ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

Τεχνικές Αυτοματοποιημένου Ελέγχου Λογισμικού

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΜΑΘΙΟΥΔΑΚΗ ΕΛΕΝΗ

Επιβλέπων : Στεφανέας Πέτρος
Λέκτορας Ε.Μ.Π.

Αθήνα, Μάρτιος 2015

Ευχαριστίες

Για την εκπόνηση της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω αρχικά τον επιβλέπων καθηγητή μου Δρ. Πέτρο Στεφανέα, αλλά και όλους τους καθηγητές της Σχολής που συνέβαλαν στην ακαδημαϊκή μου διαπαιδαγώγηση, μέσα και έξω από τα αμφιθέατρα. Θα ήθελα επίσης να τονίσω τη σημαντική συμβολή του κ. Ηλία Αλεξάκη, Deputy Project Manager, για την εισαγωγή μου στον τομέα του ελέγχου λογισμικού και την ενθάρρυνση του τα τελευταία χρόνια.
Εν συνεχεία, θα ήθελα να ευχαριστήσω από καρδιάς τους γονείς μου και την αδερφή μου, για την υπομονή και εμπιστοσύνη που μου έδειξαν, αλλά και για την υποστήριξη τους, οικονομική και ηθική, για να καταφέρω να ανταπεξέλθω στις υποχρεώσεις των σπουδές μου. Συμπαραστάτες και συνοδοιπόροι μου σε όλα τα χρόνια της φοιτητικής μου πορείας στάθηκαν οι αδερφικοί μου φίλοι, που χωρίς αυτούς δε θα ήμουν εδώ και δεν θα ήμουν εγώ. Τέλος, δε θα μπορούσα να μην αναφερθώ και να ευχαριστήσω τους Κνουτ, Χανς και Μάρκους που βρίσκονται ξανά εδώ για να με δουν να ολοκληρώνω τις σπουδές μου.
Αφιερώνεται στους λύκους που δεν χορεύουν πια.

Περίληψη

Η παρούσα διπλωματική εργασία με τίτλο «Τεχνικές αυτοματοποιημένου ελέγχου λογισμικού» εκπονήθηκε στη Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών του Εθνικού Μετσοβίου Πολυτεχνίου υπό την επίβλεψη του λέκτορα Δρ. Πέτρου Στεφανέα.

Αρχικά πραγματοποιείται μια εισαγωγή στην ποιότητα και τον έλεγχο λογισμικού, ορίζοντας τις απαιτούμενες έννοιες καθώς και μια σύντομη παρουσίαση των πιο γνωστών μοντέλων ποιότητας λογισμικού.

Έπειτα ο έλεγχος λογισμικού αναλύεται σε επίπεδα ελέγχου βάσει του V- μοντέλου ανάπτυξης λογισμικού, που περιλαμβάνουν τα επίπεδα μονάδας, συνένωσης, συστήματος κι αποδοχής. Ακολούθως ο έλεγχος κατηγοριοποιείται βάσει των τύπων ελέγχου σε λειτουργικό, μη λειτουργικό, δομικό και παλινδρόμισης. Παρουσιάζονται οι τεχνικές ελέγχου, διαχωριζόμενες σε στατικές, που περιλαμβάνουν την αναθεώρηση, τη στατική ανάλυση και συμβολική εκτέλεση κώδικα, καθώς και τις δυναμικές, που περιλαμβάνουν δύο βασικές κατηγορίες μεθόδων, black box και white box ελέγχου.

Στη συνέχεια αναπτύσσεται η έννοια του αυτοματοποιημένου ελέγχου, διερευνούνται οι συνθήκες κάτω απ' τις οποίες επιλέγεται ως πρακτική, αναφέρονται τα πλεονεκτήματα και τα μειονεκτήματά του, επεξηγείται για ποιές κατηγορίες ενδείκνυται και στην περίπτωση του λειτουργικού ελέγχου αναλύεται με βάση τα επίπεδα ελέγχου. Έπειτα παρουσιάζονται οι τεχνικές αυτοματοποίσης των ελέγχων, οι οποίες αποτελούν την τεχνική «καταγραφής/επανάληψης», τη γραμμική, την δομημένη τεχνική, την τεχνική βάσει δεδομένων, την τεχνική βάση λέξεων-κλειδιά και οι υβριδικές τεχνικές, ως πιθανοί συνδυασμοί όλων των προηγούμενων. Διαχωρίζονται τα εργαλεία αυτοματοποιημένου ελέγχου ανάλογα με την προσβασιμότητα στον κώδικα και την διάθεσή τους αλλά και μια λίστα με τα πιο διαδεδομένα σημερινά πλαίσια αυτοματοποιημένου ελέγχου, κατηγοροποιημένα ανάλογα με το επίπεδο ελέγχου στο οποίο εξειδικεύονται.

Τέλος, ως εφαρμογή της θεωρίας ελέγχου λογισμικού, χρησιμοποιείται το «πρόβλημα των τριγώνων», στο οποίο εφαρμόζεται ένα σύνολο περιπτώσεων ελέγχου αποδοχής με το πλαίσιο αυτοματοποιημένου ελέγχου Robot, που προέκυψαν ύστερα από ανάλυση κι έλεγχο βάσει απαιτήσεων. Σκοπός είναι να παρουσιαστεί η διαδικασία σχεδιασμού και κατασκευής σεναρίων ελέγχου στα πλαίσια της black box μεθόδου κι η αξιολόγηση της χρήσης αυτοματοποιημένου ελέγχου ώστε να επαληθευτούν τα συμπεράσματα για την επιλογή του σε σχέση με τον χειρωνακτικό» έλεγχο.

Abstract

The subject of this thesis is “Automated software testing techniques” and was carried out at the School of Applied Mathematics and Physics (National Technical University of Athens) under the supervision of lecturer Dr. Petros Stefaneas.

First of all, the concepts of software quality and software testing are introduced, by defining the necessary terms, as well as a brief presentation of the best known software quality models.

Software testing is then analyzed into levels based on the V- model of software development, which include the unit, integration, system and acceptance test levels. Subsequently, based on its types testing is categorized into functional, non-functional, structural and regression testing. Testing techniques are presented divided into the static ones, which include reviews, static analysis and symbolic executions (of the code) and the dynamics ones, which include two basic method categories, black box and white box methods.

Thereafter, the concept of test automation is defined and described; the conditions under which it’s selected as a practice are explored, its advantages and disadvantages are indicated, the categories for which is appropriate are explained and in the case of functional testing it is analyzed bases on the test levels. Furthermore, the test automation techniques are presented, which are the “record/ playback” technique, linear scripting, modular scripting, data- driven testing and keyword driven testing, as well as the hybrid techniques, as any possible combination of all the previous basic ones. Automated testing tools are cited and distinguished according to code access and their availability, along with a list with the current, most popular test automation frameworks, depending on the test level they specialize in.

Finally, an application of the software testing theory is presented into the “triangle problem”, on which a set of test cases of automated acceptance testing is applied on a web application using the Robot Framework. Those test cases were derived after analyzing the problem and choosing to conduct requirements- based testing. The objective is to present the test case design and test suite implementation process in the context of the black box method and to evaluate the usage and impact of test automation, in order to verify it over manual testing.

Περιεχόμενα

1

Εισαγωγή

1.1. Γενικοί ορισμοί [1] [2] [3]

Όταν η συμπεριφορά ενός προϊόντος λογισμικού δεν είναι η επιθυμητή, δηλαδή όταν δεν πληρούνται οι *απαιτήσεις (requirements)* και οι *προδιαγραφές (specifications)*, τότε έχουμε ένα *σφάλμα (error)* ή μια *αποτυχία (failure)*. *Το* σφάλμα αυτό δημιουργείται από ένα *ελάττωμα στο λογισμικό (defect*), το οποίο συνήθως συμβαίνει είτε από λάθος του προγραμματιστή, όπως στην περίπτωση λάθος εντολών, ξεχασμένου κώδικα κτλ. είτε από λάθος του συστήματος.

Ορίζουμε έτσι ως *Έλεγχο Λογισμικού (Software Testing)* τη διαδικασία εκτέλεσης του λογισμικού χρησιμοποιώντας ένα επιλεγμένο σύνολο από δεδομένα με σκοπό την ανίχνευση των ελαττωμάτων, την ανάλυση του κώδικα για την πρόληψη ελαττωμάτων και την αξιολόγηση της *ποιότητά του λογισμικού (software quality)*.

Οι επιπτώσεις από παράλειψη της διαδικασίας του ελέγχου λογισμικού μπορεί να οδηγήσουν σε:

- Αύξηση του κόστους παραγωγής του λογισμικού
- Κίνδυνο διακοπής λειτουργίας του λογισμικού σε απρόβλεπτο σημείο και με απρόβλεπτες συνέπειες
- Μείωση της αξιοπιστίας του λογισμικού
- Μείωση της αξιοπιστίας της εταιρείας

Οι *περιπτώσεις ελέγχου (test cases)* ορίζονται ως ένα σύνολο *δεδομένων εισόδου* (*input*), προϋποθέσεων, αναμενόμενων αποτελεσμάτων, *δεδομένων εξόδου* (*output*) και *κριτηρίων εξόδου* (*exit criteria*) πάνω στο *αντικείμενο υπό έλεγχο* (*Subject Under Test – SUT*) λογισμικού. Μπορούν να διαχωριστούν σε δύο κατηγορίες σε αυτές που ελέγχουν την αναμενόμενη συμπεριφορά του λογισμικού, *έγκυρες (expected test cases)*, και σε αυτές που εξετάζουν τη συμπεριφορά του αντικειμένου για μη αναμενόμενα δεδομένα εισόδου, *μη έγκυρες (unexpected test case)* περιπτώσεις ελέγχου.

Εκτέλεση ελέγχου (test run) ονομάζεται η εκτέλεση ενός ή περισσότερων περιπτώσεων ελέγχου, ενώ *πακέτο σεναρίων ελέγχου* ή απλά *σενάρια ελέγχου* (*test*

procedure/test suite) είναι ένα σύνολο περιπτώσεων ελέγχου, όπου το *εξαγόμενο αποτέλεσμα ελέγχου* (*output*) αποτελεί το *δεδομένο εισόδου* (*input*) της επόμενης περίπτωσης ελέγχου.

Η διαδικασία εκτέλεσης ενός συγκεκριμένου συνόλου περιπτώσεων ελέγχου πάνω στο αντικείμενο υπό έλεγχο με σκοπό την ανίχνευση σφαλμάτων, ονομάζεται *έλεγχος (test)*. Η οργάνωση, ο σχεδιασμός, η υλοποίηση και η ανάλυση ενός ελέγχου είναι βασικό κομμάτι του ελέγχου λογισμού (test ≤ testing).

Ο εντοπισμός κα η διόρθωση ελαττωμάτων *(debugging)* συνιστά δουλειά του προγραμματιστή και είναι κάτι εντελώς διαφορετικό από τον έλεγχο λογισμικού, γι'αυτό οι δύο αυτές έννοιες δεν πρέπει να συγχέονται.

1.2. Βασικές Αρχές [2]

Οι επτά βασικές αρχές του Ελέγχου Λογισμικού είναι οι εξής:

1. Ο έλεγχος δείχνει την παρουσία ελαττωμάτων λογισμικού και όχι την απουσία τους. Μπορεί να καταδείξει ελαττώματα, αλλά δε μπορεί να αποδείξει ότι δεν υπάρχει κανένα άλλο ελάττωμα. Δηλαδή, ο έλεγχος μειώνει την πιθανότητα ύπαρξης κρυφών ελαττωμάτων στο λογισμικό, αλλά ακόμα και αν δεν ευρεθούν ελαττώματα, αυτό δε σημαίνει ότι δεν υπάρχουν.

2. Ο διεξοδικός έλεγχος είναι αδύνατος.

3. Οι διαδικασίες ελέγχου πρέπει να ξεκινάνε το νωρίτερο δυνατό. Οι δραστηριότητες του ελέγχου πρέπει να ενσωματώνονται στον κύκλο ζωής του λογισμικού (software lifecycle).

4. Τα ελαττώματα λογισμικού συγκεντρώνονται μαζεμένα (cluster). Η πιθανότητα ύπαρξης πρόσθετων σφαλμάτων σε ένα κομμάτι λογισμικού είναι συνήθως ανάλογη του συνόλου των σφαλμάτων που έχουν ανιχνευθεί εκεί.

5. Αν οι ίδιες περιπτώσεις ελέγχου τρέξουν επανειλημμένα στο αντικείμενο υπό έλεγχο, κάποια στιγμή δε βρίσκονται νέα ελαττώματα (*παράδοξο του παρασιτοκτόνου - pesticide paradox)*. Για το λόγο αυτό, οι περιπτώσεις έλεγχου πρέπει να επανεξετάζονται και να διορθώνονται συστηματικά, και νέες περιπτώσεις ελέγχου πρέπει να γράφονται για τον έλεγχο διαφορετικών κομματιών του κώδικα ή του συστήματος.

6. Ο έλεγχος λογισμικού εξαρτάται από το περιεχόμενο και το περιβάλλον. Για κάθε προϊόν λογισμικού διαφορετικού περιεχομένου ακολουθείται

διαφορετικός έλεγχος. Για παράδειγμα, το λογισμικό πρόγραμμα για τα τελωνεία μιας χώρας ελέγχεται διαφορετικά από μια σελίδα ηλεκτρονικού εμπορίου.

7. Το λάθος του «δεν υπάρχουν άλλα σφάλματα». Η ανίχνευση και η διόρθωση των ελαττωμάτων λογισμικού δε σημαίνει ότι το σύστημα είναι έτοιμο και ικανοποιεί τις προδιαγραφές και τις ανάγκες του πελάτη ή χρήστη.

1.3. Ποιότητα λογισμικού [4] [5]

Βασικός στόχος του ελέγχου λογισμικού είναι επίσης η βελτίωση της ποιότητας του λογισμικού, όχι μόνο ανιχνεύοντας τα σφάλματα και ελαχιστοποιώντας τα ελαττώματα του, αλλά αποκτώντας γνώση πάνω στα ποιοτικά χαρακτηριστικά του.

Σύμφωνα με το Πρότυπο *IEEE Std 729-1983* ως ποιότητα λογισμικού ορίζεται:

a. Το σύνολο γνωρισμάτων και χαρακτηριστικών ενός προϊόντος που εξυπηρετούν δεδομένες ανάγκες και προδιαγραφές.
b. Ο βαθμός που το λογισμικό κατέχει ένα επιθυμητό συνδυασμό από ιδιότητες.
c. Ο βαθμός στον οποίο ο πελάτης ή ο χρήστης αντιλαμβάνεται ότι το προϊόν ικανοποιεί τις προσδοκίες του.
d. Τα σύνθετα χαρακτηριστικά του λογισμικού τα οποία καθορίζουν το βαθμό που το λογισμικό υπό χρήση ικανοποιεί τις προσδοκίες του πελάτη.

Δεδομένου της αφηρημένης έννοιας του λογισμικού, ερευνητές αλλά και επαγγελματίες του χώρου αναζητούν τρόπους *μέτρησης της ποιότητας* (*metrics*) από τις αρχέστης δεκαετία του 70. Διάφορα μοντέλα ποιότητας λογισμικού έχουν ανατπυχθεί, από τα οποία έχουν επικρατήσει μέχρι σήμερα τα παρακάτω μοντέλα:

- McCall (1977)
- Boehm (1978)
- FURPS, FURPS+
- Dromey (1995)
- ISO 9000, ISO 9126 (1993)

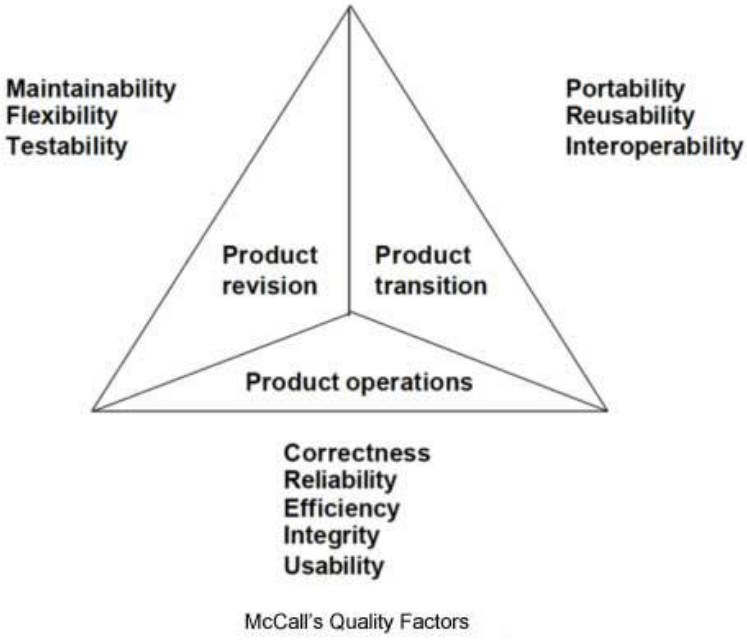
1.3.1 Μοντέλο McCall

Το πιο παλιό αλλά και πιο διαδεδομένο μοντέλο, το μοντέλο McCall, δημιουργήθηκε για τις US Air force Electronic System Division (ESD), Rome Air Development Centre (RADC) και General Electric. Στο μοντέλο αυτό αναγνωρίστικαν αρχικά 55 παράγοντες μεγάλης επιρροής στην ποιότητα λογισμικού, και στη συνέχεια

κατέληξαν, για λόγους οικονομίας, σε 11 *παράγοντες ποιότητας (quality factors)* , οι οποίοι είναι οι εξής:

1. *Συντηρησιμότητα (Maintainability)*
2. *Προσαρμοστικότητα (Flexibility)*
3. *Ελεγξιμότητα (Testability)*
4. *Μεταβιβασιμότητα (Portability)*
5. *Χρησικανότητα (Reusability)*
6. *Διασυνδεσιμότητα (Interoperability)*
7. *Σωστή λειτουργία (Correctness)*
8. *Αξιοπιστία (Reliability)*
9. *Αποτελεσματικότητα (Efficiency)*
10. *Ακεραιότητα (Integrity)*
11. *Χρησικανότητα (Usability)*.

Το μοντέλο McCall αναπτύχθηκε για να προσδιορίσει τις σχέσεις μεταξύ των εξωτερικών παραγόντων και των κριτηρίων ποιότητας του προϊόντος. Για το λόγο αυτό, οι ποιοτικοί παράγοντες διαχωρίζονται σε τρεις βασικές κατηγορίες:



Σχήμα 1.1: Απεικόνιση των ποιοτικών παραγόντων του μοντέλου McCall.

Σύμφωνα με την ικανότητα του λογισμικού να αλλάζει (*product revision*), διακρίνονται οι παρακάτω παράγοντες:

1. *Συντηρησιμότητα:* Η ικανότητα να ανιχνεύει και να διορθώνει τα ελαττώματα του.
2. *Προσαρμοστικότητα:* Η ικανότητα να αλλάζει σύμφωνα με της απαιτήσεις των προδιαγραφών.
3. *Ελεγξιμότητα:*Η ικανότητα να επικυρώνει τις προδιαγραφές του.

Επιπλέον, η προσαρμοστικότητά του προϊόντος σε νέα περιβάλλοντα (*product transition*) αναγνωρίζει τους εξής παράγοντες:

4. *Μεταβιβασιμότητα*: Η ικανότητα να μεταφέρεις το λογισμικό από ένα περιβάλλον σε ένα άλλο.
5. *Χρησικανότητα*: Η δυνατότητα να επαναχρησιμοποιηθούν υπάρχοντα κομμάτια λογισμικού σε διαφορετικά σημεία.
6. *Διασυνδεσιμότητα*: Ο βαθμός στον οποίο μπορούν να λειτουργήσουν μαζί διαφορετικά κομμάτια λογισμικού.

Τέλος τα βασικά χαρακτηριστικά διαχείρισης του προϊόντος (*product operations*). αφορούν τους παράγοντες:

7. *Σωστή λειτουργία:* Ολα τα χαρακτηριστικά ικανοποιούν τις προδιαγραφές του προιόντος.
8. *Αξιοπιστία:* ο βαθμός στον οποίο το σύστημα αποτυγχάνει.
9. Αποτελεσματικότητα: Σωστή χρήση των πόρων (CPU, μνήμη, δίκτυο) του συστήματος.
10. *Ακεραιότητα:* Προστασία από μη εγκεκριμένη πρόσβαση και χρήση.
11. *Χρησικανότητα:* Το μέτρο ικανότητας ενός προϊόντος να ανταπεξέλθει στις ανάγκες των χρηστών και η ευκολία χρήσης του.

Η μεγαλύτερη συνεισφορά του μοντέλου McCall είναι η σύνδεση μεταξύ ποιοτικών παραγόντων και μετρικών. Παρολα αυτά, το μοντέλο δεν λαμβάνει άμεσα υπόψη του τη *λειτουργικότητα (functionality)* του προϊόντος λογισμικού.

