      timerSubject.setState(n);  //αλλαγή κατάστασης υποκειμένου
    }
//Διαγραφή Παρατηρητή
    timerSubject.detach(binObs);
    System.out.println();
    System.out.println("BinObserver Detached from Timer");
```

```
    for(int i=0; i<2; i++) {
      System.out.print( "\nEnter a number: " );
      String text = JOptionPane.showInputDialog(null, "Current
                    time? (integer)");
      int n = Integer.parseInt(text);
      timerSubject.setState(n); //αλλαγή κατάστασης υποκειμένου
    }
  }
}
```

10.6 Επισκέπτης – Παράδειγμα

Θεωρούμε μια εφαρμογή η οποία διαχειρίζεται το εισόδημα (Income) και τις μέρες αδείας (VacationDays) των υπαλλήλων μιας εταιρείας. Στο σύστημα υπάρχουν αρχικά δύο κατηγορίες υπαλλήλων, Τεχνικοί Υπάλληλοι (Technicians) και Διοικητικοί Υπάλληλοι (AdministrativeEmployees). Η αναπαράσταση του συστήματος ως διάγραμμα κλάσεων παρουσιάζεται στο σχήμα.



Σύστημα Υπαλλήλων Εταιρείας

Η υλοποίηση του προτύπου "Επισκέπτης" για το σύστημα της προηγούμενης ενότητας,στη γλώσσα προγραμματισμού Java παρουσιάζεται στη συνέχεια. Η διαφοροποίηση των δύο κατηγοριών υπαλλήλων συνίσταται στο ότι οι Διοικητικοί Υπάλληλοι λαμβάνουν ένα επιπρόσθετο bonus. Οι δύο παράγωγες κλάσεις της ιεραρχίας επισκέπτη υλοποιούν τις εξής λειτουργίες: Η κλάση "επισκέπτης" IncomeVisitor αυξάνει το εισόδημα των Τεχνικών κατά 10% ενώ στους Διοικητικούς Υπαλλήλους αυξάνει μόνο το bonus κατά 20%. Η κλάση "επισκέπτης" VacationDaysVisitor αυξάνει την άδεια των Τεχνικών κατά 3 ημέρες ενώ των Διοικητικών υπαλλήλων κατά 2 ημέρες. Στη συνάρτηση main της κλάσης VisitorTest εμφανίζονται οι ημέρες αδείας και οι συνολικές αποδοχές για έναν υπάλληλο της κάθε κατηγορίας πριν και μετά τη χρήση των κλάσεων "Επισκέπτη".

```
// κλάση Employee
abstract class Employee {

  public Employee(String text, double amount, int days)
  {
    name = text;
    income = amount;
    vacationDays = days;
  }

  public void setName(String text) { name = text; }
  public void setIncome(double amount) { income = amount; }
  public void setVacationDays(int days) { vacationDays = days; }
  public String getName() { return name; }
  public int getVacationDays() { return vacationDays; }

  public abstract void accept(Visitor visitor);

  private String name;
  protected double income;
  private int vacationDays;
}

// κλάση Technician
class Technician extends Employee {

  public Technician(String text, double amount, int days)
  {
    super(text, amount, days);
  }

  public double getIncome() { return income; }

  public void accept(Visitor visitor) {visitor.visit(this);}
}
```

```
// κλάση AdministrativeEmployee
class AdministrativeEmployee extends Employee {

  public AdministrativeEmployee(String text, double amount,
                                int days, int extra)
  {
    super(text, amount, days);
    bonus = extra;
  }

  public void setBonus(double amount) { bonus = amount; }
  public double getBonus() { return bonus; }

  public double getIncome() { return income; }

  public void accept(Visitor visitor) {visitor.visit(this);}

  private double bonus;
}
//διασύνδεση Visitor

interface Visitor {

  public void visit(Technician employee);
  public void visit(AdministrativeEmployee employee);
}
```

```
//κλάση "Επισκέπτης" IncomeVisitor
class IncomeVisitor implements Visitor {

  public void visit(Technician employee)
  {
    //Οι Τεχνικοί λαμβάνουν αύξηση 10% του μισθού
    employee.setIncome(employee.getIncome() * 1.10);
  }

  public void visit(AdministrativeEmployee employee)
  {
    //Οι Διοικητικοί λαμβάνουν αύξηση 20% του bonus
    employee.setBonus(employee.getBonus() * 1.20);
  }
}
```