1.3.2 Μοντέλο Boehm

Το μοντέλο Boehm αναπτύχθηκε το 1978 με βάση το μοντέλο McCall, προσθέτοντας ένα δεύτερο σύνολο ποιοτικών παραγόντων με έμφαση στη συντηρησιμότητα. Η επιπλέον ποιοτικοί παράγοντες του συγκεκριμένου μοντέλου είναι:

- *Σαφήνεια (Clarity)*
- *Επεξεργασιμότητα (Modifiability)*
- *Επίσημα έγγραφα (Documentation)*
- *Ανθεκτικότητα στην αποτυχία (Resilience)*
- *Κατανόηση (Understandability)*
- *Εγκυρότητα (Validity)*
- *Λειτουργικότητα (Functionality)*
- *Γενικότητα (Generality)*
- *Εξοικονόμηση (Economy)*

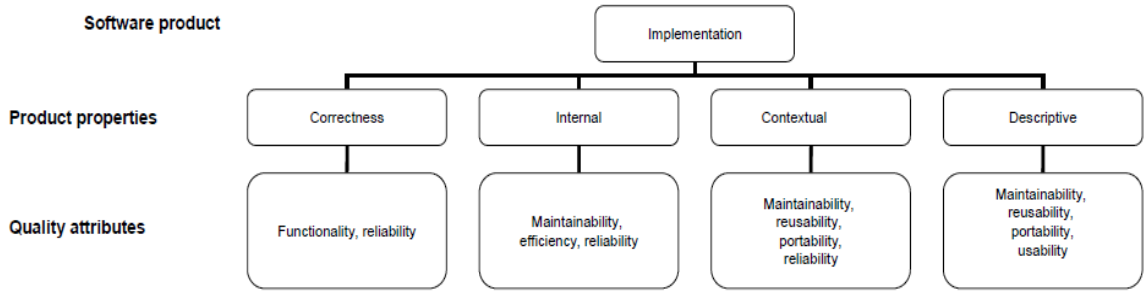
Ο στόχος του μοντέλου αυτού ήταν να προσδιορίσει τα ελαττώματα των μοντέλων που αυτόματα και ποσοτικά αξιολογούν την ποιότητα του λογισμικού. Το μοντέλο αναπαριστά τα χαρακτηριστικά του λογισμικού ιεραρχικά και με αναφορά στη χρησικανότητα του προγράμματος ή της εφαρμογής. Ωστόσο, το μοντέλο περιέχει μόνο διαγράμματα χωρίς καμία αναφορά σε μέτρηση των ποιοτικών παραγόντων.

1.3.3 Μοντέλο FURPS και FURPS+

Το μοντέλο FURPS προτάθηκε από τον Robert Grady και την εταιρία Hewlett-Packard Co. Τα χαρακτηριστικά του ταξινομούνται σε δύο κατηγορίες, σύμφωνα με τις λειτουργικές και μη λειτουργικές προϋποθέσεις. *Λειτουργικές προϋποθέσεις (functional requirements)* ορίζονται τα εισερχόμενα και τα αναμενόμενα εξερχόμενα, ενώ μη λειτουργικές προϋποθέσεις (non-functional requirements) θεωρούνται η χρησικανότητα, η αξιοπιστία, η επίδοση (performance) και η υποστήριξη (supportability). Στη συνέχεια το μοντέλο επεκτάθηκε από την IBM Rational Software σε FURPS+. Στην τελική του μορφή το μοντέλο λαμβάνει υπόψη του τις προϋποθέσεις του χρήστη αλλά αψηφά τις απαιτήσεις του προγραμματιστή. Επιπλέον, αποτυγχάνει να συμπεριλάβει σημαντικά χαρακτηριστικά του προϊόντος όπως τη μεταβιβασιμότητα και τη συντηρησιμότητα.

1.3.4 Μοντέλο Dromey

Το μοντέλο Dromey παρουσιάστηκε πρώτη φορά το 1995 από τον R. Geoff Dromey και αναφέρει ότι η εξέλιξη κάθε προϊόντος λογισμικού είναι διαφορετική, με την δυναμική προσέγγιση του μοντέλου να είναι απαραίτητη. Για το λόγο αυτό, η βασική ιδέα του μοντέλου είναι να είναι αρκετά ευρύ ώστε να εφαρμόζεται σε διαφορετικά συστήματα. Το μοντέλο ορίζει 2 επίπεδα χαρακτηριστικών ποιότητας, τα ανώτερα χαρακτηριστικά (high-level attributes) και τα κατώτερα χαρακτηριστικά (subordinate attributes), επιδιώκοντας να αυξήσει την κατανόηση μεταξύ των δύο επιπέδων χαρακτηριστικών. Επίσης, προσπαθεί να συνδέσει τα γνωρίσματα του προϊόντος λογισμικού και τα γνωρίσματα ποιότητας λογισμικού.

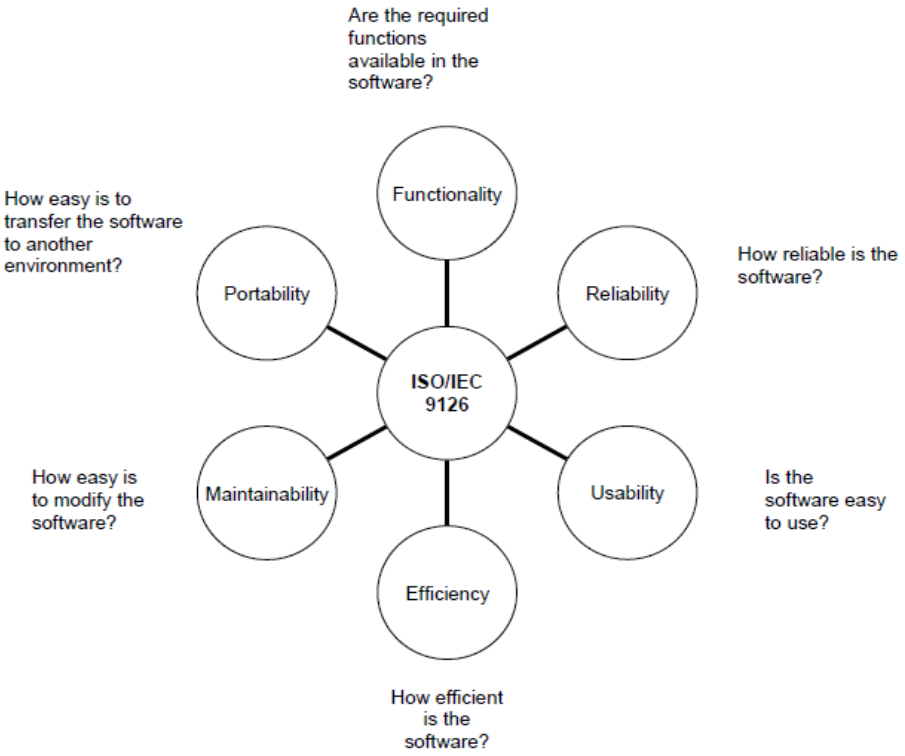


Σχήμα 1.2: Σχεδιαγραμματική απεικόνιση των γνωρισμάτων προϊόντος και γνωρισμάτων ποιότητας λογισμικού κατά Dromey.

1.3.5 Μοντέλο ISO

Με την ύπαρξη τόσο πολλών και διαφορετικών μοντέλων δεν άργησε να υπάρξει μια σύγχυση ως προς το πώς ορίζεται και μετράται η ποιότητα λογισμικού. Έτσι, η ISO/IEC JTC1 [1] δημιούργησε σειρές προτύπων, όπως το ISO 9000 , το οποίο είναι το πιο παλιό και σημαντικό στο τομέα του *Ελέγχου Ποιότητας* (*Quality Assurance*). Επίσης το 1993 αναπτύχθηκε το πρότυπο ISO 9126: *Αξιολόγηση Προϊόντος Λογισμικού: Πρότυπο για τα Ποιοτικά Χαρακτηριστικά και Οδηγοί για τη Χρήση τους (Software Product Evaluation - Quality Characteristics and Guidelines for their Use – standard)*. Το πρότυπο αυτό βασίστηκε στο μοντέλο McCall και Boehm προσθέτοντας ένα νέο παράγοντα, τη λειτουργικότητα (functionality), και αναγνωρίζοντας τα εσωτερικά και εξωτερικά χαρακτηριστικά ποιότητας του προϊόντος λογισμικού. Επίσης, διαχωρίζει τους ποιοτικούς παράγοντες σε έξι κατηγορίες: Λειτουργικότητα, Αξιοπιστία, Αποτελεσματικότητα, Συντηρησιμότητα, Μεταβιβασιμότητα και Χρησικανότητα.

[1] ISO/IEC JTC 1: κοινή επιτροπή του Διεθνή Οργανισμού Προτυποποίησης ISO (International Organization for Standardization) και της Διεθνής Ηλεκτροτεχνικής Επιτροπής IEC (International Electrotechnical Commission). Σκοπός της είναι να αναπτύξει, να διατηρήσει και να προωθήσει τα πρότυπα στον τομέα της Πληροφορικής (IT) και των Τηλεπικοινωνιών (ICT). Το ISO επιλέχτηκε στη θέση του IOS διότι στα ελληνικά ISO σημαίνει ίσο, εκφράζοντας έτσι την ισότητα και την ιδέα για κοινά πρότυπα για όλους.

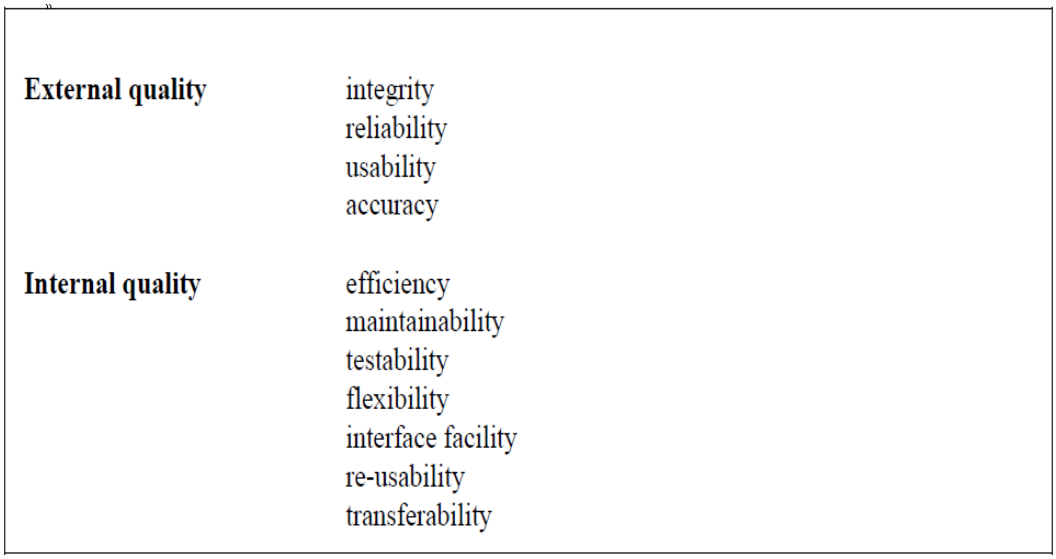


Σχήμα 1.3: Σχεδιαγραμματική απεικόνιση των ποιοτικών παραγόντων λογισμικού κατά ISO 9126.

Criteria/goals	*McCall, 1977*	*Boehm, 1978*	*ISO 9126, 1993*
Correctness	*	*	maintainability
Reliability	*	*	*
Integrity	*	*	
Usability	*	*	*
Effiency	*	*	*
Maintainability	*	*	*
Testability	*		maintainability
Interoperability	*		
Flexibility	*	*	
Reusability	*	*	
Portability	*	*	*
Clarity		*	
Modifiability		*	maintainability
Documentation		*	
Resilience		*	
Understandability		*	
Validity		*	maintainability
Functionality			*
Generality		*	
Economy		*	

Πίνακας 1: Συγκριτική παρουσίαση παραγόντων ποιότητας λογισμικού.

Ολοκληρώνοντας τις κατηγοριοποιήσεις των παραγόντων ποιότητας με βάση τα διάφορα κριτήρια που θέτουν τα μοντέλα, υπάρχει ένας ακόμα γενικός διαχωρισμός των παραγόντων, σε αυτά που συνδέονται με την *εξωτερική ποιότητα (external quality)* και την *εσωτερική ποιότητα (internal quality).* Ως εξωτερική ποιότητα ορίζεται η ποιότητα του τελικού προϊόντος, δηλαδή η ποιότητα όπως εμφανίζεται στον κόσμο όταν τελειώνει από τη γραμμή παραγωγής. Από την άλλη πλευρά εσωτερική ονομάζεται η ποιότητα του προϊόντος όταν αυτό παράγεται, δηλαδή κατά τη διάρκεια της γραμμής παραγωγής. Γενικά, θα μπορούσαμε να πούμε ότι η εξωτερική ποιότητα συνίσταται στα χαρακτηριστικά που είναι σαφώς εμφανή στον τελικό χρήστη, ενώ εσωτερική ποιότητα θεωρείται το σύνολο των τεχνικών χαρακτηριστικών του λογισμικού. Σύμφωνα με τον Ghezzi *et al.*(1991), «σε γενικές γραμμές, οι χρήστες ενδιαφέρονται μόνο για την εξωτερική ποιότητα, αλλά η εσωτερική ποιότητα είναι αυτή που βοηθάει τους προγραμματιστές να πετύχουν την εξωτερική.»



Σχήμα 1.4: Κατηγοροποίηση των ποιοτικών παραγόντων λογισμικού ως προς την εσωτερική και την εξωτερική ποιότητα του προϊόντος

2

Έλεγχος λογισμικού

2.1. Το V- Μοντέλο [1] [2]

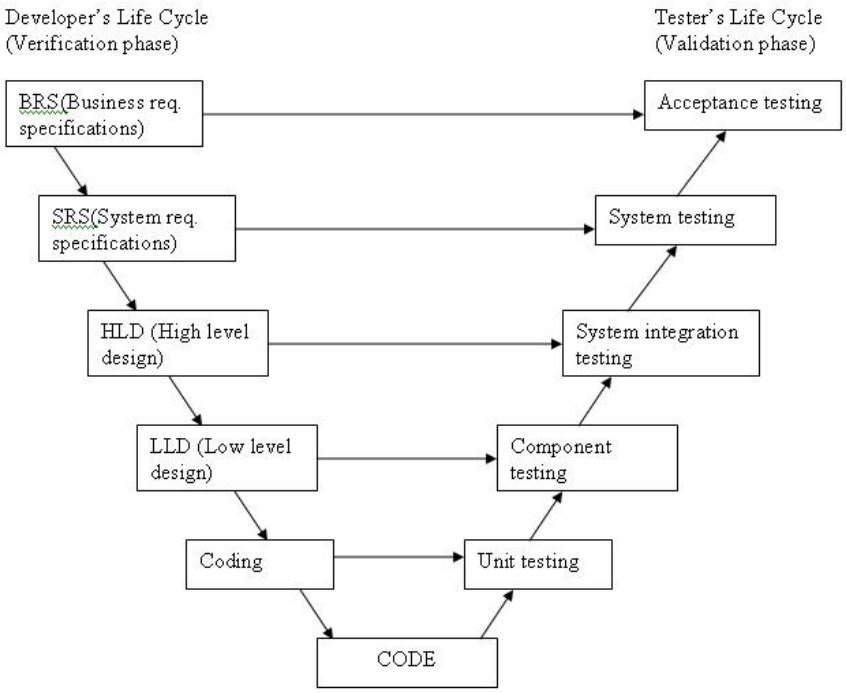
Όλες οι δραστηριότητες ελέγχου λογισμικού δεν υπάρχουν αποκλειστικά μόνες τους αλλά σε σχέση με τις δραστηριότητες ανάπτυξης λογισμικού. Εάν η μεθοδολογία της ανάπτυξης λογισμικού αλλάξει, τότε άμεσα αλλάζει και η προσέγγιση του ελέγχου λογισμικού. Ο κύκλος ζωής του λογισμικού αποτελεί μια διαδικασία που περιλαμβάνει το *σχεδιασμό*, τη *δημιουργία*, τον *έλεγχο*, την *εφαρμογή* και τη *συντήρηση* του προϊόντος *(planning, creating, testing, deploying and maintaining).* Στο κεφάλαιο αυτό περιγράφεται ο ρόλος του ελέγχου λογισμικού μέσα στο κύκλο ζωής του λογισμικού.

Μία από τις βασικές αλλά και πιο παλιές μεθοδολογίες ανάπτυξης λογισμικού είναι το μοντέλο Waterfall. Το μοντέλο αυτό περιγράφει τις δραστηριότητες του κύκλου ζωής του λογισμικού γραμμικά, από τις προδιαγραφές στην εφαρμογή και στη διατήρηση. Ο έλεγχος λογισμικού σε αυτό το μοντέλο είναι το τελικό βήμα. Μια βελτιωμένη έκδοση του Waterfall είναι το γενικό V-μοντέλο, το οποίο υποδεικνύει πως ο έλεγχος λογισμικού μπορεί να ενσωματωθεί σε κάθε φάση της διαδικασίας του κύκλου ζωής.

Το V-μοντέλο έχει τη μορφή του λατινικού γράμματος V. Τα δύο κλαδιά του συμβολίζουν την ισοδύναμη σημασία της ανάπτυξης και του ελέγχου λογισμικού. Το αριστερό τμήμα ορίζει τα επίπεδα ανάπτυξης από τη πλευρά του προγραμματιστή ενώ το δεξί τμήμα ορίζει τα επίπεδα ελέγχου για κάθε επίπεδο ανάπτυξης από τη πλευρά του *ειδικού ελέγχου (tester).*

Τα επίπεδα ελέγχου σύμφωνα με αυτό το μοντέλο, είναι οι *Δοκιμασίες Μονάδας* (*Unit testing και Component testing*)[2], οι *Δοκιμασίες Συνένωσης* (*Integration testing*), οι *Δοκιμασίες Συστήματος* (*System testing*) και οι *Δοκιμασίες Αποδοχής* (*Acceptance testing*).

[2] Το unit testing είναι το ίδιο με το component testing με τη μόνη διαφορά ότι το δεύτερο μπορεί να καλεί και άλλες μονάδες με τη βοήθεια drivers. Συνήθως αποτελεί ένα επίπεδο ελέγχου.



Σχήμα 2.1: Σχηματική αναπαράσταση του γενικού V-μοντέλου.

Επίσης, το V-μοντέλο αντιπροσωπεύει τα verification και validation, δηλαδή την *επαλήθευση* και την *επικύρωση*, δυο σημαντικές διαδικασίες ελέγχου. Η *επαλήθευση (verification)* είναι η διαδικασία αξιολόγησης ενός συστήματος για το κατά πόσο αυτό ικανοποιεί τις συνθήκες-απαιτήσεις που τέθηκαν αρχικά. Η αξιολόγηση γίνεται με *φορμαλιστικές μεθόδους* (*formal methods*) και η διαδικασία απαντά στο ερώτημα «φτιάχνουμε το προϊόν σωστά;» ("are we building the product right?"). Από την άλλη πλευρά, η *επικύρωση (validation)* είναι η διαδικασία αξιολόγησης ενός συστήματος στο τέλος της ανάπτυξής του για να βεβαιώσουμε ότι είναι ελεύθερο σφαλμάτων και ικανοποιεί τις προδιαγραφές. Η αξιολόγηση αυτή γίνεται με δεδομένα και η διαδικασία απάντα στο ερώτημα «φτιάχνουμε το σωστό προϊόν;» ("are we building the right product?").

2.2. Επίπεδα ελέγχου [1]

Όπως αναφέραμε και προηγουμένως, τα επίπεδα ελέγχου ενός λογισμικού προϊόντος είναι τέσσερα:

a. Οι *Δοκιμασίες Μονάδας* (*Unit testing και Component testing*) ελέγχουν εάν η μονάδα λογισμικού λειτουργεί σωστά και σύμφωνα με τις απαιτήσεις.
b. Οι *Δοκιμασίες Συνένωσης* (*Integration testing*) ελέγχουν εάν οι ομάδες μονάδων λογισμικού ανταποκρίνονται μεταξύ τους σωστά και σύμφωνα με το σχέδιο συστήματος (*technical system design*).

c. Οι *Δοκιμασίες Συστήματος* (*System testing*) επιβεβαιώνουν ότι το σύστημα ως ολότητα ανταποκρίνεται στις προδιαγραφές του.
d. Οι *Δοκιμασίες Αποδοχής* (*Acceptance testing*) ελέγχουν αν το σύστημα ικανοποιεί τις προδιαγραφές που τέθηκαν στο συμβόλαιο από τη πλευρά του τελικού χρήστη.

Ο έλεγχος λογισμικού σε διαφορετικά επίπεδα του κύκλου ζωής του λογισμικού αποφέρει και διαφορετικά είδη ελαττωμάτων. Για παράδειγμα, οι δοκιμασίες μονάδας είναι πιο πιθανό να βρουν λογικά ελαττώματα του κώδικα παρά ελαττώματα σχεδίου συστήματος (system design defects), τα οποία βρίσκονται κυρίως στις δοκιμασίες συστήματος.

2.2.1. Δοκιμασίες Μονάδας

Σε αυτό το επίπεδο ελέγχονται οι μονάδες λογισμικού ξεχωριστά και σε απομόνωση από το υπόλοιπο σύστημα. Μονάδα θεωρείται το μικρότερο στοιχείο που μπορεί να μεταγλωτιστεί σε γλώσσα μηχανής (*compile*). Ανάλογα με τη γλώσσα προγραμματισμού, μονάδες μπορεί να είναι κλάσεις, συναρτήσεις, μέθοδοι ή ένα *σύνολο ανεξάρτητων κομματιών κώδικα (modules)*. Ο tester στην συγκεκριμένη περίπτωση ελέγχει τα εσωτερικά στοιχεία και τις συμπεριφορές των μονάδων. Οι δοκιμασίες μονάδας είναι το χαμηλότερο επίπεδο ελέγχου και ονομάζεται επίσης και δοκιμασίες προγραμματιστή καθώς συνήθως ο προγραμματιστής είναι αυτός που τις εκτελεί. Σε αντίθετη περίπτωση, ο tester έχει πρόσβαση στον κώδικα και εκτελεί τις δοκιμασίες.

Στις δοκιμασίες μονάδας, οι μονάδες αντικαθίστανται από stubs ή simulators και οι μονάδες που καλούνται μέσα σε αυτές αντικαθίστανται από drivers. *Stub* ονομάζεται μια εφαρμογή λογισμικού που έχει ως αποκλειστικό σκοπό να ελέγξει μια μονάδα. Ενώ, simulator ονομάζεται μια συσκευή, ένα πρόγραμμα υπολογιστή ή ένα σύστημα που χρησιμοποιείται κατά τη διάρκεια του testing και το οποίο συμπεριφέρεται ή λειτουργεί με τον ίδιο τρόπο με το σύστημα ύπο έλεγχο, δηλαδή δέχεται τα ίδια εισερχόμενα δεδομένα και πράγει τα ίδια εξερχόμενα δεδομένα. Ως *driver* ή *test driver* ορίζεται μια μονάδα λογισμικού ή ένα *εργαλείο* (*test tool*) το οποίο αντικαθιστά μια μονάδα που αναλαμβάνει να καλεί μια άλλη μονάδα ή σύστημα.

Ο βασικός στόχος σε αυτό το επίπεδο είναι η εξασφάλιση εγγύησης ότι το αντικείμενο υπό έλεγχο εκτελεί τη λειτουργικότητα του σωστά και εξ ολοκλήρου έτσι όπως ορίζεται από τις προδιαγραφές (*functionality testing*). Δηλαδή, η μονάδα ελέγχεται με μια σειρά περιπτώσεων ελέγχου, όπου κάθε περίπτωση καλύπτει ένα συγκεκριμένο ζευγάρι εισερχόμενων/εξερχόμενων δεδομένων από όλους τους δυνατούς συνδυασμούς. Τα συνήθη ελαττώματα λογισμικού που αποκαλύπτονται μετά από τις *λειτουργικές δοκιμασίες μονάδας* (functionality component test) είναι λάθος υπολογισμοί και λάθος μονοπάτια κώδικα.

Μια άλλη πλευρά που καλύπτει αυτό το επίπεδο είναι οι δοκιμές ανθεκτικότητας (robustness testing) και οι *αρνητικές δοκιμές* (negative tests). Οι δοκιμές ανθεκτικότητας ελέγχουν το βαθμό στον οποίο μία μονάδα ή ένα σύστημα μπορεί να λειτουργήσει σωστά στην περίπτωση μη έγκυρων δεδομένων ή κακών συνθηκών του περιβάλλοντος. Οι αρνητικές δοκιμές στοχεύουν να αναδείξουν ότι μια μονάδα ή ένα σύστημα δεν δουλεύει. Αυτό γίνεται είτε με την *εισαγωγή μη έγκυρων δεδομένων ελέγχου,* τα οποία θα απορριφθούν από τη μονάδα ή το σύστημα (invalid testing), είτε με *δοκιμές εξαιρέσεων συστήματος* (system exceptions). Η *διαχείριση λαθών* (*error/fault tolerance*) και η *διαχείριση εξαιρέσεων* (*exception handling*) συνίσταται στην ικανότητα της μονάδας να μπορεί να συνεχίσει τη λειτουργία της παρά την παρουσία λαθών και να διατηρεί ένα καλό επίπεδο επίδοσης. Είναι πολύ σημαντική διαδικασία και πρέπει να δοκιμάζεται όσο πιο νωρίς γίνεται ακόμα και στα μικρότερα κομμάτια λογισμικού.

Οι δοκιμασίες μονάδας ελέγχουν επίσης και μη λειτουργικά χαρακτηριστικά, που έχουν μεγάλη επίδραση στην ποιότητα της μονάδας και δε μπορούν να δοκιμαστούν σε ανώτερο επίπεδο ελέγχου. Τέτοια χαρακτηριστικά είναι η αποδοτικότητα και η συντηρησιμότητα. Οι δοκιμές αποδοτικότητας αποφαίνονται για το πόσο αποδοτικά η μονάδα χρησιμοποιεί τους πόρους του υπολογιστή, όπως χρήση μνήμης, *χρόνος απόκρισης* (*computing time*), χρόνος πρόσβασης σε δίσκο ή δίκτυο και πόσο χρόνο χρειάζονται οι συναρτήσεις και οι αλγόριθμοι της μονάδας να εκτελεστούν. Συνήθως, η αποδοτικότητα ελέγχεται σε υψίστης σημασίας μονάδες του συστήματος ή εάν το προσδιορίζουν οι απαιτήσεις του πελάτη. Οι δοκιμές συντηρησιμότητας μετρούν πόσο εύκολο ή δύσκολο είναι να αλλάξεις το πρόγραμμα ή να συνεχίσεις να το αναπτύσσεις. Συνεπώς σημαντικοί παράγοντες για τη συντηρησιμότητα είναι η δομή του κώδικα, το *δομοστοιχείωση* (*modularity*), τα σχόλια στον κώδικα, η υπακοή στα πρότυπα, η κατανόηση και η επικαιροποίηση των σχετικών εγγράφων κ.ά. Αυτά τα χαρακτηριστικά δε μπορούν να ελεγχθούν με δυναμικό τρόπο, γι'αυτό χρησιμοποιούνται στατικοί μέθοδοι.

Όπως ήδη αναφέραμε είτε ο προγραμματιστής είτε ο tester έχει πρόσβαση στον κώδικα. Η μελέτη και η κατανόηση της εσωτερικής δομής της μονάδας, των συναρτήσεως και των μεταβλητών του κώδικα είναι ιδιαίτερα χρήσιμη για το σχεδιασμό περιπτώσεων ελέγχων. Συνεπώς, οι δοκιμασίες μονάδας απαιτούν απο τον tester να έχει πρόσβαση στον κώδικα (ανήκουν στο white box έλεγχο λογισμικού), παρόλα αυτά μπορούν να γίνουν και με black box έλεγχο. Τέλος, σημαντική απόφαση στο επίπεδο αυτό είναι σε ποιές μονάδες θα πρέπει να γίνουν οι δοκιμασίες, σε μικρές βασικές μονάδες ή σε μεγαλύτερες μονάδες που προκύπτουν από συνενώσεις άλλων μονάδων.

2.2.2. Δοκιμασίες Συνένωσης

Συνένωση είναι η σύνθεση μίας ομάδας μονάδων για τη δημιουργία μιας μεγαλύτερης δομημένης μονάδας ή ενός υποσυστήματος. Ο στόχος των δοκιμασιών συνένωσης είναι να εκθέσουν τα ελαττώματα των διεπαφών (interfaces) και των διασυνδέσεων μεταξύ των συνενωμένων μονάδων, η εύρεση προβλημάτων συνεργασίας μεταξύ των μονάδων και η απομόνωση των αιτιών.

Για το επίπεδο αυτό χρησιμοποιούνται test drivers για τις δοκιμασίες, τα οποία ελέγχουν τα δεδομένα στο αντικείμενο υπό έλεγχο και λαμβάνουν τα αποτελέσματα. Συνήθως, αν οι δοκιμασίες μονάδας είναι καλά οργανωμένες , οι testers μπορούν να επαναχρησιμοποιήσουν τα ίδια test drivers. Σε αντίθετη περίπτωση επιβάλλεται η δημιουργία, αλλαγή ή επιδιόρθωση του περιβάλλοντος ελέγχου. Κατά τη διάρκεια των δοκιμασιών συνένωσης επιπλέον εργαλείων ελέγχου, τα επονομαζόμενα monitors, είναι απαραίτητα. *Monitor* είναι ένα εργαλείο λογισμικού ή μια *συσκευή τεχνικού εξοπλισμού (hardware device)* η οποία τρέχει ταυτόχρονα με τη μονάδα ή το σύστημα υπό έλεγχο και επιτηρεί, καταγράφει και αναλύει τη συμπεριφορά της μονάδας ή του συστήματος και την κίνηση μεταξύ δύο ή παραπάνω συστημάτων.

Σφάλματα και προβλήματα σε αυτό το επίπεδο μπορούν να βρεθούν μόνο με δυναμικό τρόπο. Επιπρόσθετα του λειτουργικού ελέγχου μπορούν να εκτελεσθούν μη λειτουργικές δοκιμές, όπως δοκιμές επίδοσης (performance testing) και η ικανότητας των διεπαφών. Μερικές φορές, οι δοκιμασίες μονάδας μπορούν να παραληφθούν, όμως το κόστος είναι λιγότερα σφάλματα και μεγαλύτερη δυσκολία στη διάγνωση των ελαττωμάτων. Έτσι, ο συνδυασμός δοκιμασιών μονάδας και συνένωσης είναι συχνά πιο αποτελεσματικός και αποδοτικός.

Επειδή όμως οι μονάδες υλοποιούνται σε διαφορετικό χρόνο, πρέπει να υπάρχει συγκεκριμένος σχεδιασμός της στρατηγικής συνένωσης έτσι ώστε να επιτευχθεί η πιο αποδοτική συνένωση. Η στρατηγική αυτή πρέπει να βελτιστοποιεί δύο παράγοντες: το χρόνο συνένωσης και το κόστος του περιβάλλοντος ελέγχου, λαμβάνοντας πάντα υπόψη την αρχιτεκτονική του συστήματος, το *σχέδιο έργου* (project plan) και το *σχέδιο ελέγχου* (test plan).

Ενδεικτικά παρουσιάζονται πέντε στρατηγικές δοκιμασιών συνένωσης. Στην πράξη επιλέγεται συνήθως ένας συνδυασμός αυτών:

a. Top-down συνένωση

   Το test ξεκινάει με την *ανώτερη* (*top level*) μονάδα του συστήματος, η οποία καλεί άλλες μονάδες αλλά δεν καλεί τον εαυτό της, και στη συνέχεια προχωρά με τις *κατώτερες μονάδες* (*low level*) μέχρι την ολοκλήρωση του. Stubs αντικαθιστούν όλες τις κατώτερες μονάδες, ενώ test drivers δεν χρησιμοποιούνται. Προτέρημα της στρατηγικής αυτής είναι η ευκολότερη εύρεση

χαμένου branch link, ενώ βασικό μειονέκτημα της είναι το αυστηρά ιεραρχικό πρόγραμμα συστήματος που απαιτείται χωρίς να συναντάται εύκολα.

b. Bottom-up συνένωση

Η στρατηγική αυτή ξεκινά με τις κατώτερες βασικές μονάδες οι οποίες δεν καλούν άλλες μονάδες. Μεγαλύτερα υποσυστήματα συναθροίζονται βήμα-βήμα μετά από κάθε δοκιμασία συνένωσης. Δε χρησιμοποιούνται stubs αλλά οι ανώτερες μονάδες προσομοιώνονται από test drivers. Αυστηρά ιεραρχικό πρόγραμμα συστήματος απαιτείται και από αυτή τη στρατηγική αλλά δε συναντάται εύκολα. Βασικό της προτέρημα είναι ότι τα ελαττώματα λογισμικού βρίσκονται εύκολα.[3]

c. Ad-hoc συνένωση

Οι μονάδες συνενώνονται για τις δοκιμασίες με τη σειρά με την οποία αναπτύχθηκαν. Αυτό γλιτώνει χρόνο, όμως stubs και test drivers είναι απαραίτητα.

d. Backbone συνένωση

Μια «ραχοκοκαλιά» χτίζεται, πάνω στην οποία συνενώνονται διαδοχικά οι μονάδες με τυχαία σειρά. Απαιτείται μία labor intensive skeleton.

e. Big-bang συνένωση

Με αυτή τη προσέγγιση, οι μονάδες συνενώνονται σε ένα βήμα, όταν όλα τα στοιχεία έχουν αναπτυχθεί, για να αποτελέσουν ένα ολοκληρωμένο σύστημα ή ένα μεγάλο μέρος του συστήματος. Η στρατηγική αυτή συμφέρει από κόστος χρόνου, αλλά υπάρχει μεγάλη επικινδυνότητα αποτυχίας καθώς βασίζεται στις δοκιμασίες μονάδας και περιμένει ελάχιστα προβλήματα σε αυτές. Σε περίπτωση που οι δοκιμασίες μονάδες έχουν γίνει για όλες τις μονάδες με σωστά αποτελέσματα, τότε σε αυτό το επίπεδο μπορούν να ανιχνευτούν σημαντικά προβλήματα που προκαλούνται από τη διασύνδεση των μονάδων με το περιβάλλον.

2.2.3. Δοκιμασίες Συστήματος

Οι δοκιμασίες συστήματος ελέγχουν αν το συνενωμένο σύστημα πληρεί τις απαιτήσεις και τις προδιαγραφές. Αυτό σημαίνει ότι ελέγχουν το σύστημα από τη πλευρά του πελάτη ή του μελλοντικού χρήστη, σε αντίθεση με τα κατώτερα επίπεδα ελέγχου που το εξέταζαν τεχνικά. Οι δοκιμασίες συστήματος αντιμετωπίζουν το

[3] Sandwich testing λέγεται η στρατηγική δοκιμασιών συνένωσης που συνδυάζει τις δυο στρατηγικές Top-down και Bottom-up.

προϊόν ή το σύστημα σαν ολότητα και έτσι πολλές λειτουργίες και χαρακτηριστικά συστήματος είναι ορατά μόνο σε αυτό το επίπεδο.

Οι δοκιμασίες που εκτελούνται σε αυτό το επίπεδο βασίζονται πάνω στις απαιτήσεις, προδιαγραφές, *διαδικασίες* (*business processes*), *περιπτώσεις χρήσης* (*use cases)* ή σε διάφορες συνθήκες με τις συμπεριφορές και τους πόρους του συστήματος, τις αλληλεπιδράσεις μεταξύ των διαφορετικών συστημάτων και τις αναλύσεις επικινδυνότητας. Γίνεται έλεγχος τόσο λειτουργικών όσο και μη λειτουργικών απαιτήσεων (functional και non-functional requirements). Τα συνήθη μη λειτουργικά χαρακτηριστικά που εξετάζονται είναι η οι επιδόσεις και η αξιοπιστία, ενώ τα λειτουργικά εξετάζονται συνήθως με τη μέθοδο black box.

Οι δοκιμασίες λαμβάνουν χώρα σε ένα ξεχωριστό ελεγχόμενο *περιβάλλον ελέγχου* (*test environment*), όμοιο με αυτό της παραγωγής (δηλαδή με ίδιο τεχνικό εξοπλισμό, λειτουργικό σύστημα, δίκτυα, εξωτερικά συστήματα κ.ά.). Είναι απολύτως απαραίτητο οι δοκιμασίες να γίνουν σε ξεχωριστό περιβάλλον από αυτό του πελάτη, γιατί αλλιώς μπορούν να προκληθούν ζημιές. Τέλος, αυτό αποτελεί το τελευταίο επίπεδο ελέγχου που μπορεί να εκτελέσει ο ίδιος ο προγραμματιστής ενώ συνήθως γίνεται από μια ανεξάρτητη *ομάδα ελέγχου (testing team).*

2.2.4. Δοκιμασίες Αποδοχής

Όταν η διαδικασία ανάπτυξης λογισμικού έχει τελειώσει και οι δοκιμασίες συστήματος έχουν αποφέρει ελαττώματα τα οποία έχουν διορθωθεί, το προϊόν ή το σύστημα είναι έτοιμο για το τελευταίο επίπεδο ελέγχου, τις δοκιμασίες αποδοχής. Πολλοί οργανισμοί ονομάζουν τον έλεγχο πριν και μετά τη μεταφορά στο περιβάλλον του πελάτη Factory *Acceptance Testing* (FAT) και *Site Acceptance Testing* (SAT).

Υπεύθυνος του επιπέδου αυτού είναι συνήθως μια ανεξάρτητη ομάδα ελέγχου, αλλά πολλές φορές, ανάλογα το προϊόν, σε αυτό το επίπεδο συμμετέχει και ο χρήστης ή ο πελάτης. Οι δοκιμασίες πρέπει να εκτελούνται σε ένα περιβάλλον ελέγχου *όσο περισσότερο κοντά στην παραγωγή* ("as-if production"). Στόχος τους επιπέδου αυτού είναι να εδραιωθεί η εμπιστοσύνη προς το σύστημα, σε μέρος του συστήματος ή σε συγκεκριμένα μη λειτουργικά χαρακτηριστικά όπως η χρησικανότητα του συστήματος. Η εύρεση *ελαττωμάτων* (*defects*) δεν είναι ο βασικός στόχος του ελέγχου, αντίθετα στοχεύει στην επικύρωση και αξιολόγηση του συστήματος για το κατα πόσον είναι έτοιμο για *ανάθεση στο server (deployment)* του πελάτη και χρήση.

Γνωστές κατηγορίες δοκιμασιών αποδοχής είναι:

a. *Δοκιμασίες αποδοχής του χρήστη* (*User acceptance test*)

Αξιολογείται η ευκολία χρήσης του προϊόντος ή του συστήματος από τους τελικούς χρήστες. Προτείνεται όταν ο πελάτης και ο χρήστης είναι διαφορετικά πρόσωπα και

πρέπει να λαμβάνονται υπόψη όλες οι κατηγορίες χρηστών με ανάλογες δοκιμασίες για την κάθε κατηγορία. Συνήθως εκτελείται από τους χρήστες ή από διαχειριστές εφαρμογών (application managers).

b. *Λειτουργικές δοκιμασίες αποδοχής* (*Operational acceptance test*)

Διασφαλίζουν την αποδοχή του συστήματος από το διαχειριστή συστήματος (system administrator). Εξετάζονται παράγοντες όπως backup/restore, ανάκτηση σε περίπτωση καταστροφής (disaster recovery), εργασίες διατήρησης και περιοδικοί έλεγχοι για αδυναμίες ασφάλειας.

c. *Διαδικασίες δοκιμασίας κατά το συμβόλαιο* (Contract acceptance testing)

Ελέγχεται αν οι όροι του συμβολαίου έχουν τηρηθεί. Ειδικά κριτήρια και test cases σχεδιάζονται με βάση το συμβόλαιο.

d. *Διαδικασίες αποδοχής συμβατότητας* (*Compliance acceptance testing*)

Το προϊόν ή το σύστημα δοκιμάζεται αν τηρεί τα τους κανονισμούς , τους νόμους και τους κανόνες ασφάλειας.

e. *Alpha και Beta δοκιμασίες αποδοχής* (*Alpha and Beta testing*)

Αν το προϊόν είναι *ευρέως χρήσεως* (*COTS: customers off-the-shelf software*) δεν είναι πάντα εφικτό να εκτελεστούν οι δοκιμασίες αποδοχής. Τότε επιστρατεύονται οι ίδιοι οι χρήστες για να παρέχουν γνώμες, σχόλια και εντυπώσεις. Alpha testing ονομάζονται οι δοκιμασίες που εκτελούνται στο περιβάλλον του παραγωγού/προγραμματιστή ενώ Beta testing είναι οι δοκιμασίες που γίνονται στο περιβάλλον του πελάτη.

2.3 Γενικοί τύποι ελέγχου [1] [2] [3]

Οι έλεγχοι μπορούν να διακριθούν στους παρακάτω τύπους:

2.3.1 Λειτουργικός έλεγχος

Λειρουργικός αποκαλείται κάθε έλεγχος που επαληθεύει την αναμενόμενη συμπεριφορά των δεδομένων εισόδου-εξόδου. Για τον σχεδιασμό των σχετικών περιπτώσεων ελέγχου, ο tester χρησιμοποιεί κυρίως την black box μέθοδο (black box method).