```
//κλάση "Επισκέπτης" VacationDaysVisitor
class VacationDaysVisitor implements Visitor {

  public void visit(Technician employee)
  {
    // Οι Τεχνικοί λαμβάνουν αύξηση άδειας κατά 3 ημέρες
    employee.setVacationDays(employee.getVacationDays() + 3);
  }

  public void visit(AdministrativeEmployee employee)
  {
    // Οι Διοικητικοί λαμβάνουν αύξηση άδειας κατά 2 ημέρες
    employee.setVacationDays(employee.getVacationDays() + 2);
  }
}

public class VisitorTest {

  public static void main(String[] args) {

    Technician E1 = new Technician("Fred", 2000, 23);

    AdministrativeEmployee E2 =
        new AdministrativeEmployee("John", 3000, 25, 200);

    System.out.println(E1.getName() + ", Total Income: " +
                       E1.getIncome() + ", Vacation Days: " +
                       E1.getVacationDays() );
```

```
    System.out.println(E2.getName() + ", Total Income: " +
                       (E2.getIncome()+E2.getBonus()) +
                       ", Vacation Days: " +
                       E2.getVacationDays() );

    IncomeVisitor v1 = new IncomeVisitor();  //Βήμα 1
    VacationDaysVisitor v2 = new VacationDaysVisitor();

    E1.accept(v1);                           //Βήμα 2
    E2.accept(v1);

    E1.accept(v2);
    E2.accept(v2);

    System.out.println("After visits have been made....");

    System.out.println(E1.getName() + ", Total Income: " +
                       E1.getIncome() + ", Vacation Days: "+
                       E1.getVacationDays() );

    System.out.println(E2.getName() + ", Total Income: " +
                       (E2.getIncome()+E2.getBonus()) +
                       ", Vacation Days: " +
                       E2.getVacationDays() );
  }
}
```

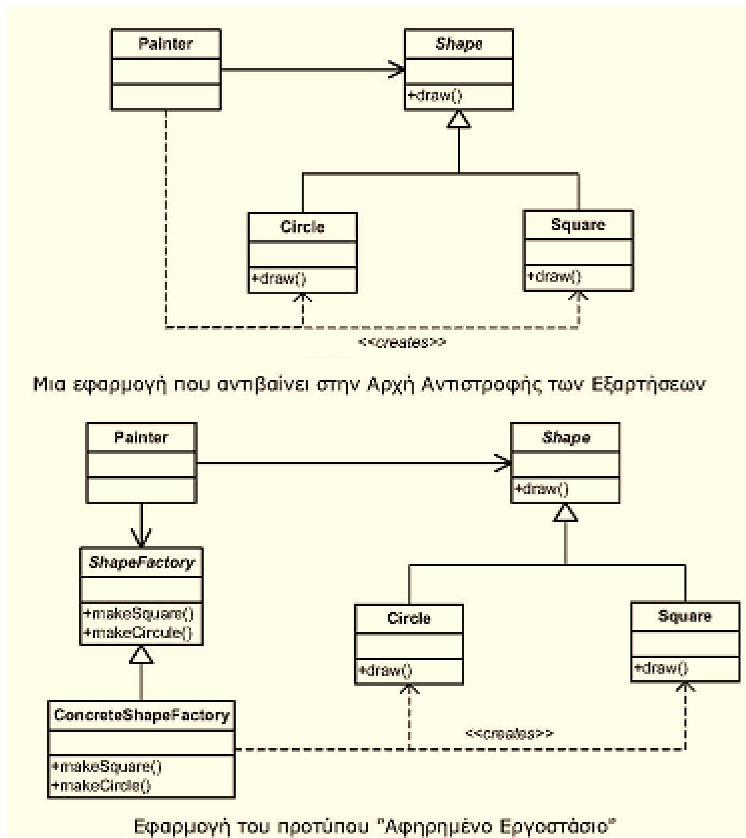
10.7 Αφηρημένο εργοστάσιο – Παράδειγμα

Μια από τις βασικές αρχές του αντικειμενοστρεφούςπρογραμματισμού (Αρχή Αντιστροφής των Εξαρτήσεων) υπαγορεύει ότι θα πρέπει να προτιμώνται οι

εξαρτήσεις από αφηρημένες κλάσεις και να αποφεύγονται εξαρτήσεις από συγκεκριμένες κλάσεις. Για παράδειγμα, αν θεωρήσουμε ότι σε μια εφαρμογή υπάρχει το ακόλουθο τμήμα κώδικα για τη δημιουργία ενός τετραγώνου:

```
Square* s = new Square(xUpperLeft, yUpperLeft, width);
```

Η ανωτέρω αρχή παραβιάζεται, καθώς η Square είναι μια συγκεκριμένη κλάση. Οποιαδήποτε αλλαγή στην κλάση Square, θα επιφέρει αλλαγές και στις κλάσεις που εξαρτώνται από αυτήν (αν αλλάξει η υπογραφή του κατασκευαστή θα πρέπει να τροποποιηθεί η κλήση του, αν αλλάξει οτιδήποτε άλλο θα πρέπει επίσης να γίνει επαναμεταγλώττιση). Υπό την έννοια αυτή, ο τελεστής new θα πρέπει να αποφεύγεται. Ωστόσο, αν η κλάση από την οποία εξαρτάται μια άλλη, δεν έχει μεγάλη πιθανότητα να τροποποιηθεί (όπως για παράδειγμα η κλάση String) δεν υπάρχει λόγος ανησυχίας. Το πρότυπο σχεδίασης "Αφηρημένο Εργοστάσιο" επιτρέπει ακριβώς αυτό: τη δημιουργία στιγμιοτύπων συγκεκριμένων αντικειμένων επιβάλλοντας εξάρτηση μόνο από αφηρημένες κλάσεις. Θεωρούμε μια εφαρμογή όπου μια κλάση η οποία ονομάζεται Painter δημιουργεί σχήματα (κύκλους, τετράγωνα). Για την κλήση των μεθόδων (π.χ. τη μέθοδο σχεδίασης draw()) εξαρτάται από τη διασύνδεση Shape (ορθά) αλλά για την κατασκευή αντικειμένων τύπου Square και Circle εξαρτάται από τις συγκεκριμένες κλάσεις. Η κατάσταση αυτή που απεικονίζεται στο διάγραμμα κλάσεων του σχήματος που ακολουθεί είναι προβληματική καθώς αντιβαίνει στην αρχή Αντιστροφής των Εξαρτήσεων που αναφέρθηκε. Το πρόβλημα αυτό επιλύεται με την εφαρμογή του προτύπου "Αφηρημένο Εργοστάσιο"όπως παρουσιάζεται στο 2ο σχήμα. Η διασύνδεση της αφηρημένης κλάσης Shape- Factory έχει δύο μεθόδους: makeSquare() και makeCircle(). Η πρώτη μέθοδος επιστρέφει ένα αντικείμενο τύπου Square και η δεύτερη ένα αντικείμενο τύπου Circle. Ωστόσο, και οι δύο μέθοδοι είναι δηλωμένες στη διασύνδεση ShapeFactory ότι επιστρέφουν αντικείμενο τύπου Shape και κατά συνέπεια ο κώδικας της εφαρμογής Painter δεν χρειάζεται να γνωρίζει τις συγκεκριμένες κλάσεις που χρησιμοποιεί. Η υλοποίηση των μεθόδων της αφηρημένης κλάσης ShapeFactory πραγματοποιείται στην παράγωγη κλάση ConceteShapeFactory η οποία και "γνωρίζει" τις κλάσεις και τους κατασκευαστές των συγκεκριμένων σχημάτων που πρέπει να δημιουργηθούν.



Μια εφαρμογή που αντιβαίνει στην Αρχή Αντιστροφής των Εξαρτήσεων

Εφαρμογή του προτύπου "Αφηρημένο Εργοστάσιο"

Ο κώδικας των κλάσεων Shape και Circle είναι (C++):

```
class Shape {
public:
  virtual void draw() = 0;
};

class Circle : public Shape {
public:
  void draw();
};

void Circle::draw()
{
  cout << "A Circle is drawn on the screen" << endl;
}
```

Ο κώδικας για την αφηρημένη κλάσηShapeFactory καθώς και για την συγκεκριμένη κλάση ConcreteShapeFactory δίνεται παρακάτω.