Στον *έλεγχο βάσει απαιτήσεων (requirements-based testing)*, ως βάση ελέγχου χρησιμοποιούνται οι *απαιτήσεις* (requirements). Στις προδιαγραφές ελέγχου, σχεδιάζεται και τεκμηριώνεται τουλάχιστον μια περίπτωση ελέγχου, ενώ συνήθως χρειάζονται περισσότερες από μία περιπτώσεις για να ελεγχθεί μια λειτουργική απαίτηση. Αν ολες οι ορισμένες περιπτώσεις ελέγχου έχουν τρέξει χωρίς κάποια αποτυχία, τότε η ανάλογη λειτουργικότητα θεωρείται επικυρωμένη (validated functionality). Αυτός ο τύπος ελέγχου χρησιμοποιείται κυρίως για τις δοκιμασίες συστήματος και αποδοχής.
Στον *έλεγχο βάσει επιχειρησιακών διαδικασιών (business- process-based testing)*, ο σχεδιασμός ελέγχου βασίζεται στην περιγραφή και στη γνώση των διαδικασιών. Μετά από αυτό, οι περιπτώσεις και τα σενάρια ελέγχου κατασκευάζονται βάσει αυτών, ενώ η προτεραιότητα ιεραρχείται βάσει της συχνότητας και της σημαντικότητας της διαδικασίας.
Οι διαφορά των δύο τύπων ελέγχου έγκειται στο γεγονός ότι ο έλεγχος βάσει απαιτήσεων εστιάζει σε λειτουργίες ενιαίου συστήματος, ενώ ο έλεγχος βάσει επιχειρησιακών διαδικασιών επικεντρώνεται στη συνολική, πολυβηματική διαδικασία.

2.3.2 Μη λειτουργικός έλεγχος

Οι μη λειτουργικές απαιτήσεις περιγράφουν τα γνωρίσματα ενός συστήματος ως ενιαίου συνόλου, καθώς και το γνώρισμα της λειτουργικής συμπεριφοράς. Χαρακτηριστικά των απαιτήσεων αυτών αποτελούν η αξιοπιστία, η χρησικανότητακαι η αποτελεσματικότητα. Τα παρακάτω μη λειτουργικά χαρακτηριστικά ενός συστήματος πρέπει να λαμβάνονται υπόψη στους ελέγχους, συνήθως στις δοκιμασίες του συστήματος:

- *Έλεγχος φόρτου (Load testing)*
- *Έλεγχος επίδοσης (Performance testing)*
- *Έλεγχος όγκου (Volume testing)*
- *Έλεγχος έντασης (Stress testing)*
- *Έλεγχος ασφάλειας (Security testing)*
- *Έλεγχος σταθερότητας (Stability testing)*
- *Έλεγχος ανθεκτικότητας (Robustness testing)*
- *Έλεγχος συμβατότητας και μετατροπής δεδομένων (Testing for compatibility and data conversion)*
- *Έλεγχος διαφορετικών συνθέσεων (Testing of different configurations)*
- *Έλεγχος χρησικανότητας (Usability testing)*
- *Έλεγχος για δυνατότητα τεκμηρίωσης (Checking for the documentation)*
- *Έλεγχος συντηρησιμότητας (Checking for maintainability)*

Το κύριο πρόβλημα των μη λειτουργικών χαρακτηριστικών είναι ότι συνήθως προκύπτουν από ανακριβείς απαιτήσεις και συνεπώς δεν είναι ελέγξιμα. Συνεπώς, σκοπός του tester αποτελεί η εξασφάλιση ότι οι μη λειτουργικές απαιτήσεις είναι μετρήσιμες και η δημιουργία κατάλληλων μετρικών.

2.3.3 Έλεγχος δομών λογισμικού ή δομικός έλεγχος

Οι δομικές τεχνικές (*structural testing*) χρησιμοποιούν πληροφορίες απ' την εσωτερική δομή κι αρχιτεκτονική του κώδικα του αντκειμένου υπό έλεγχο (white box έλεγχος). Μπορούν να χρησιμοποιηθούν δηλώσεις, ιεραρχία, δομές επιλογών μενού και αφηρημένα μοντέλα του λογισμικού, όπως το μοντέλο ροής διαδικασιών ή το μοντέλο μεταβατικής κατάστασης. Σκοπός του δομικού ελέγχου είναι η κάλυψη όλων των δομικών στοιχείων και συνήθως χρησιμοποιείται στις δοκιμασίες μονάδας και συνένωσης.

2.3.4 Έλεγχος αλλαγών και έλεγχος παλινδρόμισης

Αν το υπάρχον λογισμικό τροποποιηθεί, τα αλλαγμένα μέρη πρέπει να επανελεχθούν και οι υπάρχοντες περιπτώσεις ελέγχου πρέπει να επαναληφθούν μαζί με τα καινούρια μέρη που θα ελεχθούν πρώτη φορά. Ο *έλεγχος παλινδρόμισης (regression testing)* είναι ένας επανέλεγχος ενός ήδη ελεγμένου προγράμματος, που έχει υποστεί τροποποιήσεις, με σκοπό να εξασφαλιστεί η μη εισαγωγή νέων ή η αποκάλυψη υπαρχόντων σφαλμάτων ως αποτέλεσμα των αλλαγών. Ο έλεγχος παλινδρόμισης μπορεί να πραγματοποιηθεί σε όλα τα επίπεδα ελέγχου. Χρησιμοποιείται στον λειτουργικό, μη λειτουργικό και δομικό έλεγχο και ανάλογα με τον όγκο του ελέγχου μπορεί να διακριθεί σε:

- *Επανέλεγχος ελαττωμάτων και έλεγχος διάταξης (Defect retest and configuration testing)*
- *Έλεγχος τροποποιημένης λειτουργικότητας (Testing of altered functionality)*
- *Έλεγχος νέας λειτουργικότητας (Testing of new functionality)*
- *Έλεγχος πλήρους παλινδρόμισης (Complete regression test)*

Επειδή οι τροποποιήσεις του λογισμικού μπορούν να δημιουργήσουν παράπλευρες απώλειες αυθαίρετα σε άλλα μέρη του συστήματος, χρειάζεται ένας έλεγχος πλήρους παλινδρόμισης. Επειδή όμως αυτό είναι συνήθως χρονοβόρο και δαπανηρό, επιλέγεται ένα σύνολο περιπτώσεων παλινδρόμισης, ανάλογα με τις προτεραιότητες του προϊόντος. Παραλείπονται έτσι ορισμένες διαφοροποιήσεις και περιορίζονται οι έλεγχοι σε συγκεριμένες διατάξεις, υποσυστήματα και επίπεδα ελέγχου.

2.3.5 Άλλοι τύποι ελέγχων

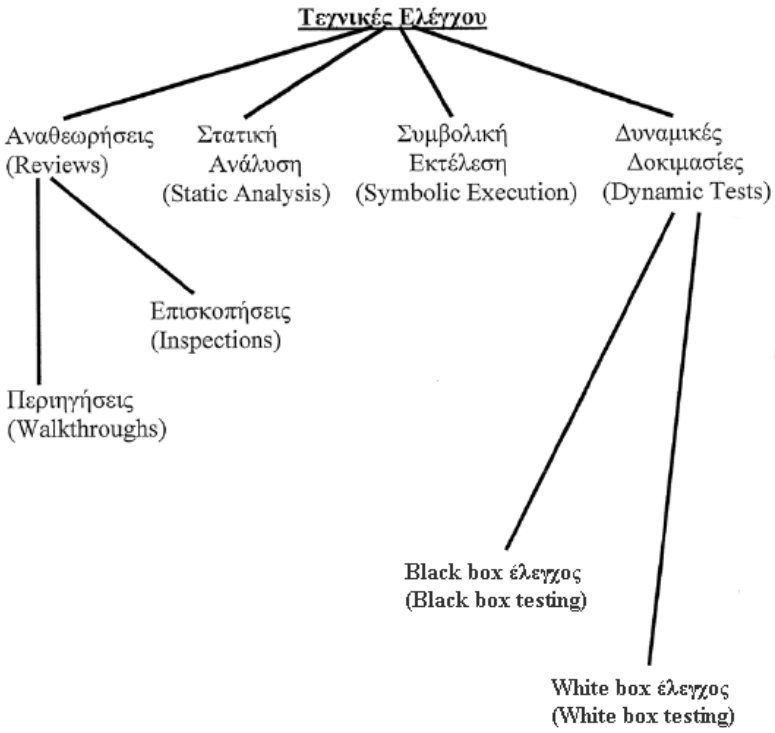
Εκτός από τους παραπάνω τύπους ελέγχου, υπάρχουν και άλλοι δύο σημαντικοί έλεγχοι που δεν υπόκεινται στις υπάρχουσες κατηγοριοποιήσεις.

Με κριτήριο το χρονικό σημείο εκτέλεσης του ελέγχου ορίζουμε τον *έλεγχο καπνού (Smoke or Sanity testing)* ως τον προκαταρκτικό έλεγχο για την ανίχνευση σοβαρών σφαλμάτων που οδηγούν στην απόρριψη της νέας έκδοσης του λογισμικού. Αποτελείται από ένα υποσύνολο περιπτώσεων ελέγχου που καλύπτουν τις πιο σημαντικές λειτουργικότητες του προϊόντος.

Ο διερευνητικός έλεγχος (Explanatory testing) είναι ένα είδος προσέγγισης του ελέγχου λογισμικού κατά τον οποίο παίρνουμε πληροφορίες και γνώση για το σύστημα υπό έλεγχο, σχεδιάζοντας μικρούς ελέγχους και εκτελώντας τους ταυτόχρονα. Αυτός ο σύντομος κύκλος ελέγχου συνεχίζεται μέχρι ο tester να αναγνωρίσει όλες τις δυνατότητες και τους περιορισμούς του συστήματος. Τα στάδια του διερευνητικού ελέγχου είναι σχεδιασμός του ελέγχου, εκτέλεση και παρατήρηση για εξαγωγή γνώσης. Ο σχεδιασμός ελέγχου περιέχει την αναγνώριση των περιπτώσεων ελέγχου χωρίς την καταγραφή τους σε επίσημα σενάρια ελέγχου. Ο σχεδιασμός αφορά ελέγχους για δεδομένα ή για καταστάσεις και βασίζεται σε όλες τις παραδοσιακές μεθόδους. Αφού προσδιοριστεί ένα μικρό σύνολο περιπτώσεων ελέγχου με βάση ένα συγκεκριμένο κριτήριο, στη συνέχεια αυτό εκτελείται. Εδώ έγκειται η διαφορά του διερευνητικού ελέγχου, δε χρειάζεται να σχεδιαστεί ένα μεγάλο σύνολο περιπτώσεων για να ελέγθει όλη την λειτουργικότητα σε κάποιο μελλοντικό έλεγχο. Κατά την εκτέλεση των περιπτώσεων ελέγχου παρατηρούμε την εφαρμογή για το τι κάνει και τι δεν κάνει. Ανακαλύπτουμε πως λειτουργεί, τις δυνατότητες της, τις ιδιομορφίες της και τους περιορισμούς της. Οι πληροφορίες της διερεύνησης προσφέρουν γνώση για το αντικείμενο υπό έλεγχο.

2.4 Τεχνικές ελέγχου [1] [2]

Υπάρχουν πολλές κατηγορίες *τεχνικών ελέγχου (test techniques),* η καθεμία ειδικεύεται στην εξεύρεση συγκεκριμένων ειδών ελαττωμάτων, με διαφορετικά πλεονεκτήματα και μειονεκτήματα. Γενικά, οι τεχνικές ελέγχου λογισμικού χωρίζονται σε δύο βασικές διαδικασίες, στατικές και δυναμικές.



Σχήμα 2.2: Κατηγοριοποίηση των τεχνικών ελέγχου λογισμικού.

2.4.1 Στατικές τεχνικές

Οι στατικές τεχνικές δεν εκτελούν το αντικείμενο υπό έλεγχο και χρησιμοποιούνται επιτυχώς για τη βελτίωση ποιότητας της διαδικασίας ελέγχου λογισμικού, είτε ελέγχοντας επίσημα έγγραφα είτε τον ίδιο τον κώδικα. Περιλαμβάνουν δύο βασικές κατηγορίες, τις αναθεωρήσεις και τη στατική ανάλυση. Η συμβολική εκτέλεση του κώδικα μπορεί να θεωρηθεί στατική μέθοδος, παρόλο που εκτελείται ο κώδικας.

2.4.1.1 Αναθεωρήσεις

Οι *αναθεωρήσεις (reviews)* περιλαμβάνουν δυο τρόπους ελέγχου, τις *περιηγήσεις (walkthroughs)* και τις *επισκοπήσεις (inspections).* Και οι δύο τρόποι ασχολούνται με την προσπάθεια ανεύρεσης των ελαττωμάτων διατρέχοντας τον κώδικα με την απλή ανάγνωση. Όσο και αν φαίνεται απλοϊκή μέθοδος, έχει παρατηρηθεί ότι πάρα πολλά σφάλματα έχουν ανιχνευτεί με απλή ανάγνωση του κώδικα και σε πολύ αρχικό στάδιο. Η διαφορά μεταξύ περιηγήσεων και επισκοπήσεων έγκειται στη διαδικασία

που ακολουθείται (ποιος είναι υπεύθυνος, πως συνέρχονται οι ομάδες ελέγχου για συζήτηση των σφαλμάτων κλπ.).

Οι αναθεωρήσεις χρησιμοποιούνται επίσης συχνά στον έλεγχο επίσημων εγγράφων, όπως του σχέδιου συστήματος, *λειτουργικών προδιαγραφών* (*functional specifications*), *προδιαγραφών απαιτήσεων* (*requirement specifications*), περιπτώσεων χρήσης, σχέδιου ελέγχου, περιπτώσεων ελέγχων κ.ά.

2.4.1.2 Στατική ανάλυση

Η στατική ανάλυση (static analysis) χρησιμοποιείται για τον έλεγχο των προδιαγραφών απαιτήσεων, του σχεδίου λογισμικού και κυρίως του κώδικα. Ο έλεγχος γίνεται χωρίς την εκτέλεση του προγράμματος με δεδομένα. Η τεχνική εφαρμόζεται είτε χειρονακτικά είτε με την χρήση ειδικών αυτόματων εργαλείων, των *στατικών αναλυτών* (*static analyzers*).

Δε μπορούν όλα τα ελαττώματα λογισμικού να ανιχνευτούν με αυτή τη τεχνική, διότι μερικά χρειάζονται την εκτέλεση του κώδικα. Όμως πολλά προβλήματα μπορούν να αναγνωριστούν στα πλαίσια της στατικής ανάλυσης όπως:

a. Συντακτικά λάθη του κώδικα

Ο compiler είναι ένα εργαλείο στατικής ανάλυσης για αναγνώριση συντακτικών λαθών της γλώσσας προγραμματισμού, ο οποίος επιπλέον δημιουργεί μια λίστα με όλα τα διαφορετικά στοιχεία του κώδικα, ελέγχει αν είναι σωστοί οι τύποι των δεδομένων, αναγνωρίζει μη δηλωμένες μεταβλητές ή κώδικα που δε χρησιμοποιείται ποτέ, ελέγχει τα άνω και κάτω όρια των πεδίων κ.ά.

b. Απόκλιση από τα πρότυπα

Σε αυτή την φάση αναδιαμορφώνεται ο κώδικας για σωστή ταξινόμηση με τέτοιο τρόπο ώστε να τηρούνται οι προδιαγραφές και τα πρότυπα της γλώσσας προγραμματισμού και συγχρόνως να είναι ευκολότερη η ανάλυση του κώδικα.

c. Data flow ανωμαλίες

Με τη data flow ανάλυση ελέγχονται οι ανωμαλίες του κώδικα που ενδεχομένως να οδηγήσουν σε σφάλματα, χωρίς αυτό να είναι απαραίτητο. Τα ελαττώματα σε αυτές τις περιπτώσεις εμφανίζονται συνήθως στα δεδομένα που χρησιμοποιούνται στα διάφορα μονοπάτια του κώδικα. Έτσι, όλες οι μεταβλητές εξετάζονται λεπτομερώς κατά την ανάλυση τους.

Οι μεταβλητές χωρίζονται ανάλογα με τη χρήση ή την κατάσταση τους σε:

- *(d) Ορισμένες* (*defined*): Οι μεταβλητές έχουν λάβει τιμή.

- *(r) Αναφερόμενες (referenced*): Η τιμή της μεταβλητής διαβάζεται και/ή χρησιμοποιείται.
- *(u) Μη ορισμένες* (undefined): Οι μεταβλητές δεν έχουν λάβει ορισμένη τιμή.

Επομένως, μπορούμε να ορίσουμε τρεις τύπους data flow ανωμαλιών:

- *ur-ανωμαλία*: Μία μη ορισμένη τιμή (u) της μεταβλητής διαβάζεται στο μονοπάτι (r)
- *du-ανωμαλία*: Η μεταβλητή λαμβάνει τιμή (d) η οποία γίνεται μη έγκυρη/μη ορισμένη(u) χωρίς να έχει χρησιμοποιηθεί ακόμα.
- *dd-ανωμαλία*: Η μεταβλητή λαμβάνει τιμή για δεύτερη φορά (d) αλλά η πρώτη φορά δεν έχει χρησιμοποιηθεί ακόμα (d).

Οι data flow ανωμαλίες δεν είναι εύκολα ορατές, αλλά επίσης δεν οδηγούν πάντα σε σφάλματα.

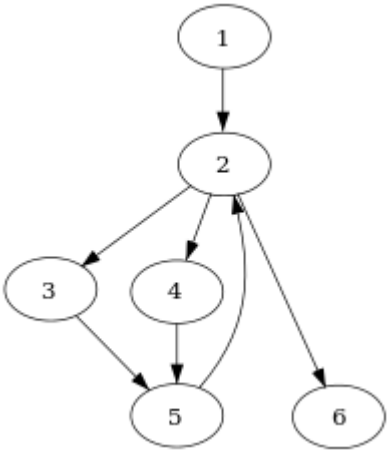
d. Control flow ανωμαλίες

Ο γράφος control flow είναι μια ακολουθία γεγονότων (μονοπάτι) κατά την εκτέλεση του κώδικα. Είναι ένας κατευθυνόμενος γράφος, όπου οι δηλώσεις (statements) του προγράμματος αναπαριστόνται με κόμβους. Χάρη στην σαφήνεια του γράφου, πιθανές ανωμαλίες είναι εύκολα αναγνωρίσιμες. Τέτοιες ανωμαλίες συνήθως είναι έξοδοι από βρόχο, κομμάτι κώδικα που έχει πολλές εξόδους κ.ά. Αν ο κώδικας είναι πολύ περίπλοκος, με τη βοήθεια του control flow γράφου επανεξετάζεται και απλοποιείται. Μερικοί στατικοί αναλυτές αυτής της μεθόδου πιθανόν να αναφέρουν και το συνολικό αριθμό των μονοπατιών που ενδεχομένως έχει το πρόγραμμα προς ανάλυση.

Τέλος ένας άλλος στόχος της στατικής ανάλυσης είναι η μέτρηση ποιοτικών χαρακτηριστικών μέσω των μετρικών. Οι μετρικές βοηθούν στο να μετρούνται ποσοτικά διάφοροι παράγοντες του λογισμικού. Για παράδειγμα, η *μετρική κυκλωματικής πολυπλοκότητας* (*cyclomatic complexity metric*) είναι μια μετρική που μετρά την πολυπλοκότητα της δομής του κώδικα μέσω του control flow γράφου. Υπολογίζει δηλαδή τον αριθμό των ανεξάρτητων μονοπατιών με την βοήθεια του τύπου:

$$v(G) = e - n + 2$$

όπου *e* ο αριθμός των ακμών και *n* ο αριθμός των κόμβων.



Σχήμα 2.3: Control flow γράφος αναπαράστασης κομματιού κώδικα

Ένα κομμάτι κώδικα αναπαριστάται από τον control flow γράφο της εικόνας, που ξεκινά από τον κόμβο 1 (αρχική κατάσταση) για να καταλήξει στον κόμβο 6 (τελική κατάσταση). Η μετρική κυκλωματικής πολυπλοκότητας υπολογίζεται ως εξής: e = 7 και n = 6 οπότε

$$v(G) = e - n + 2 = 7 - 6 + 2 = 3$$ ανεξάρτητα μονοπάτια.

Όσο μεγαλύτερος είναι ο αριθμός κυκλωματικής περιπλοκότητας, τόσο πιο περίπλοκο και δύσκολο είναι να καταλάβεις ένα κομμάτι κώδικα. Συνεπώς, η μετρική αυτή παρέχει σημαντική πληροφορία για το μέγεθος του ελέγχου που θα ακολουθήσει και χρησιμοποιείται κυρίως για να εκτιμήσει την *ελεγξιμότητα* (*testability*) και τη συντηρησιμότητα του συστήματος.

2.4.1.3 Συμβολική εκτέλεση κώδικα

Συμβολική εκτέλεση κώδικα (*symbolic execution*) είναι η εκτέλεση ολόκληρου του κώδικα (όλων των γραμμών) με συμβολικές τιμές, διαφορετικές από τις πραγματικές τιμές εισόδου τις οποίες απαιτεί η κανονική εκτέλεση του.

2.4.2 Δυναμικές τεχνικές [1] [2] [3]

Οι δυναμικές τεχνικές ελέγχου παρέχουν στο αντικείμενο υπό έλεγχο δεδομένα εισόδου, στη συνέχεια αυτό εκτελείται και τέλος παίρνουμε κάποια δεδομένα εξόδου. Αυτό σημαίνει ότι το αντικείμενο πρέπει να είναι εκτελέσιμο, γι'αυτό στα κατώτερα επίπεδα ελέγχου, στις δοκιμασίες μονάδας και συνένωσης, το αντικείμενο τρέχει πάνω σε ένα «κρεβάτι ελέγχου» *το οποίο* αποτελείται από stubs και test drivers για

την προσομοίωση κομματιών του προγράμματυος. Οι δυναμικές τεχνικές χωρίζονται σε δύο βασικές κατηγορίες, στο *black box έλεγχο* και τη *white box έλεγχο*. Υπάρχει επίσης και ο *grey box έλεγχος,* που συνδυάζει στοιχεία και των δύο μεθόδων.

Για να αποφασιστεί ποια μέθοδος ελέγχου θα ακολουθηθεί και ποιες περιπτώσεις ελέγχου θα επιλεχθούν, ο tester πρέπει να επιλέξει μια συστηματική προσέγγιση σύμφωνα με τα παρακάτω κριτήρια:

- Να οριστούν οι συνθήκες, οι προϋποθέσεις και οι στόχοι του ελέγχου.
- Να αποσαφηνιστούν οι περιπτώσεις ελέγχου και να προσδιοριστούν τα αναμενόμενα αποτελέσματα και συμπεριφορές.
- Να προσδιοριστούν με ποιο τρόπο θα εκτελεστούν οι δοκιμασίες.

2.4.2.1 Black box έλεγχος

Ο black box έλεγχος εξετάζει τη λειτουργικότητα του προϊόντος ή του συστήματος όπως αυτή απορρέει από τις προδιαγραφές απαιτήσεων, αντιμετωπίζονταςτο αντικείμενο υπό έλεγχο σαν ένα μαύρο κουτί αδιαφορώντας για τις εσωτερικές του λειτουργίες και δομές. Χρησιμοποιείται συχνά στα ανώτερα επίπεδα ελέγχου, όπως στις δοκιμασίες συστήματος και αποδοχής, όμως μπορεί να εφαρμοστεί και στις δοκιμασίες μονάδας. Επίσης, η μέθοδος αυτή χρησιμοποιείται συχνά και στο σχεδιασμό σεναρίων ελέγχου όταν ο κώδικας και η υλοποίηση δεν έχει ξεκινήσει ακόμα.

Υπάρχουν πολλοί μέθοδοι σύμφωνα με τον black box έλεγχο, στην παρούσα εργασία ωστόσο θα εξεταστούν οι εξης από αυτούς:

a. *Διαχωρισμός σε κλάσεις ισοδυναμίας (Equivalence class partitioning)*

   Το πεδίο ορισμού των πιθανών εισερχόμενων δεδομένων χωρίζεται σε κλάσεις ισοδυναμίας, και μία τιμή από κάθε κλάση επιλέγεται για τον έλεγχο. Κλάση ισοδυναμίας ορίζεται μια ομάδα δεδομένων όπου ο έλεγχος τις επεξεργάζεται το ίδιο.
   Ας δούμε λεπτομερώς πως προκύπτουν τα δεδομένα ελέγχου. Πρώτα, πρέπει να οριστεί πλήρως το πεδίο ορισμού και να χωριστεί σε δύο κλάσεις, τις έγκυρες και μη έγκυρες τιμές. Στη συνέχεια, περισσότερος διαχωρισμός σε κλάσεις πρέπει να γίνει σύμφωνα με τις προδιαγραφές και να επιλεγεί μια αντιπροσωπευτική τιμή . Τέλος, ο tester πρέπει να ορίζει τις προϋποθέσεις και τα αναμενόμενα αποτελέσματα για κάθε περίπτωση ελέγχου. Ο διαχωρισμός σε κλάσεις μπορεί να συμβεί και στα εξερχόμενα δεδομένα.

   Για να οριστεί μια περίπτωση ελέγχου, όλες οι αντιπροσωπευτικές τιμές των έγκυρων κλάσεων ισοδυναμίας πρέπει να συνδυαστούν σε μια (*πολλαπλασιαστική μέθοδος*), φτιάχνοντας έτσι μια περίπτωση σενάριο ελέγχου. Ενώ οι

αντιπροσωπευτικές τιμές των μη έγκυρων κλάσεων ισοδυναμίας μπορούν να συνδυαστούν μόνο με τις τιμές των αντίστοιχων έγκυρων κλάσεων (*προσθετική μέθοδος)* δημιουργώντας έτσι μια αρνητική περίπτωση ελέγχου. Ο συνολικός αριθμός των σεναρίων ελέγχου είναι περιορισμένος, συνεπώς υπάρχουν κάποιοι κανόνες για τη μείωση του αριθμού, όπως προτεραιότητα των σεναρίων με βάση τις υψηλής σημασίας περιπτώσεις, βεβαιώνοντας έτσι ότι όλες οι αντιπροσωπευτικές τιμές των κλάσεων εμφανίζονται τουλάχιστον μία φορά.
Κριτήριο ολοκλήρωσης του ελέγχου για τη μέθοδο του διαχωρισμού σε κλάσεις ισοδυναμίας είναι το ορισμένο ποσοστό αυτοδύναμων κλάσεων:

EC-κάλυψη = (αριθμός των κλάσεων ισοδυναμίας που ελέγχθηκαν / συνολικός αριθμός των κλάσεων ισοδυναμίας) * 100 %

b. *Ανάλυση οριακών τιμών (Boundary value analysis)*

Η ανάλυση των οριακών τιμών στον έλεγχο λογισμικού, γίνεται συνήθως σε συνδυασμό με τη μέθοδο του διαχωρισμού σε κλάσεις ισοδυναμίας. Δίνει περισσότερες πληροφορίες για το λογισμικό και καταφέρνει να βρει πολλά ελαττώματα, συνήθως λόγω έλλειψης επακριβούς ορισμού των ορίων.

Σε κάθε όριο, πρέπει να ελέχονται οι οριακές αλλά και οι δυο γειτονικές τιμές, μέσα και έξω από τις κλάσεις ισοδυναμίας.. Συχνά μια γειτονική τιμή αρκεί για να δώσει αποτελέσματα ελέγχου. Οι τιμές από τη μέση μιας κλάσης ισοδυναμίας δεν προσφέρουν επιπλέον πληροφορία για το έλεγχο και θεωρούνται άχρηστες. Η ανάλυση των οριακών τιμών δε μπορεί να χρησιμοποιηθεί για εισερχόμενα δεδομένα που λαμβάνουν διακριτές τιμές, μπορεί όμως να χρησιμοποιηθεί για τις εξερχόμενες κλάσεις ισοδυναμίας.

Όμοια με την προηγούμενη μέθοδο, οι έγκυρες οριακές τιμές συνδυάζονται σε μία περίπτωση ελέγχου της ανάλυσης, (πολλαπλασιαστική μέθοδος), ενώ οι μη έγκυρες οριακές τιμές πρέπει να δοκιμαστούν ξεχωριστά.

Το κριτήριο ολοκλήρωσης του ελέγχου για τη μέθοδο της ανάλυσης οριακών τιμών ορίζεται απο το ποσοστό:

BV-κάλυψη = (αριθμός των οριακών τιμών που ελέγχθηκαν / συνολικός αριθμός οριακών τιμών) * 100%

c. Έλεγχος μετάβασης καταστάσεων (State transition testing)

Σε πολλές περιπτώσεις, το ιστορικό της εκτέλεσης βημάτων είναι σημαντικό στη συμπεριφορά του εξεταζόμενου αντικειμένου. Το διάγραμμα μετάβασης καταστάσεων βασίζεται στο *διάγραμμα καταστάσεων*, το οποίο μοντελοποιεί τη δυναμική συμπεριφορά ενός αντικειμένου και δείχνει όλες τις πιθανές καταστάσεις

τις οποίες μπορεί να βρεθεί αυτό. Ως *κατάσταση* (*state*), ορίζεται το σύνολο τιμών που περιγράφουν την κατάσταση ενός αντικειμένου μια δεδομένη στιγμή. Το *διάγραμμα μετάβασης καταστάσεων* συμβαίνει ως αποτέλεσμα ενός *γεγονότος* (event) και περιγράφει την κατάσταση που θα έχει ένα σύστημα ή μια μονάδα υποδεικνύοντας τα γεγονότα ή τις συνθήκες που το προκάλεσαν.

Ο έλεγχος πρέπει να εκτελέσει όλες τις πιθανές καταστάσεις τουλάχιστον μια φορά. Για το σχεδιασμό των σεναρίων ελέγχου, δημιουργούμε ένα διάγραμμα μετάβασης καταστάσεων, από το οποίο κάθε μονοπάτι αναπαριστά μια περίπτωση, έτσι όλες οι καταστάσεις να εκτελούνται. Για να σχεδιαστεί μια περίπτωση ελέγχου με βάση αυτή τη μεθοδολογία, τα ακόλουθα δεδομένα είναι απαραίτητα:

- Η αρχική κατάσταση του αντικειμένου υπό έλεγχο.
- Τα δεδομένα εισόδου.
- Τα αναμενόμενα αποτελέσματα και δεδομένα εξόδου.
- Η αναμενόμενη τελική κατάσταση.

Για κάθε μετάβαση κατάστασης του σεναρίου ελέγχου, τα ακόλουθα πρέπει να ορίζονται:

- Η κατάσταση πριν την μετάβαση.
- Το αρχικό γεγονός που προκάλεσε τη μετάβαση.
- Η αναμενόμενη αντίδραση που θα προκαλέσει η μετάβαση.
- Η επόμενη κατάσταση που αναμένεται να πάρει το αντικείμενο.

Κριτήρια εξόδου και ολοκλήρωσης τους ελέγχου είναι:

- Το αντικείμενο έχει λάβει όλες τις καταστάσεις τουλάχιστον μία φορά.
- Κάθε μετάβαση έχει εκτελεστεί τουλάχιστον μία φορά.
- Κάθε μετάβαση που παραβιάζεις τις προδιαγραφές έχει εξεταστεί.

Στον έλεγχο μετάβασης καταστάσεων το αντικείμενο υπό έλεγχο μπορεί να είναι ένα ολόκληρο σύστημα με διαφορετικές καταστάσεις συστήματος ή μια κλάση ενός προγράμματος αντικειμενοστραφούς γλώσσας.

Η τεχνική αυτή είναι κατάλληλη για τις δοκιμασίες συστήματος και ειδικά για τον έλεγχο του *γραφικού περιβάλλοντος του χρήστη GUI* (Graphical User Interface). Το GUI αποτελείται από ένα σύνολο οθονών και ελέγχων, όπως μενού και κουτιά διαλόγου κ.ά.. Αν οι οθόνες και οι έλεγχοι θεωρηθούν σαν καταστάσεις και οι εισερχόμενες κινήσεις του χρήστη πάνω σε αυτές σαν μεταβάσεις (π.χ. “OK”, “Cancel”), τότε μπορεί να εξεταστεί με την τεχνική αυτή.

Ο έλεγχος μετάβασης καταστάσεων πρέπει να εφαρμόζεται όταν οι καταστάσεις είναι σημαντικές και όταν η λειτουργικότητα του συστήματος επηρεάζεται από αυτές. Η μέθοδος αυτή είναι ιδιαίτερα χρήσιμη για τους αντικειμενοστραφείς ελέγχους διότι λαμβάνει υπόψη της τις διάφορες φάσεις του αντικειμένου.

Κριτήριο ολοκλήρωσης του ελέγχου για τη μέθοδο είναι:

ST-κάλυψη = (αριθμός μεταβάσεων καταστάσεων που ελέγχθηκαν / συνολικός αριθμός μεταβάσεων καταστάσεων) * 100%

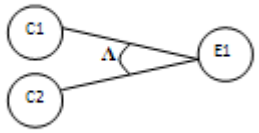
κάτι που είναι σπάνιο λόγω του πολύ μεγάλου αριθμού καταστάσεων.

d. *Γράφος αίτιο-αιτιατό και πίνακας αποφάσεων (Cause-Effect graphing and Decision Table)*

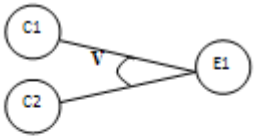
Ο *γράφος αίτιο-αιτιατό* βασίζεται στις σχέσεις ανάμεσα στα δεδομένα εισόδου και δεδομένων εξόδου σύμφωνα με τις προδιαγραφές απαιτήσεων. Τα *αίτια* (*causes*) και τα *αιτιατά* (*effects*) αναπαριστώνται στο γράφο σαν κόμβοι, ενώ οι σχέσεις μεταξύ τους σαν ακμές. Τα αίτια αποτελούνται από *εισερχόμενες προϋποθέσεις* (*conditions*) και λογικούς τελεστές AND, OR και NOT. Έτσι ένα αίτιο μπορεί να είναι *αληθές* (*true*) ή *ψευδές* (*false*).

Έστω ότι έχουμε δύο αίτια C1 και C2 και ένα αιτιατό E1. Η αναπαράσταση στο γράφο ανάλογα με τον τελεστή γίνεται ως εξής:

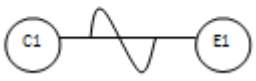
- (C1 AND C2) → E1



- (C1 OR C2) → E1



- (NOT C1) → E1



Για τη δημιουργία του γράφου φέρουμε πρώτα τους κόμβους, αριστερά τα αίτια και δεξιά τα αιτιατά, και έπειτα δημιουργούμε τις ακμές ξεκινώντας πάντα από τα αιτιατά. Εν συνεχεία, ο γράφος μετατρέπεται σε πίνακα αποφάσεων.

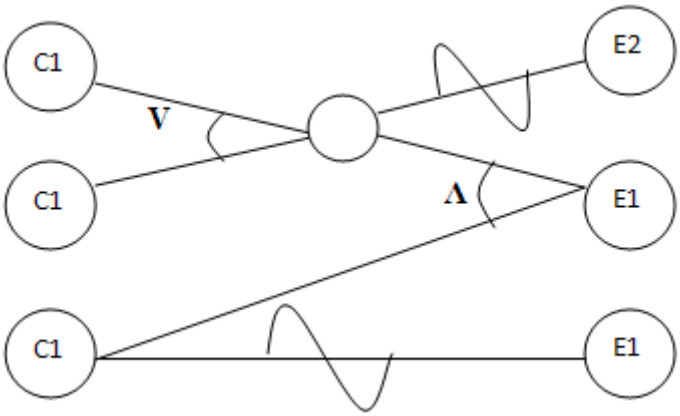
Ο πίνακας αποφάσεων έχει δύο μέρη, το πάνω μισό όπου καταγράφονται τα αίτια και το κάτω μισό όπου περιέχει τα αιτιατά. Όταν ικανοποιείται ένα αίτιο ή αιτιατό τότε αυτό σημειώνεται με Yes/No, True/False ή 1/0. Από τις στήλες του πίνακα αποφάσεων προκύπτουν οι περιπτώσεις ελέγχου.

Τα βήματα για τη μετατροπή του γράφου σε πίνακα είναι: παραθέτουμε τα αίτια και τα αιτιατά στη πρώτη στήλη, επιλέγουμε το πρώτο αιτιατό, από το γράφο βρίσκουμε τους συνδυασμούς των αιτιών που έχουν αυτό το αιτιατό και τους συνδυασμούς των αιτιών που δεν έχουν αυτό το αιτιατό, προσθέτουμε μια στήλη στον πίνακα για κάθε ένα από αυτούς τους συνδυασμούς των αιτιών που προκαλούν αυτό το αιτιατό και τέλος ελέγχουμε αν ο πίνακας αποφάσεων έχει ίδιες στήλες και διαγράφοντας τις διπλοεγγραφές.

Για παράδειγμα, έχουμε το παρακάτω πρόβλημα όπου το πρόγραμμα διαβάζει τις τιμές των πρώτων δύο χαρακτήρων ενός αρχείου και ανάλογα εκτυπώνει:

- Ο πρώτος χαρακτήρας πρέπει να είναι «Α» ή «Β».
- Ο δεύτερος χαρακτήρας πρέπει να είναι ψηφίο.
- Αν ο πρώτος χαρακτήρας είναι «Α» ή «Β» και ο δεύτερος χαρακτήρας ψηφίο τότε το όνομα του αρχείου πρέπει να ανανεωθεί.
- Αν ο πρώτος χαρακτήρας είναι λάθος (ούτε «Α» ούτε «Β») τότε μήνυμα «Χ» πρέπει να εκτυπωθεί.
- Αν ο δεύτερος χαρακτήρας είναι λάθος (δεν είναι ψηφίο) τότε μήνυμα «Υ» πρέπει να εκτυπωθεί.

Τότε ο γράφος αιτίου-αιτιατού είναι:



Και ο πίνακας αποφάσεων είναι:

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

Γενικά κάθε συνδυασμός αιτιών θεωρείται μια περίπτωση ελέγχου, το οποίο σημαίνει ότι *n* συνθήκες (αίτια) έχουν 2^n συνδυασμούς. Για το λόγο αυτό συνήθως οι πίνακες βελτιστοποιούνται και πολλές στήλες φεύγουν. Οι πίνακες αποφάσεων είναι λογικοί έλεγχοι και πρέπει να έχουν ακριβή δεδομένα ελέγχου για να εκτελεστούν. Ελάχιστη

προϋπόθεση για την ολοκλήρωση αυτού του τύπου ελέγχου είναι να εκτελέσουμε όλες τις στήλες από μια φορά. Τα ελαττώματα που προκύπτουν από τη μέθοδο αυτή είναι πολύ ενδιαφέρονται, γιατί προέρχονται από συνδυασμούς δεδομένων εισόδου. Ομως, πολλά λάθη προκύπτουν από την κατασκευή ή την βελτιστοποίηση των πινάκων και χωρίς κατάλληλα εργαλεία η τεχνική δεν είναι εύκολα εφαρμόσιμη.

e. Έλεγχος περιπτώσεων χρήσης (Use case testing)

Η *Ενοποιημένη Γλώσσα Μοντελοποίησης* (*Unified Modeling Language*) ορίζει τη γραφική σημειογραφία, τα UML διαγράμματα, τα οποία χρησιμοποιούνται για την παρουσίαση των απαιτήσεων σε ένα θεωρητικό επίπεδο και περιγράφουν τις αλληλεπιδράσεις του χρήστη με το σύστημα. Τα διαγράμματα των περιπτώσεων χρήσης εξυπηρετούν κυρίως για την παρουσίαση της εξωτερικής εικόνας του συστήματος. Επομένως, ένας έλεγχος βασισμένος σε μία περίπτωση χρήσης είναι χρήσιμος στις δοκιμασίες συστήματος και αποδοχής.

Ο έλεγχος δείχνει τις κανονικές ροές (normal workflow) του συστήματος και έτσι έχει μεγάλη σημασία για τον πελάτη και τον tester επίσης. Για κάθε περίπτωση χρήσης, συγκεκριμένες προυποθέσεις (preconditions) πρέπει να τηρούνται για να εκτελεστεί ο έλεγχος, όπως και μετά την εκτέλεση ορίζονται συγκεκριμένες αναμενόμενες συνθήκες (post-conditions). Επίσης, υπάρχουν γεγονότα που οδηγούν σε εναλλακτικές ροές (alternative workflows), οι οποίες καταγράφονται στις περιπτώσεις χρήσης.

Ένα πιθανό κριτήριο εξόδου του ελέγχου είναι όταν όλες οι ροές των περιπτώσεων χρήσης , κανονικές και εναλλακτικές, εκτελεστούν τουλάχιστον μια φορά. Ωστόσο, τα δεδομένα εισόδου και τα αποτελέσματα του ελέγχου δεν απορρέουν άμεσα από τις περιπτώσεις χρήσης, Για το λόγο αυτό, ο έλεγχος περιπτώσεων χρήσης συνδυάζεται με άλλες μεθόδους, όπως η ανάλυση οριακών τιμών.