```
class ShapeFactory {
public:
  virtual Circle* makeCircle() = 0;
  virtual Square* makeSquare() = 0;
};

class ConcreteShapeFactory : public ShapeFactory {
public:
  Circle* makeCircle();
  Square* makeSquare();
};

Circle* ConcreteShapeFactory::makeCircle()
{
  return new Circle();
}

Square* ConcreteShapeFactory::makeSquare()
{
  return new Square();
}
```

Στην όλη εφαρμογή απομένει η δημιουργία του "εργοστασίου" κλάσεων. Η συγκεκριμένη υλοποίηση της ShapeFactory ανατίθεται συνήθως στη συνάρτηση main ή σε μιασυνάρτηση αρχικοποίησης που καλείται από τη main:

```
int main()                //έχει το ρόλο του προγράμματος-πελάτη
{
  ConcreteShapeFactory factory;
  Circle* aShape = factory.makeCircle();

  aShape -> draw();

  return (0);
}
```

10.8 Στρατηγική – Παράδειγμα

Θεωρούμε μια κλάση Find η οποία παρέχει μεθόδους για την εύρεση του ελαχίστου, μεγίστου και διαμέσου ενός διανύσματος ακεραίων, υλοποιώντας έναν αλγόριθμο ταξινόμησης φυσαλίδας (Bubblesort). Ο αλγόριθμος είναι κωδικοποιημένος μέσα στην κλάση και κατά συνέπεια οποιοδήποτε πρόγραμμα πελάτης της Find είναι υποχρεωμένο να χρησιμοποιήσει τον αλγόριθμο φυσαλίδας ακόμα και αν είναι διαθέσιμος κάποιος άλλος αλγόριθμος. Η δυνατότητα επαναχρησιμοποίησης της κλάσης Find θα ήταν σαφώς μεγαλύτερη αν ήταν δυνατή η επιλογή του αλγορίθμου ταξινόμησης που χρησιμοποιείται από την κλάση Find, χωρίς βέβαια να πρέπει να τροποποιηθεί ο κώδικας της κλάσης.

Find
+readVector() +getMin() +getMax() +getMed() -sort()

Κλάση που ενσωματώνει έναν γενικό αλγόριθμο κωδικοποιώντας και συγκεκριμένους τρόπους υλοποίησης

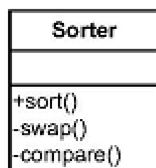
```
class Find {
public:
  void readVector( int v[], int n);
  int getMin();
  int getMax();
  double getMed();
private:
  int min, max;
  double med;
  void sort(int v[], int n);
};

void Find::readVector(int v[], int n) {
  sort(v, n);
  min = v[0];
  max = v[n-1];
  if(n%2 == 0)
    med = (v[(n/2)-1] + v[n/2]) / 2.0;
  else
    med = v[n/2];
}
```