Σε γενικές γραμμές, οι διάφοροι μέθοδοι black box ελέγχου έχουν σαν βάση τις απαιτήσεις και τις προδιαγραφές του προιόντος ή του συστήματος. Συνεπώς, δε μπορούν να βρούν ελλαττώματα στις προδιαγραφές αλλά ούτε και επιπλέον λειτουργικότητες που δεν απαιτούνται.Συνεπώς έχει ως μόνο σκοπό την επιβεβαίωση των καταγεγγραμένων λειτουργιών. Επιπλέον, υπάρχουν και άλλα είδη ελέγχου αυτής της τεχνικής όπως *Syntax test, Random test* και *Smoke test*.

2.4.2.2 White box έλεγχος

Η βάση του white box ελέγχου είναι ο έλεγχος του κώδικα του αντικειμένου υπό εξέταση. Επομένως, ο κώδκας πρέπει να είναι διαθέσιμος ενώ σε πολλές περιπτώσεις θα πρέπει να είναι δυνατό και να τον επεξεργαστείς. Η γενική ιδέα είναι να εξεταστεί

κάθε κομμάτι του κώδικα τουλάχιστον μια φορά. Υπάρχουν τέσσερις βασικές τεχνικές του white box ελέγχου:

a. *Κάλυψη καταστάσεων (Statement coverage)*

Η κάλυψη καταστάσεων εστιάζει σε κάθε κατάσταση του αντικειμένου υπό έλεγχο. Το πρώτο βήμα είναι η μετατροπή του κώδικα σε control flow γράφο. Όπως αναφέραμε παραπάνω, ο control flow γράφος αναπαριστά τις καταστάσεις σε κόμβους. Αν μια ακολουθία *μη υποθετικών προτάσεων* (*unconditional statements*) εμφανίζεται στον κώδικα, τότε αναπαριστώνται σαν ένας κόμβος, αφου η εκτέλεση του πρώτου εγγυάται την εκτέλεση και των υπολοίπων. Αντίθετα, οι *υποθετικές* προτάσεις (conditional statements) if και then, όπως και οι *βρόχοι* (*loops*) for, while και do…while έχουν περισσότερες ακμές να εξέρχονται για να καλύψουν όλες τις περιπτώσεις.

Μετά την εκτέλεση πρέπει να επιβεβαιωθεί ποίες από τις καταστάσεις έχουν εκτελεστεί και αν το προκαθορισμένο όριο έχει καλυφθεί τότε ο έλεγχος τερματίζεται. Η μέθοδος εστιάζει στην κάλυψη των ακμών του control flow γράφου. Το κριτήριο εξόδου της μεθόδου κάλυψη καταστάσεων, γνωστό και ως *CO-κάλυψη (C0-coverage)* είναι:

C0-κάλυψη = (αριθμών καταστάσεων που εκτελέστηκαν / συνολικός αριθμός καταστάσεων) * 100%

Στα πλεονεκτήματα της μεθόδου είναι η επιβεβαίωση ότι ο κώδικας κάνει αυτά που πρέπει και δεν πρέπει και ο έλεγχος των μονοπατιών του προγράμματος. Επίσης, αν απαιτείται απόλυτη κάλυψη, δηλαδή 100%, αλλά ορισμένες καταστάσεις δεν μπορούν να καλυφθούν, τότε αυτό μπορεί να είναι ένδειξη *απροσπέλαστου κώδικα* (unreachable source code - dead statements). Βασικά μειονεκτήματα είναι ότι δεν μπορεί να ελέγξει τις ψευδείς συνθήκες, δεν αναφέρει άν ο βρόχος έφτασε στην τερματική του κατάσταση και δεν καταλαβαίνει τους λογικούς τελεστές.

b. *Κάλυψη ακμών (Branch coverage)*

Επίκεντρο της μεθόδου κάλυψης ακμών αποτελούν οι αποφάσεις (ακμές) του αντικειμένου υπό έλεγχο. Στον έλεγχο πρέπει να έχει εξασφαλιστεί ότι η εκτέλεση κάθε απόφασης μπορεί να έχει 2 αποτελέσματα: Αληθές και Ψευδές (TRUE and FALSE). Σε αντίθεση με την κάλυψη καταστάσεων, η κάλυψη ακμών λαμβάνει υπόψη της τα άδεια σημεία, επειδή εκτελούνται όλες οι υπό συνθήκη καταστάσεις. Συνεπώς είναι απαραίτηση η εισαγωγή επιπλέον σεναρίων ελέγχου, που να καλύπτουν όλες τις ακμές και να μπορούν έτσι να ανιχνεύονται οι ελλείψεις καταστάσεων σε άδειες ακμές.

Αναλογικά με την κάλυψη καταστάσεων, το κριτήριο ολοκλήρωσης του ελέγχου για την μέθοδο κάλυψης των ακμών (*C1-coverage)* ορίζεται ώς:

Κάλυψη ακμών = (αριθμός ακμών που εκτελέστηκαν / συνολικός αριθμός ακμών) * 100%

c. *Έλεγχος συνθηκών (Test of conditions)*

Αν μια απόφαση βασίζεται σε διάφορες (μερικές) συνθήκες συνδεόμενες με λογικούς τελεστές τότε στον έλεγχο πρέπει να ληφθεί υπόψη η πολυπλοκότητα της κατάστασης. Οι παρακάτω έλεγχοι περιγράφουν διαφορετικές απαιτήσεις και διαφορετική ένταση των υπό σύνθεση προϋποθέσεων:

- *Έλεγχος συνθηκών ακμών (Branch condition testing)*
  Το μονομερές (μερικό) σύνολο των προϋποθέσεων είναι μια λογική προϋπόθεση που δεν έχει κανέναν λογικό τελεστή (AND, OR and NOT). Σκοπός του ελέγχου είναι η αξιόλογηση κάθε μέρους του προϋποθέσεων από μία φορά, για κάθε μία απ' τις λογικές τιμές Αληθές ή Ψευδές (TRUE or FALSE). Ο έλεγχος συνθηκών ακμών αποτελεί συνεπώς ένα πιο αδύναμο κριτήριο σε σχέση με την κάλυψη καταστάσεων ή ακμών.

- *Έλεχος συνδυασμού συνθηκών ακμών (Branch condition combination testing)*
  Αυτός ο έλεγχος, που επίσης καλείται έλεγχος κάλυψης πολλαπλών συνθηκών, απαιτεί όλοι οι συνδυασμοί Αληθές – Ψευδές να έχουν εκτελεστεί τουλάχιστον μία φορά.Συνεπώς, ο έλεγχος συνδυασμού συνθηκών ακμών ικανοποιεί τα κριτήρια τόσο της κάλυψης καταστάσεων όσο και της κάλυψης ακμών. Στα μειονεκτήματα της μεθόδου συμπεριλαμβάνονται το υψηλό της κόστος, καθώς χρειάζεται 2^n περιπτώσεις ελέγχου για n μέρη προϋποθέσεων, καθώς και ότι δεν είναι δυνατόν να εκτελεστούν όλοι οι συνδυασμοί.

- *Έλεγχος προσδιορισμού προϋποθέσεων (Condition determination testing)*
  Στην μέθοδο αυτή χρησιμοποιείται μόνο εκείνος ο συνδυασμός των λογικών τιμών, για τον οποίο η τροποποίσή τους για μια προϋπόθεση μπορεί να αλλάξει την λογική τιμή του συνόλου των προϋποθέσεων. Ο αριθμός των σεναρίων ελέγχου είναι αισθητά μικρότερος σε σχέση με αυτόν της μεθόδου ελέγχου συνδυασμού συνθηκών ακμών και συνιστά την καλύτερη τεχνική σχεδιασμού σεναρίων ελέγχου για την περίπτωση πολύπλοκών προϋποθέσεων. Επιπλέον, καταλήγει σε κάλυψη τόσο καταστάσεων όσο και ακμών.

Το κριτήριο ολοκλήρωσης του ελέγχου, ανολογικά και με τις προαναφερθείσες μεθόδους, ορίζεται ως η αναλογία ανάμεσα στις εκτελεσθείσες τιμές προς όλες τις απαιτούμενες λογικές τιμές της προϋπόθεσης :

Συνθήκη τερματισμού = (αριθμός των τιμών που εκτελέστηκαν / συνολικός αριθμός των απαιτούμενων λογικών τιμών) * 100%

d. Κάλυψη μονοπατιών (Path coverage)

Τέλος, αν το αντικείμενο περιέχει βρόχους ή επαναλήψεις, οι προηγούμενες μέθοδοι δεν επαρκούν και απαιτείται κάλυψη όλων των πιθανών μονοπατιών του αντικειμένου.

3

Αυτοματοποιημένος έλεγχος λογισμικού

3.1. Πότε και γιατί αυτοματοποίηση [10] [11] [12]

Ο αυτοματοποιημένος έλεγχος λογισμικού έχει πλέον εξελιχθεί από πολυτέλεια σε αναγκαιότητα. Ο *μη αυτοματοποιημένος έλεγχος* ή *«χειρωνακτικός» (manual testing)* δεν μπορεί να συμβαδίσει με τη συνεχή εξέλιξη του μεγέθους και της πολυπλοκότητας των εφαρμογών και των συστημάτων. Τα πρώτα εργαλεία αυτοματοποιημένου ελέγχου είναι διαθέσιμα ήδη από τις αρχές τις δεκαετίας του ’90, με πρώτο το Microsoft Test[4] version 1.0 που βγήκε σε beta έκδοση το 1992.

Ως *μη αυτοματοποιημένος έλεγχος* ορίζεται κάθε «χειρωνακτική» δοκιμή ελέγχου λογισμικού για την εύρεση ελαττωμάτων. Απαιτεί απο τον tester να παίξει το ρόλο του τελικού χρήστη και να χρησιμοποιήσει όλα τα χαρακτηριστικά της εφαρμογής, ώστε να διασφαλίσει την ορθή της συμπεριφορά. Απο την άλλη, *ο αυτοματοποιημένος έλεγχος* συνίσταται στην χρήση ειδικού λογισμικού, διαφορετικού από αυτό που ελέγχεται, που ρυθμίζει την εκτέλεση των δοκιμών και την σύγκριση πραγματικών με τα αναμενόμενα αποτελέσματα. Επιτρέπει το τρέξιμο μιας αλληλουχίας δοκιμών και την δημιουργία ενός αρχείου καταγραφής, που δείχνει ποιες δοκιμές πέτυχαν (*pass*) και απέτυχαν (*fail*) τα αναμενόμενα αποτελέσματα. Αυτό το αρχείο καταγραφής μπορεί είτε να αξιολογηθεί «χειρωνακτικά» από τον tester είτε να επεξεργαστεί αυτόματα για να προσδιοριστεί αν όλα τα αποτελέσματα των δοκιμών ανταποκρίνονται όπως αναμένεται στο προϊόν.

Οι δοκιμές που αυτοματοποιούνται συνήθως μπορούν να τρέξουν και «χειρωνακτικά», κάτι που όμως αποτελεί μια κοπιαστική και χρονοβόρα διαδικασία, καθώς επίσης και επιρρεπή σε ανθρώπινα λάθη. Σε γενικές γραμμές, ένας αυτοματοποιημένος έλεγχος αυτοματοποιεί κάποιες επαναλαμβανόμενες αλλά αναγκαίες διαδικασίες, π.χ. ελέγχους παλινδρόμησης ή αυτοματοποιεί διαδικασίες που θα ήταν αδύνατο να πραγματοποιηθούν «χειρωνακτικά» , π.χ. ελέγχους επίδοσης.

[4]Το Microsoft Test ή MS-Test, αργότερα ονομάστηκε Visual Test, είναι ένα αυτοματοποιημένο εργαλείο λειτουργικού ελέγχου που εξετάζει εφαρμογές σε Windows και αναπτύχθηκε από τη Microsoft.

Ως προς το «τι» πρέπει να αυτοματοποιηθεί, «πότε», ακόμα και το «αν» χρειάζεται αυτοματοποιημένος έλεγχος συνιστούν καίριες αποφάσεις που πρέπει να λάβει η ομάδα διοίκησης (management team) ή ελέγχου (testing team). Η ορθή επιλογή κατάλληλων χαρακτηριστικών του προϊόντος που ελέγχονται αυτόματα καθορίζει την επιτυχία της αυτοματοποίησης. Τα εργαλεία αυτοματοποιημένου ελέγχου μπορεί να είναι ακριβά, καθώς επίσης και ο σχεδιασμός και η υλοποίηση των δοκιμών αυτών μπορούν επίσης να αποδειχθούν χρονοβόροι. Παρ' όλα αυτά, η επένδυση σε μια αυτοματοποιημένη λύση ελέγχου μπορεί μακροπρόθεσμα να γίνει οικονομικά αποδοτική, ειδικά όταν οι δοκιμές τρέξουν πολλές φορές. Τα *σενάρια ελέγχου* (*test scripts*) χρειάζεται να δημιουργηθούν μόνο μία φορά και το κόστος που προστίθεται σε κάθε τρέξιμο της δοκιμής μπορεί να γίνει πολύ χαμηλό σε σύγκριση με το κόστος μιας μη αυτοματοποιημένης δοκιμής ελέγχου.

Επίσης, η επιτυχία της αυτοματοποίησης εξαρτάται και από άλλους παράγοντες. Ένας από αυτούς είναι «πόση αυτοματοποίηση» μας επιτρέπει να κάνουμε η εφαρμογή ή το σύστημα μας. Οι περιπτώσεις που μπορεί να αυτοματοποιηθεί ένα υποσύνολο δοκιμών, γενικά τουλάχιστον το 70% των δοκιμών, και ειδικά οι περιπτώσεις που είναι αναγκαίο το συχνό τρέξιμο αυτών των δοκιμών ελέγχου, συνιστούν ένα αδιάσειστο επιχείρημα υπέρ της επένδυσης σε ένα αυτοματοποιημένο σύστημα ελέγχου.
Επιπλέον, τα scripts αυτοματοποιημένου ελέγχου απαιτούν συστηματική συντήρηση για να διατηρήσουν την λειτουργικότητα και τη χρηστικότητά τους. Επομένως, ο χρόνος ζωής του αυτοματοποιημένου ελέγχου και η ικανότητά του να βρίσκει επιπλέον ελαττώματα σε σχέση με την πρώτη φορά που τρέχει συμβάλλουν καθοριστικά στην επιτυχία του. Ακόμα και μικρές αλλαγές κατά τη διάρκεια του χρόνου ζωής της εφαρμογής μπορούν να προκαλέσουν τη διακοπή των υπαρχόντων δοκιμών, οι οποίες μπορεί να δούλευαν κανονικά σε κάποια προηγούμενη φάση. Στην περίποτωση αυτή η δοκιμή πρέπει είτε να διορθωθεί ή να απορριφθεί. Για παράδειγμα, ένας αυτοματοποιημένος έλεγχος γραφικού περιβάλλοντος του χρήστη δεν έχει μεγάλο χρόνο ζωής, αφού επηρεάζεται από όλες τις αλλαγές στο επίπεδο του GUI. Η συντήρηση αυτοματοποιημένων ελέγχων συνιστά δύσκολο έργο, καθώς μπορεί να αποδειχθεί δυσνόητη κι απαιτητική, κάτι που οδηγεί πολλές ομάδες στην απόρριψη των στρατηγικών αυτοματοποιημένου ελέγχου.
Τέλος, αξίζει να σημειωθεί ότι δεν βασίζονται όλες οι τεχνικές αυτοματοποιημένου ελέγχου στη συγγραφή κώδικαόπως για παράδειγμα οι έλεγχοι με γνώμονα *λέξεις-κλειδιά* (*keyword-driven tests*), σε περιπτώσεις όμως που τα σενάρια ελέγχου είναι γραμμένα σε μορφή πηγαίου κώδικα, τότε ο μηχανικός ελέγχου ή ο ειδικός στη διασφάλιση ποιότητας λογισμικού πρέπει να έχει προγραμματιστικές γνώσεις.

Συνοπτικά τα πλεονεκτήματα του αυτοματοποιημένου ελέγχου είναι:

- Η μείωση του χρόνου ελέγχου.
- Η αυξημένη κάλυψη ελέγχου.
- Η εφαρμογή ελέγχων που δεν μπορούν να πραγματοποιηθούν αλλιώς, π.χ. έλεγχοι επίδοσης, φόρτου και έντασης.

- Τα εργαλεία αυτοματισμού δεν κοιτάζουν απλά την οθόνη, αλλά διερευνούν και τις δομές των δεδομένων, ελέγχοντας επίσης και τις επιχειρησιακές διαδικασίες.
- Κατάλληλο για ελέγχους παλινδρόμησης (regression testing) και καπνού (smoke testing).
- Ο σχεδιασμός των σεναρίων ελέγχου γίνεται πριν την ανάληψη νέας έκδοσης του λογισμικού.
- Στην περίπτωση που δεν είναι εύκολη η αναπαραγωγή ενός σφάλματος που εντοπίστηκε, ο tester μπορεί να ανατρέξει στο αρχείο καταγραφής.

Από την άλλη πλευρά, τα μειονεκτήματα της αυτοματοποίησης του ελέγχου είναι:

- Χρονοβόρο και απαιτεί περισσότερες δεξιότητες απο τους testers.
- Απλοϊκές δοκιμές μπορούν να επηρεάσουν αρνητικά την έκβαση του ελέγχου.
- Φθίνει η λειτουργικότητά του ως προς το χρόνο, λόγω συντήρησης.
- Οι tester μπορούν να παρατηρήσουν αλλόκοτα και παράξενα αποτελέσματα (τα script ελέγχου είναι προσαρμοσμένα στον σκοπό).
- Αν ο «χειρωνακτικός» έλεγχος βρει ένα ελάττωμα, ο tester θα ξανατρέξει τη δοκιμή στο ίδιο σημείο για να επιβεβαιώσει την διόρθωσή του, πιθανώς και περισσότερες από μία φορές (αξιοποιώντας πρακτικά την αρχή ότι τα σφάλματα συνήθως συγκεντρώνονται σε κοντινά σημεία).

3.2. Γενικές κατηγορίες αυτοματοποιημένου ελέγχου λογισμικού [6] [20] [21]

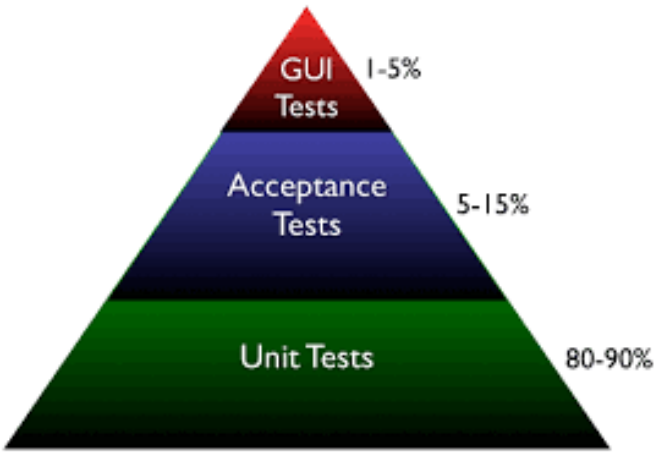
Ο στόχος της αυτοματοποίησης και αντίστοιχα οι τεχνικές και τα εργαλεία του αυτοματοποιημένου ελέγχου εξαρτώνται απόλυτα από το είδος και το επίπεδο ελέγχου.

Συνήθως τα παρακάτω είδη ελέγχου λογισμικού ενδείκνυται να αυτοματοποιηθούν εξολοκλήρου ή σε ένα βαθμό:

- *Λειτουργικός έλεγχος* (*Functional automated testing*)
  - *Έλεγχος παλινδρόμησης* (*Regression testing*)
  - *Αρνητικές δοκιμές* (*Negative testing*)
- *Μη λειτουργικός έλεγχος* (*Non-functional testing*)
  - *Έλεγχος επίδοσης* (*Performance testing*)
  - *Έλεγχος έντασης* (*Stress testing*)
  - *Έλεγχος φόρτου* (*Load testing*)

Ο λειτουργικός έλεγχος, ο οποίος ελέγχει τις σωστές λειτουργίες του λογισμικού είναι ο πιο καλός υποψήφιος για αυτοματοποίηση και μπορεί να κατηγοριοποιηθεί σε

τρία επίπεδα, ανάλογα το επίπεδο της εφαρμογής που ελέγχει. Η *πυραμίδα του αυτοματοποιημένου ελέγχου* (*test automation pyramid*), σύμφωνα με την agile μεθοδολογία ανάπτυξης λογισμικού[5], χωρίζει τον αυτοματοποιημένο έλεγχο σε τρία επίπεδα και αναλόγως προκύπτουν οι τρεις γενικές κατηγορίες αυτοματοποιημένου ελέγχου:



Σχήμα 3.1: Πυραμίδα του αυτοματοποιημένου ελέγχου

- *Αυτοματοποιημένες δοκιμασίες μονάδας* (*Unit level testing or Code-driven testing*)

Οι κλάσεις, τα modules και οι *βιβλιοθήκες* (libraries) δοκιμάζονται με ένα εύρος δεδομένων εισόδου και ελέγχονται αν όλα τα αποτελέσματα είναι σωστά.

- *Αυτοματοποιημένες δοκιμασίες γραφικού περιβάλλοντος* (*Graphical User Interface testing or GUI testing*).

Οι δοκιμασίες παράγουν γεγονότα που αλληλεπιδρούν με το γραφικό περιβάλλον της εφαρμογής, όπως εισαγωγή κειμένου σε ένα πεδίο ή πάτημα ενός κουμπιού, και παρατηρεί τις αλλαγές που γίνονται με στόχο την επικύρωση τους.

- *Αυτοματοποιημένες δοκιμασίες αποδοχής* (*Acceptance level testing or Service/Middle tier based testing or API driven testing*).

Ξεπερνώντας το επίπεδο γραφικού περιβάλλοντος η εφαρμογή ελέγχεται στη μεσαία βαθμίδα, εκεί όπου επικυρώνεται η συμπεριφορά του λογισμικού υπό έλεγχο. Επίσης χρησιμοποιεί μια προγραμματιστική διεπαφή για να «οδηγήσει» το γραφικό περιβάλλον του χρήστη στις επιθυμητές ενέργειες.

[5]Σύμφωνα με την Agile μεθοδολογία (ευελιξίας) το λογισμικό αναπτύσσεται σταδιακά σε μικρούς κύκλους. Αυτό οδηγεί σε μικρές επαναληπτικές και γρήγορες παραδόσεις στην παραγωγή και με κάθε παράδοση να «χτίζεται» στην προηγούμενη και να ελέγχεται διεξοδικά.

3.2.1 Αυτοματοποιημένες δοκιμασίες μονάδας

Στη βάση της πυραμίδας βρίσκονται οι αυτοματοποιημένες δοκιμασίες μονάδας. Όπως και στο χειρωνακτικό έλεγχο λογισμικού, οι δοκιμασίες με βάση τον κώδικα ή τη μονάδα είναι μεγάλης σημασίας για την ανάπτυξη του λογισμικού και για αυτό πρέπει να είναι η βάση μιας αυτοματοποιημένης στρατηγικής ελέγχου.

Οι δοκιμασίες μονάδας γράφονται για να ορίσουν τη λειτουργικότητα πριν ο κώδικας γραφτεί. Στο μεταξύ, οι δοκιμασίες εξελίσσονται και μεγαλώνουν όσο ο κώδικας προχωράει, και όταν όλοι οι έλεγχοι για όλες τις απαιτούμενες λειτουργίες περάσουν, θεωρούμε τον κώδικα ολοκληρωμένο. Οι αυτοματοποιημένες δοκιμασίες σε αυτό το επίπεδο παράγουν λογισμικό πιο αξιόπιστο και λιγότερο δαπανηρό από το *μη αυτοματοποιημένο διερευνητικό έλεγχο* (*manual exploratory testing*). Ο κώδικας θεωρείται πιο αξιόπιστος γιατί η κάλυψη του κώδικα είναι πολύ μεγαλύτερη ενώ η αυτοματοποίηση επιτρέπει πιο εύκολα την επανάληψη του ελέγχου πολλές φορές παράλληλα με τη διαδικασία της ανάπτυξης του λογισμικού. Επίσης, θεωρείται λιγότερο δαπανηρό γιατί ο προγραμματιστής ανακαλύπτει ελαττώματα κατευθείαν μετά από μια αλλαγή στον κώδικα, δηλαδή όσο πιο νωρίς γίνεται και όταν η διόρθωση τους κοστίζει το λιγότερο δυνατό. Τέλος, ο αυτοματοποιημένος έλεγχος μονάδας βοηθάει στην εύκολη και ασφαλή τροποποίηση του κώδικα και στην απλοποίησή αποφεύγοντας επανάληψης *μερών κώδικα* (*code duplication*) μειώνοντας έτσι τα περιθώρια για ελαττώματα στο λογισμικό.

Η αυτοματοποιημένη προσέγγιση των δοκιμασιών μονάδας είναι επίσης ιδιαίτερα δημοφιλής και πετυχημένη διότι παρέχει στους προγραμματιστές επιπλέον πληροφορίες για τα ελαττώματα που ανιχνεύουν. Συγκεκριμένα, αν ένας tester βρει ένα ελάττωμα π.χ. στο πως αποθηκεύει το σύστημα τους νέους χρήστες στη βάση δεδομένων, αυτό το ελάττωμα μπορεί να βρίσκεται σε 1.000 ή περισσότερες γραμμές κώδικα. Απο την άλλη πλευρά ένας αυτοματοποιημένος έλεγχος μονάδας μπορεί να επιστρέψει συγκεκριμένα που βρίσκεται π.χ. στη γραμμή 51 ή 62. Επίσης, ο αυτοματοποιημένος έλεγχος γράφεται συνήθως στην ίδια γλώσσα με τον κώδικα, γεγονός που καθιστά τους προγραμματιστές πιο κατάλληλους για την δημιουργία και εκτέλεση του. Τα τελευταία χρόνια αυτό το είδος αυτοματοποιημένου ελέγχου εκτελείται με τις πλατφόρμες xUnit, όπως η JUnit και NUnit, οι οποίες εξετάζουν αν μέρη του κώδικα λειτουργούν έτσι όπως αναμένεται κάτω από ειδικές συνθήκες.

3.2.2 Αυτοματοποιημένες δοκιμασίες γραφικού περιβάλλοντος χρήστη

Οι αυτοματοποιημένες δοκιμασίες ελέγχου γραφικού περιβάλλοντος εμφανίζονται στην κορυφή της πυραμίδας του αυτοματοποιημένου ελέγχου γιατί προτιμάται να γίνονται σε μικρό βαθμό λόγω των περιορισμών τους.

Για παράδειγμα,θέλουμε να ελέγξουμε μια απλή εφαρμογή αριθμομηχανής που επιτρέπει στο χρήστη να βάλει δύο ακεραίους και να πατήσει είτε το κουμπί του πολλαπλασιασμού είτε το κουμπί της διαίρεσης. Για να ελεγχθεί το γραφικό περιβάλλον αυτής της εφαρμογής χρειάζεται να κατασκευαστεί ένα σενάριο με μια σειρά ελέγχων που θα «οδηγούν» το *γραφικό περιβάλλον* (*driver script*) στα κατάλληλα *αντικείμενα* (*UI objects*). Με τον τρόπο αυτό εισάγονται οι θ τιμές στα πεδία, επιλέγοντας ένα τα δύο κουμπιά και τέλος συγκρίνοντας τις αναμενόμενες με τις πραγματικές τιμές.

Αυτός ο τύπος ελέγχου αποφέρει συνήθως αποτελέσματα με εύκολο τρόπο, απαιτώντας μεγάλο αριθμό σεναρίων, γεγονός που τον καθιστά ακριβό και χρονοβόρο. Επίσης, είναι πολύ ευπαθής αφού μια μικρή αλλαγή στο γραφικό περιβάλλον απαιτεί την ανανέωση των scripts του αυτοματοποιημένου ελέγχου.

3.2.3 Αυτοματοποιημένες δοκιμασίες αποδοχής

Οι αυτοματοποιημένες δοκιμασίες αποδοχής εξετάζουν τις λειτουργίες και τις συμπεριφορές της εφαρμογής ξεχωριστά από το γραφικό περιβάλλον. Ο έλεγχος της μεσαίας βαθμίδας είναι κρίσιμης σημασίας, όχι μόνο γιατί ελέγχει τη σωστή λειτουργία του λογισμικού αλλά γιατί υπερβαίνει το επίπεδο του γραφικού περιβάλλοντος του χρήστη, το οποίο όπως αναφέραμε είναι δαπανηρό και χρονοβόρο.

Τα scripts των αυτοματοποιημένων δοκιμασιών αποδοχής γράφονται σε μια προγραμματιστική γλώσσα ή *γλώσσα σεναρίου* (*scripting language*), εκτελούνται από μια πλατφόρμα ελέγχου ή πάλι μέσω μιας γλώσσας σεναρίου και τέλος συγκρίνουν τα αποτελέσματα με την αναμενόμενη συμπεριφορά της εφαρμογής ή του συστήματος.

Ο έλεγχος ονομάζεται επίσης και δοκιμασίες μεσαίας βαθμίδας ή *βαθμίδας υπηρεσιών* (services). Ως υπηρεσία σε αυτό το επίπεδο θεωρείται η συμπεριφορά της εφαρμογής απέναντι σε κάποια δεδομένα εισόδου ή σύνολα δεδομένων εισόδου και ο έλεγχος εξετάζει τις υπηρεσίες της εφαρμογής ξεχωριστά από το γραφικό περιβάλλον. Στο παραπάνω παράδειγμα της αριθμομηχανής, έχουμε δυο βασικές υπηρεσίες: τον πολλαπλασιασμό και τη διαίρεση. Έτσι, αντί να ελέγξουμε πολλές περιπτώσεις ελέγχου, αντί δηλαδή να εισάγουμε πολλές διαφορετικές τιμές ακεραίων μέσω του γραφικού περιβάλλοντος, ελέγχουμε άμεσα τον πολλαπλασιασμό ή τη διαίρεση.

Το λάθος που γίνεται από τις περισσότερες *ομάδες διοίκησης* (*management teams*) ή *ομάδες ελέγχου* (*testing teams*) είναι ότι για πολλά χρόνια αγνοούσαν αυτό το επίπεδο ελέγχου κατά τον αυτοματοποιημένο έλεγχο. Επειδή οι αυτοματοποιημένες δοκιμασίες μονάδας, δεν καλύπτουν πλήρως τις ανάγκες του ελέγχου, χωρίς η μεσαία βαθμίδα ελέγχου να αναπληρώνει το κενό ανάμεσα στις δοκιμασίες μονάδος και

δοκιμασίες γραφικού περιβάλλοντος, όλοι οι έλεγχοι καταλήγουν να γίνονται στο επίπεδο του γραφικού περιβάλλοντος. Αποτέλεσμα αυτού είναι οι δοκιμασίες να καθίστανται ακριβές για να γραφτούν, να τρέξουν και να εκτελεστούν.

3.3. Τεχνικές αυτοματοποιημένου έλεγχου λογισμικού [6] [7] [8] [9] [11] [13] [14] [22]

Στις περιπτώσεις των αυτοματοποιημένων δοκιμασιών αποδοχής και γραφικού περιβάλλοντος χρήστη, που εξετάζουν συνολικά το αντικείμενο υπό έλεγχο, τα διαθέσιμα εργαλεία ή πλαίσια ελέγχου (test frameworks) αυτοματοποίησης βασίζονται σε μια *τεχνική αυτοματοποιημένου ελέγχου (test automation technique)*.

Πλαίσιο αυτοματοποιημένου ελέγχου (*test automation framework*) είναι ένα συνενωμένο σύστημα που ορίζει τους κανόνες της αυτοματοποίησης ενός συγκεκριμένου προϊόντος ή συστήματος. Το σύστημα αποτελείται από τις *βιβλιοθήκες συναρτήσεων* (*test libraries*), τις πηγές δεδομένων ελέγχου, τις δομές για την καταγραφή των ελαττωμάτων και μία πλατφόρμα για τη δημιουργία δομών ελέγχου. Τέλος το πλαίσιο παρέχει τη βάση του αυτοματοποιημένου ελέγχου και συμβάλει σημαντικά στην απλοποίηση της προσπάθειας αυτοματοποίησης.

Το μεγάλο πλεονέκτημα των πλαισίων αυτοματοποιημένου ελέγχου είναι το χαμηλό κόστος συντήρησης. Αν υπάρχει έστω και μία μικρή αλλαγή σε μια περίπτωση ελέγχου τότε μόνο το αρχείο με τη συγκεκριμένη περίπτωση ελέγχου χρειάζεται να ανανεωθεί, τα *σενάρια-οδηγοί* (*driver scripts*) και τα *εναρκτήρια σενάρια* (*startup scripts*) παραμένουν ως έχουν και δε χρήζουν αλλαγής. Μερικές φορές δεν χρειάζονται ούτε τα σενάρια αλλαγές, όπως για παράγειγμα στην περίπτωση που αλλάξουν τα δεδομένα ελέγχου τα οποία είναι αποθηκευμένα σε ξεχωριστά αρχεία.

Η επιλογή του σωστού πλαισίου ελέγχου, δηλαδή της σωστής τεχνικής αυτοματοποίησης που ακολουθεί το πλαίσιο, είναι η πιο σημαντική, τόσο για την επιτυχία του ελέγχου όσο και για το κόστος της αυτοματοποίησης. Το πλαίσιο αυτοματοποιημένου ελέγχου είναι υπεύθυνο για:

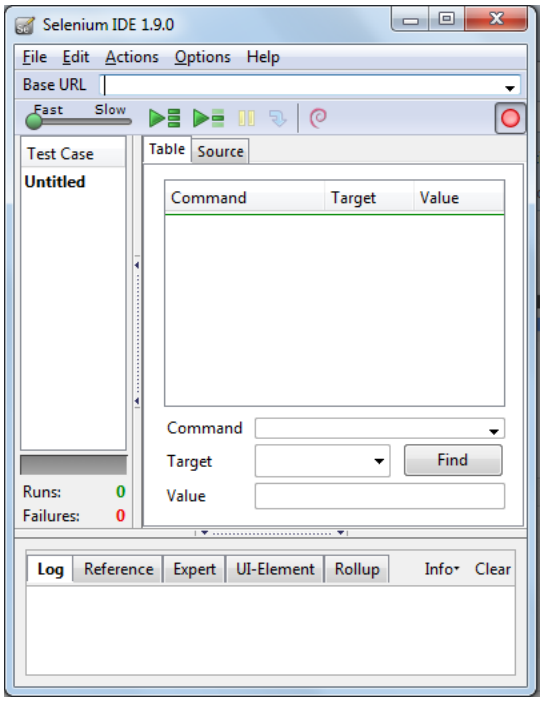
1. Τον ορισμό της μορφολογίας (format) στην οποία θα εκφραστούν οι περιπτώσεις ελέγχου.
2. Τη δημιουργία ενός μηχανισμού που θα «οδηγεί» την εφαρμογή υπό έλεγχο.
3. Την εκτέλεση των δοκιμασιών.
4. Την καταγραφή των αποτελεσμάτων ελέγχου.

Όπως θα δούμε και παρακάτω, πολλά εργαλεία ή ολοκληρωμένα πλαίσια ελέγχου ακολουθούν πολλές τεχνικές αυτοματοποίησης.

3.3.1 Τεχνική «Καταγραφή/Επανάληψη» [8] [10] [22]

Πολλά εργαλεία αυτοματοποιημένου ελέγχου παρέχουν την *τεχνική της καταγραφής/επανάληψης* (*Record/Playback technique*) επιτρέποντας την καταγραφή των αλληλεπιδράσεων και ενεργειών των χρηστών πάνω στα αντικείμενα γραφικού περιβάλλοντος. Το σενάριο που παράγεται από τη διαδικασία της καταγραφής αποτελείται από μια λίστα ενεργειών στη γλώσσα του εργαλείου, οι οποίες μπορούν να επαναληφθούν αυτοματοποιημένα πάνω στο αντικείμενο υπό έλεγχο. Αφού το σενάριο καταγραφεί, ο κώδικάς μπορεί να επεξεργαστεί «χειρωνακτικά» και να προστεθούν μεταβλητές, συνθήκες, βρόχοι κτλ.

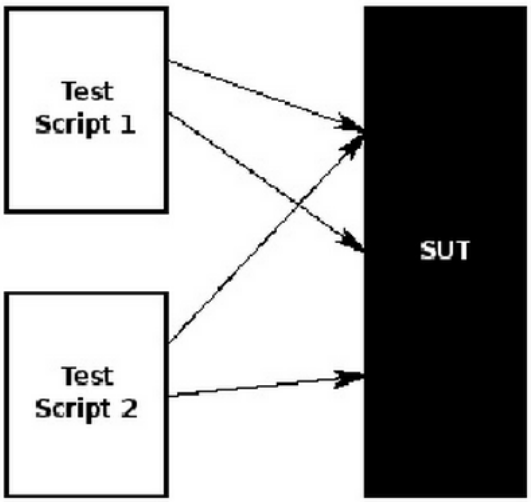
Η παραπάνω μέθοδος ανήκει στις αυτοματοποιημένες δοκιμασίες ελέγχου γραφικού περιβάλλοντος, είναι εύκολη σε χρήση και δεν απαιτεί προγραμματιστικές ικανότητες. Ωστόσο, η εξάρτηση από τα αντικείμενα γραφικού περιβάλλοντος δημιουργεί πολλά προβλήματα αξιοπιστίας και συντηρησιμότητας, ειδικά σε περίπτωση αλλαγών της διεπαφής του χρήστη. Τα σενάρια συνίστανται από μια μεγάλη λίστα ενεργειών με δεδομένα εισόδου hard-coded πάνω στα αντικείμενα του γραφικού περιβάλλοντος. Επίσης, η τεχνική κατά το στάδιο της καταγραφής προσθέτει πολλές φορές λανθασμένα ενέργειες που δε σχετίζονται με το σενάριο με αποτέλεσμα ακόμα μεγαλύτερα σενάρια και δυσκολία στη συντήρησή τους. Τέλος, το σύστημα πρέπει να είναι έτοιμο (σεταρισμένο) έτσι ώστε να ξεκινήσει ο αυτοματοποιημένος έλεγχος, ενώ σε περίπτωση μη αναμενόμενων λαθών ή εξαιρέσεων του συστήματος δε γνωρίζει πώς να τα διαχειριστεί. Γενικά θεωρείται καλή τεχνική για αυτοματοποιημένο έλεγχο γραφικού περιβάλλοντος, αλλά δεν ενδείκνυται για μεγάλες εφαρμογές ή συστήματα.



Σχήμα 3.3: Περιβάλλον εργαλείου αυτοματοποιημένου ελέγχου Selenium.

3.3.2 Γραμμική τεχνική [22]

Η *γραμμική τεχνική* (*linear scripting*) απαιτεί τη δημιουργία μικρών, ανεξάρτητων και μη δομημένων σεναρίων ελέγχου, δηλαδή σεναρίων που δεν περιέχουν συνθήκες και βρόχους, και αλληλεπιδρούν άμεσα με το υπο έλεγχο σύστημα. Τα scripts μπορούν να γραφτούν σε οποιαδήποτε γλώσσα προγραμματισμού ή μπορούν να παραχθούν από αυτοματοποιημένα εργαλεία της μεθόδου καταγραφής/επανάληψης.



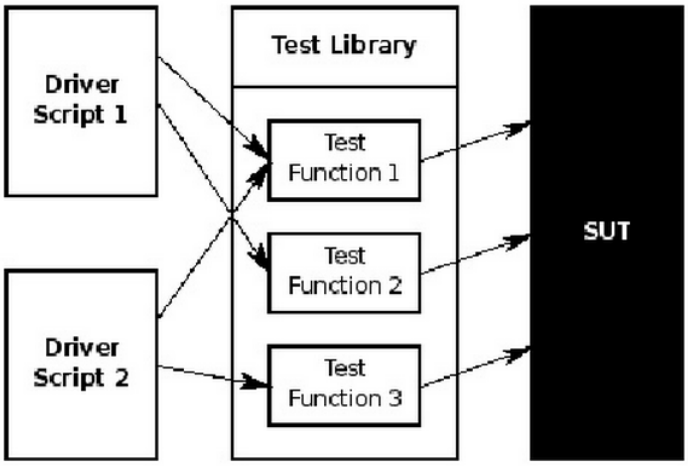
Σχήμα 3.4: Σχηματική απεικόνιση της γραμμικής τεχνικής.

Στα πλεονεκτήματα της γραμμικής τεχνικής είναι η ευκολία δημιουργίας και προσαρμοστικότητας των σεναρίων, ενώ στα μειονεκτήματα ο tester χρειάζεται να έχει προγραμματιστικές ικανότητες και τα σενάρια ελέγχου είναι πολύ εύπαθή και δύσκολο να συντηρηθούν. Για απλούς και μικρούς ελέγχους η τεχνική θεωρείται επαρκής.

3.3.3 Δομημένη τεχνική [8] [10] [22]

Με την βοήθεια της *δομημένης τεχνικής* (*modular scripting*) δημιουργούνται *«σενάρια-οδηγοί»* (*driver scripts*) που περιέχουν δομές, όπως *if…else, switch, for* και *while,* οι οποίες αλληλεπιδρούν με το σύστημα υπό έλεγχο μέσω συναρτήσεων που ανήκουν σε *βιβλιοθήκες.*

Στην συγκεκριμένη τεχνική η επαναχρησιμοποίηση των σεναρίων και η δημιουργία καινούργιων είναι πιο εύκολη, ενώ η συντηρησιμότητα του κώδικα σε περίπτωση αλλαγών απαιτεί λιγότερες διορθώσεις σε μικρότερες περιοχές. Τα «σενάρια-οδηγοί» είναι γενικά απλά, ακόμα και αρχάριοι προγραμματιστές μπορούν να τα δημιουργήσουν και να επεξεργαστούν, σε αντίθεση με τις βιβλιοθήκες που απαιτούν χρόνο και καλές προγραμματιστικές ικανότητες.