```
int Find::getMin() { return min; }

int Find::getMax() { return max; }

double Find::getMed() { return med; }

void Find::sort(int v[], int n){   //αλγόριθμος BubbleSort
  for(int i=n-1; i>0; i--)
    for(int j=0; j<i; j++)
      if(v[j] > v[j+1])
      {
        int t = v[j];
        v[j] = v[j+1];
        v[j+1] = t;
      }
}
```

10.9 Μέθοδος υπόδειγμα – Παράδειγμα

Τα κλασσικότερα παραδείγματα για την αναγκαιότητα του προτύπου "Μέθοδος Υπόδειγμα" αφορούν στους αλγορίθμους ταξινόμησης. Κάθε αλγόριθμος μπορεί να χρησιμοποιηθεί για να ταξινομήσει οποιαδήποτε συλλογή αντικειμένων (ακεραίους, αλφαριθμητικά, φοιτητές, τραπεζικούς λογαριασμούς

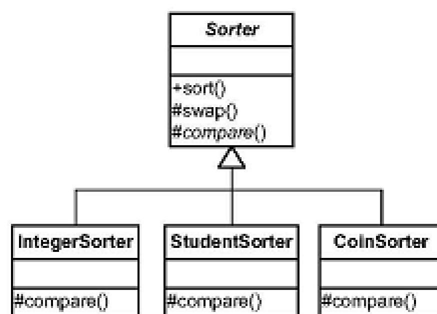
κλπ), αρκεί να υπάρχει ένας κατάλληλος τρόπος σύγκρισης δύο στοιχείων βάση των ιδιοτήτων τους. Θεωρούμε μια κλάση Sorter η οποία γνωρίζει πώς να ταξινομεί ένα διάνυσμα ακεραίων χρησιμοποιώντας τον αλγόριθμο ταξινόμησης φυσαλίδας (BubbleSort). Η κυριότερη μέθοδος της κλάσης είναι η sort() πουενσωματώνει τη γενική φιλοσοφία του αλγορίθμου. Δύο άλλες μέθοδοι, οι swap() και compare() είναι βοηθητικές. Η swap() εναλλάσσει τα περιεχόμενα δύο θέσεων τουδιανύσματος. Η μέθοδος compare() ελέγχει αν τα στοιχεία που είναι τοποθετημένα σε δύο θέσεις του διανύσματος δεν είναι ταξινομημένα ορθά και στην περίπτωση αυτή καλεί την swap(). Η σύγκριση δύο ακεραίων πραγματοποιείται χρησιμοποιώντας απλά τον τελεστή συσχέτισης '>'. Οι βοηθητικές μέθοδοι είναι δηλωμένες ως ιδιωτικές, καθώς δεν αφορούν τους πελάτες της κλάσης αλλά εξυπηρετούν μόνο την εκτέλεση της μεθόδου sort(). Το διάγραμμα της κλάσης Sorter παρουσιάζεται στο σχήμα.



Κλάση ταξινόμησης

Αν υποτεθεί, ότι αλλαγές στις απαιτήσεις επιβάλλουν και την ταξινόμηση μιας συλλογήςφοιτητών με βάση το βαθμό τους, η κλάση Sorter, αν και εμπεριέχει τον αλγόριθμο, δενμπορεί να επαναχρησιμοποιηθεί, λόγω της ιδιαιτερότητας της μεθόδου compare() σχετικά με το είδος των στοιχείων που ταξινομούνται. Με τη χρήση του προτύπου "Μέθοδος Υπόδειγμα" είναι δυνατόν να διαχωρίσουμε τον γενικό αλγόριθμο ταξινόμησης, από την υλοποίηση των βοηθητικών μεθόδων, τοποθετώντας τον αλγόριθμο σε μια μέθοδο μιας αφηρημένης βασικής κλάσης (μέθοδος υπόδειγμα sort()). Η μέθοδος υλοποιείται χρησιμοποιώντας την αφηρημένη λειτουργία compare() που δηλώνεται πλέον ως προστατευμένη μέθοδος. Η αφηρημένη αυτή λειτουργία πρέπει να υλοποιηθεί σε παράγωγες κλάσεις συμπληρώνοντας τα βήματα του αλγορίθμου που λείπουν. Με τον τρόπο αυτό, είναι δυνατόν η φιλοσοφία του αλγορίθμου να ενσωματωθεί στην βασική κλάση, έτσι ώστε να μπορεί να χρησιμοποιηθεί για την ταξινόμηση ενός διανύσματος που περιέχει ως στοιχεία αντικείμενα οποιουδήποτε τύπου. Η υλοποίηση της λειτουργίας compare(), καθώς είναι διαφορετική για κάθε είδος

στοιχείων, πραγματοποιείται σε διαφορετικές παράγωγες κλάσεις. Πολύ συχνά, οι βοηθητικές μέθοδοι που πρέπει να υλοποιηθούν στις παράγωγες κλάσεις, επισημαίνονται με ένα κατάλληλο όνομα, για παράδειγμα χρησιμοποιώντας το πρόθεμα do (δηλ. doCompare()). Σε μια παράγωγη κλάση StudentSorter η μέθοδος compare() μπορεί για παράδειγμα να συγκρίνει δύο αντικείμενα (φοιτητές) με βάση το βαθμό τους. Μια τρίτη παράγωγος κλάση θα μπορούσε να παρέχει την κατάλληλη υλοποίηση στην μέθοδο compare() ώστε να επιτρέπεται η σύγκριση (και κατά συνέπεια η ταξινόμηση) νομισμάτων. Το διάγραμμα κλάσεων για το πρότυπο σχεδίασης "Μέθοδος Υπόδειγμα" στο συγκεκριμένο παράδειγμα παρουσιάζεται στο σχήμα παρακάτω (το όνομα της λειτουργίας compare() στην αφηρημένη κλάση Sorter είναι με πλάγιους χαρακτήρες, υποδηλώνοντας ότι είναι αφηρημένη μέθοδος άνευ υλοποίησης).



Πρότυπο "Μέθοδος Υπόδειγμα" για το πρόβλημα ταξινόμησης

Θεωρούμε ότι σε ένα σύστημα υπάρχουν αντικείμενα μιας κλάσης Φοιτητής (Student) τα οποία πρέπει να ταξινομηθούν με βάση το βαθμό του κάθε φοιτητή. Ο κώδικας της κλάσης Student σε Java, δίνεται στη συνέχεια:

```
//Student.java

public class Student {
  private String name;
  private int grade;

  public Student(String text, int num) {
    name = text;
    grade = num;
  }

  public String getName() { return name; }

  public int getGrade() { return grade; }
}
```

Η υλοποίηση του προτύπου συνίσταται στη δημιουργία μιας μεθόδου-υποδείγματος ταξινόμησης sort() σε μια αφηρημένη βασική κλάση Sorter. Η κλάση περιλαμβάνει επιπλέον τις προστατευμένες μεθόδους swap() και compare(). Η

μέθοδος compare()είναι αφηρημένη και κατά συνέπεια ο κώδικας υλοποίησής της θα πρέπει να παρέχεταιστις παράγωγες κλάσεις της Sorter. Επειδή η σύγκριση δύο στοιχείων ενδέχεται να απαιτήσει την αντιμετάθεσή τους (αν τα δύο στοιχεία βρίσκονται σε λάθος σειρά με βάσηκάποια, αύξουσα ή φθίνουσα, διάταξη), η μέθοδος swap() δηλώνεται ως προστατευμένη(και όχι ως ιδιωτική) ώστε να μπορεί να προσπελαστεί από τις παράγωγες κλάσεις.

```
//Sorter.java

import java.util.ArrayList;

public abstract class Sorter {
  protected ArrayList list;

  public void setList(ArrayList collection) {
    list = collection;
  }

  public void sort() {     //Μέθοδος - Υπόδειγμα
          //Αλγόριθμος Ταξινόμησης
    if(list.size() <= 1)
     return;

    for(int end = list.size() - 1; end >= 1; end-- )
      for(int start = 0; start < end; start++)
       compare(start, end);
  }
          //Μέθοδος εναλλαγής θέσης 2 στοιχείων
  protected void swap(int i, int j) {
      Object temp;

      temp = list.get(i);
      list.set(i, list.get(j) );
      list.set(j, temp);
  }
          //Αφηρημένη μέθοδος σύγκρισης 2 στοιχείων
  protected abstract void compare(int i, int j);
}
```

Για την ταξινόμηση φοιτητών, στην παράγωγη κλάση StudentSorter υλοποιείται η αφηρημένη μέθοδος compare() χρησιμοποιώντας τη μέθοδο getGrade() κάθε φοιτητή για τη λήψη του βαθμού και τη σύγκριση δύο στοιχείων.

```
//StudentSorter.java

public class StudentSorter extends Sorter {

          //Υλοποίηση μεθόδου σύγκρισης για Φοιτητές
  protected void compare(int i, int j)
  {
    Student si = (Student)list.get(i);
    Student sj = (Student)list.get(j);

    if(si.getGrade() > sj.getGrade() )
      swap(i, j);
  }
}
```

Ο έλεγχος του προγράμματος μπορεί να πραγματοποιηθεί με το ακόλουθο πρόγραμμα ελέγχου:

```
    for(int i = 0; i < list.size(); i++) {
      Student s = (Student)list.get(i);
      System.out.println(s.getName() + " : " + s.getGrade() );
    }
  }
}
```

```
Before sorting:
John : 5
Nick : 3
Bob : 2
George : 4
After sorting:
Bob : 2
Nick : 3
George : 4
John : 5
```