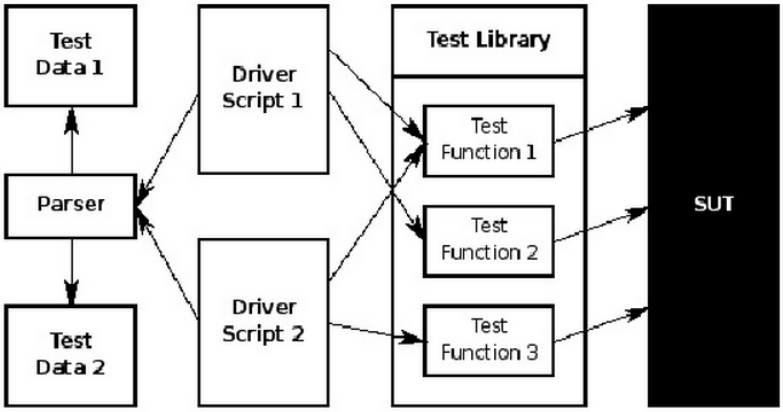
Σχήμα 3.5: Σχηματική απεικόνιση της δομημένης τεχνικής.

Επιπλέον η εισαγωγή δεδομένων ελέγχου γίνεται μόνο μέσω των σεναρίων και κάθε νέος έλεγχος απαιτεί νέα «σενάρια-οδηγούς». Γενικά, η δομημένη τεχνική είναι κατάλληλη για αυτοματοποίηση μικρών και μεσαίων εφαρμογών.

3.3.4 Τεχνική βάσει δεδομένων [6] [7] [8] [14] [22]

Η *τεχνική αυτοματοποίησης βάσει δεδομένων* (*data-driven testing*) διαχωρίζει τα δεδομένα ελέγχου από το σενάριο ελέγχου. Τα δεδομένα ελέγχου βρίσκονται σε ξεχωριστά *αρχεία* (*data files*) και από εκεί φορτώνονται στις μεταβλητές του σεναρίου ελέγχου είτε ως δεδομένα εισόδου είτε ως δεδομένα εξόδου. Με την τεχνική αυτή, οι έλεγχοι εξετάζουν πολλούς συνδυασμούς δεδομένων και τιμών, επιταχύνοντας έτσι μια καλύτερη κάλυψη των περιπτώσεων ελέγχων. Για την επιλογή των δεδομένων ελέγχου γίνεται συχνά χρήση των μεθόδων διαχωρισμού σε κλάσεις ισοδυναμίας και ανάλυσης συνοριακών τιμών.

Οτιδήποτε μπορεί να αλλάξει (όπως το περιβάλλον, τα κριτήρια ελέγχου, τα δεδομένα εισόδου και εξόδου, κτλ.) διαχωρίζεται από την λογική ελέγχου, δηλαδή τα σενάρια ελέγχου και μεταφέρεται «εξωτερικά». Κάθε φορά που ο tester θέλει να προσθέσει μια περίπτωση ελέγχου, προσθέτει ένα νέο δεδομένο στο αρχείο χωρίς να χρειαστεί να γράψει επιπλέον σενάριο.Τα πλαίσια ελέγχου αυτής της τεχνικής επιτρέπουν τη δημιουργία σεναρίων ελέγχου που «οδηγούν» την εφαρμογή (driver scripts) και τρέχουν μαζί με τα σχετικά δεδομένα ελέγχου. Τα σενάρια αυτά διαχειρίζονται το περιβάλλον ελέγχου, φορτώνουν τα δεδομένα ελέγχου και καταγράφουν τα αποτελέσματα. Επίσης, κάνουν χρήση των βιβλιοθηκών του πλαισίου και προσφέρουν μια επαναχρησιμοποιήσιμη λογική που μειώνει την συντήρηση και βελτιώνει την κάλυψη του ελέγχου.



Σχήμα 3.6: Σχηματική απεικόνιση της τεχνικής βάσει δεδομένων.

Τα «σενάρια-οδηγοί» απαιτούν χρήση κώδικα, αντίθετα με τις περιπτώσεις ελέγχου που καλύπτονται από τα δεδομένα ελέγχου και είναι πολύ απλές συνήθως σε μορφή πίνακα π.χ. '1 + 2 = 3' (πρόσθεση) ή '1 * 2 = 2' (πολλαπλασιασμός). Κάθε νέο είδος περιπτώσεων ελέγχου π.χ. '1 *2 + 3 = 6' (πρόσθεση και πολλαπλασιασμός μαζί, προτεραιότητα πράξεων) χρειάζεται νέο «σενάριο-οδηγό». Τα δεδομένα εισόδου ή εξόδου (κριτήρια ελέγχου) μπορούν να αποθηκευτούν σε μία ή πολλές κεντρικές πηγές δεδομένων ή βάσεις δεδομένων, η μορφολογία και η οργάνωση των οποίων εξαρτάται από την εφαρμογή που ελέγχουν. Οι βάσεις δεδομένων που χρησιμοποιούνται από τη μέθοδο για την αποθήκευση των δεδομένων ελέγχου είναι:

- *«Πισίνες» δεδομένων* (*Data pools*)
- *ODBC πηγές* (*sources*)
- CSVor CVS αρχεία
- Excel αρχεία
- DAO αντικείμενα
- ADO αντικείμενα

	A	B	C	D	E
1	Test Case	Number 1	Operator	Number 2	Expected
2	Add 01	1	+	2	3
3	Add 02	1	+	-2	-1
4	Sub 01	1	-	2	-1
5	Sub 02	1	-	-2	3
6	Mul 01	1	*	2	2
7	Mul 02	1	*	-2	-2
8	Div 01	2	/	1	2
9	Div 02	2	/	-2	-1

Σχήμα 3.7: Παράδειγμα αρχείου δεδομένω για αυτοματοποιημένο έλεγχο.

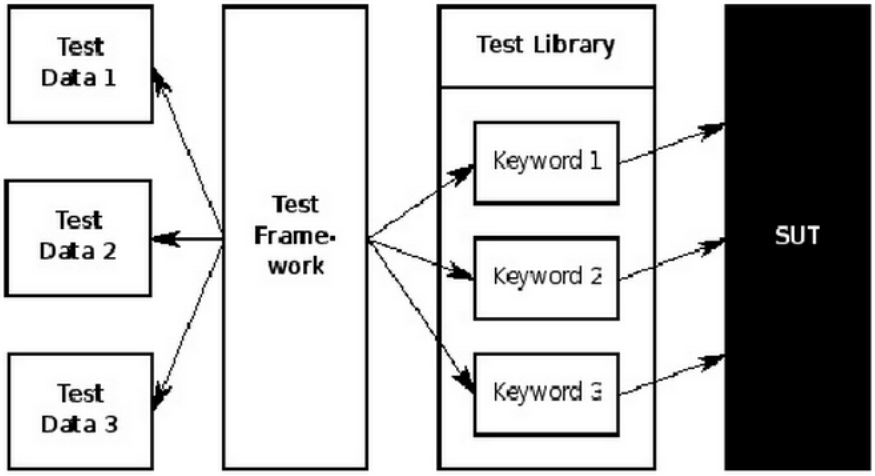
Με αυτόν τον τύπο πλαισίων αυτοματοποιημένου ελέγχου μπορεί να εκτελεστεί ένας μεγάλος αριθμός ελέγχων από ένα σενάριο, αντί να δημιουργηθεί ένας μεγάλος αριθμός σεναρίων για να ελεγχθεί κάθε μια συνθήκη ξεχωριστά. Ενδείκνυται για εφαρμογές που είναι στο στάδιο της υλοποίησης.

Μια βελτίωση της βασικής τεχνικής είναι η επιλογή της *ακολουθίας ελέγχου* (*test sequence*), δηλαδή η σειρά που εισέρχονται τα δεδομένα ελέγχου στο αντικείμενο μπορεί πλέον να καθοριστεί. Επιπλέον, στην ακολουθία ελέγχου μπορούμε να παραλείψουμε δεδομένα, μετατρέποντας έτσι τα αρχεία δεδομένων ελέγχου σε πραγματικά σενάρια ελέγχου. Αυτό είναι ένα βήμα ώστε ο έλεγχος αυτός να μην είναι απλά ένα «φόρτωμα» δεδομένων στο σύστημα.

3.3.5 Τεχνική βάσει λέξεων-κλειδιών [6] [7] [8] [14] [22]

Ο *έλεγχος βάσει λέξεων-κλειδιών,* γνωστός και ως *έλεγχος βάσει πίνακα* (*keyword-driven testing or table-driven testing framework*), είναι μια μεθοδολογία ελέγχου λογισμικού τόσο για «χειρωνακτικό» όσο και για αυτοματοποιημένο έλεγχο. Η μέθοδος ξεχωρίζει την τεκμηρίωση περιπτώσεων ελέγχου, συμπεριλαμβανομένου των δεδομένων ελέγχου, από τον τρόπο που θα εκτελεστούν οι περιπτώσεις ελέγχου. Οι λέξεις-κλειδιά κάνουν χρήση μιας ειδικής *«μετα-γλώσσας» υψηλού επιπέδου* (*high-level meta language*), αρκετά περιγραφικής ως προς το αντικείμενο που ελέγχουν. Η «μετα-γλώσσα» είναι διαφορετική για κάθε πλαίσιο αυτοματοποιημένου ελέγχου.

Ο έλεγχος βάσει λέξεων-κλειδιών μοιάζει με τον έλεγχο βάσει δεδομένων, με τη διαφορά ότι οι περιπτώσεις ελέγχου του πρώτου περιλαμβάνονται στα αρχεία των δεδομένων ελέγχου και όχι στο σενάριο ελέγχου. Το σενάριο ελέγχου στην περίπτωση αυτή είναι απλώς ένας «οδηγός» για τα δεδομένα που είναι αποθηκευμένα στις πηγές δεδομένων. Αντίθετα, στη τεχνική βάσει λέξεων-κλειδιών τα δεδομένα ελέγχου γίνονται τα σενάρια ελέγχου, τα οποία ορίζουν «τι θα κάνει» ο έλεγχος και με ποία σειρά θα το κάνει, και μαζί με τα δεδομένα ελέγχου «οδηγούν» το αντικείμενο υπό έλεγχο και την εκτέλεση του ελέγχου.



Σχήμα 3.7: Σχηματική απεικόνιση της τεχνικής βάσει λέξεων-κλειδιών.

Η τεχνική χρησιμοποιεί *λέξεις-κλειδιά* (*keywords*) για να συμβολίσει τις λειτουργικότητες που θα ελεγχθούν π.χ. Insert Username, Insert Password, ορίζοντας το σύνολο των ενεργειών που πρέπει να εκτελεστούν για να κάνει ένας χρήστης login. Το συντακτικό λέξεων-κλειδιών χρησιμοποιεί τη μορφή πίνακα, όπου η πρώτη γραμμή περιέχει το τίτλο της περίπτωσης ελέγχου που ελέγχεται, Valid Login, η πρώτη στήλη περιέχει τις λέξεις-κλειδιά και η δεύτερη τα δεδομένα ελέγχου που χρειάζονται για να εκτελεστεί η κάθε λέξη κλειδί.

```
*** Test Cases ***

Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Σχήμα 3.8: Παράδειγμα περίπτωσης ελέγχου σε πλαισίο βάσει λέξεων-κλειδιών Robot

3.3.6 Υβριδική τεχνική [14] [22]

Το πιο συνηθισμένο πλαίσιο αυτοματοποιημένου ελέγχου είναι το *υβριδικό πλαίσιο ελέγχου* (*hybrid testing framework*), το οποίο συνδυάζει κάποιες από των παραπάνω τεχνικές, κατά κύριο λόγο τις τεχνικές βάσει δεδομένων και βάσει λέξεων-κλειδιών. Το υβριδικό πλαίσιο επιτρέπει στα σενάρια ελέγχου της τεχνικής βάσει δεδομένων να εκμεταλλευτούν τις βιβλιοθήκες και όλες τις ωφέλειες της αρχιτεκτονικής της

μεθόδου βάσει λέξεων-κλειδιών, με αποτέλεσμα τα σενάρια να είναι πιο συμπαγή και λιγότερο επιρρεπή σε σφάλματα. Συνεπώς, όλοι οι έλεγχοι του υβριδικού αυτοματοποιημένου ελέγχου προσφέρουν υψηλότερο επίπεδο αυτοματοποίησης. Επίσης, τα πλαίσια αυτής της τεχνικής διαχειρίζονται λάθη, εξαιρέσεις του συστήματος ή μη αναμενόμενα παράθυρα. Συνίσταται για μεσαίες και μεγάλες εφαρμογές.

3.4. Εργαλεία αυτοματοποιημένου ελέγχου σήμερα

Τα εργαλεία αυτοματοποιημένου ελέγχου λογισμικού, ανάλογα με την προσβασιμότητα στο κώδικα και την διάθεσή τους, χωρίζονται στις εξής τρεις κατηγορίες [22]:

- *Εμπορικά εργαλεία* (*Commercial tools*).

Καλά εργαλεία αλλά και ακριβά, ακόμα και αυτά που έχουν φθηνές άδειες (licenses) δεν επιτρέπουν τη χρήση τους από ολόκληρη την ομάδα. Επίσης, είναι δύσκολο να συνενωθούν με άλλα εργαλεία αυτοματοποιημένου ελέγχου, ειδικά με εργαλεία άλλης εταιρίας, και είναι δύσκολο έως αδύνατο να προσαρμοστούν από το χρήστη. Τέλος, υπάρχει ο κίνδυνος η εταιρία να κλείσει ή το εργαλείο να αποσυρθεί.

- *Εργαλεία ανοιχτού λογισμικού* (Open source tools)

Υπάρχει μεγάλη ποικιλία εργαλείων ανοιχτού λογισμικού, επιρεάζοντας όμως το επίπεδο ποιότητας. Βασικό τους χαρακτηριστικό είναι ότι διατίθονται δωρεάν και ελεύθερα να χρησιμοποιηθούν από όλους με έμφαση στη δυνατότητα προσαρμογής στις ανάγκες της εφαρμογής. Επιπλέον είναι πολύ εύκολο να συνενωθούν με άλλα εργαλεία.

- *Ελεύθερα εργαλεία* (Freeware tools)

Ελεύθερα εργαλεία είναι πλέον σπάνια σήμερα, αφού όλα είναι και ανοιχτού λογισμικού. Όσα υπάρχουν, είναι δωρεάν και χωρίς κόστος άδειας (license). Είναι πιο εύκολο να τα συνενώσεις με άλλα αυτοματοποιημένα εργαλεία από ότι τα εμπορικά, άλλα όχι τόσο εύκολο όσο τα ανοιχτού λογισμικού. Επίσης, ελοχεύει ο κίνδυνος απόσυρσης του εργαλείου.

Τα σημαντικότερα εργαλεία ή πλαίσια αυτοματοποιημένου λειτουργικού ελέγχου που κυκλοφορούν σήμερα, μπορούν να διαχωριστούν ανάλογα με το επίπεδο και τον έλεγχο που εκτελούν σε:

Εργαλεία για αυτοματοποιημένες δοκιμασίες μονάδας:

- **JUnit** [23]

Το JUnit είναι ένα πλαίσιο ανοιχτού λογισμικού για τις δοκιμασίες μονάδας της γλώσσας προγραμματισμού Java. Ανήκει στην κατηγορία πλαισίων αρχιτεκτονικής xUnit και συνδέεται σαν JAR κατά τη διάρκεια της μεταγλώττισης του κώδικα. Ο JUnit έλεγχος είναι ουσιαστικά ένα αντικείμενο Java, όπου οι μέθοδοι του ελέγχου συνοδεύονται από το λεκτικό "`@Test`". Είναι επίσης εφικτό να οριστεί πότε η μέθοδος θα εκτελέσει τον έλεγχο πριν (`@Before`) ή μετά (`@After`), για μία ή όλες τις μεθόδους (`@BeforeClass` ή `@AfterClass`).

- **NUnit** [24]

NUnit είναι επίσης ένα πλαίσιο ανοιχτού λογισμικού για τις δοκιμασίες μονάδας που απευθύνεται σε όλες της γλώσσες .NET. Αυτό το πλαίσιο xUnit αρχιτεκτονικής ανήκει στην εταιρία Microsoft και είναι εξολοκλήρου γραμμένο σε C#.

Εργαλεία για αυτοματοποιημένες δοκιμασίες αποδοχής:

- **Robot Framework** [25]

Το Robot Framework είναι ένα γενικό πλαίσιο αυτοματοποιημένου ελέγχου για δοκιμές αποδοχής. Ακολουθεί την τεχνική βάσει λέξεων-κλειδιών και χρησιμοποιεί συντακτικό για τα δεδομένα ελέγχου σε μορφή πίνακα. Οι δυνατότητες του πλαισίου επεκτείνονται από τις διαθέσιμες βιβλιοθήκες ελέγχου που είναι υλοποιημένες σε Python ή Java. Οι χρήστες μπορούν επίσης να δημιουργήσουν λέξεις-κλειδιά από τις ήδη υπάρχουσες, χρησιμοποιώντας το ίδιο συντακτικό με αυτό των περιπτώσεων ελέγχου.

Οι πίνακες μπορούν να είναι σε μορφή απλού κειμένου (.txt), HTML (.html, .htm, xhml), tab-separated values (.tsv), ή eStructured Text rest (.robot) τύπους αρχείων και γράφονται σε ένα οποιοδήποτε επεξεργαστή κειμένου (text editor) ή στο Robot Integrated Development Environment (RIDE). Ο RIDE απλοποιεί το γράψιμο των περιπτώσεων ελέγχου παρέχοντας στο πλαίσιο *ειδική συμπλήρωση κώδικα* (*code completion*), *επισήμανση συντακτικού (syntax highlighting)* κ.ά..

Το Robot είναι ανεξάρτητο από την εφαρμογή και το λειτουργικό σύστημα στο οποίο τρέχει. Είναι επίσης ανοιχτού λογισμικού, όπως και οι περισσότερες βιβλιοθήκες και εργαλεία που υποστηρίζει. Η ανάπτυξη του πλαισίου υποστηρίζεται από την εταιρία Nokia Networks.

- **FitNesse** [26]

Το FitNesse είναι ένα πλαίσιο αυτοματοποιημένου ελέγχου λογισμικού για δοκιμασίες αποδοχής γραμμένο σε Java υποστηρίζοντας επίσης C++, Python, Ruby, Delphi, C# κ.ά..

Οι δοκιμασίες που γίνονται στο FitNesse βασίζονται στην τεχνική των λέξεων-κλειδιών και είναι οι περιπτώσεις ελέγχου σε μορφή ζευγαριού των δεδομένων εισόδου και αναμενόμενων δεδομένων εξόδου. Τα ζευγάρια εκφράζονται σε διάφορες παραλλαγές πίνακα όπως πίνακες ελέγχων που εκτελούν queries, πίνακες που εκφράζουν σενάρια ελέγχου με ακριβή αρίθμηση των βημάτων που πρέπει να ακολουθηθούν για να φτάσουν στο αποτέλεσμα, όπως επίσης και πίνακες ελεύθερης δομής.

- **Cucumber** [27]

Το Cucumber είναι ένα πλαίσιο ανοιχτού λογισμικού γραμμένο σε Ruby που αφορά πάντα δοκιμασίες αποδοχής. Δεν απευθύνεται μόνο σε εφαρμογές λογισμικού της Ruby π.χ. cuke4php και cuke4lua. Το πλαίσιο Cucumber επιτρέπει επίσης την εκτέλεση επίσημων εγγράφων με τα τεχνικά *χαρακτηριστικά λογισμικού* (*feature documentation*) σε μορφή ειδικού συντακτικού πλαισίου.

Εργαλεία για αυτοματοποιημένες δοκιμασίες γραφικού περιβάλλοντος χρήστη:

- **Selenium** [28]

Το Selenium είναι ένα εργαλείο ελέγχου λογισμικού για εφαρμογές του διαδικτύου. Αφορά μόνο δοκιμασίες γραφικού περιβάλλοντος και είναι ανοιχτού λογισμικού. Παρέχει την τεχνική «καταγραφής/επανάληψης» μέσω του Selenium I DE που υπάρχει σαν plug-in στους περισσότερους μοντέρνους *περιηγητές διαδικτύου* (web browsers), όπως οι Firefox, Chrome.

Παρέχει επίσης και ένα ολοκληρωμένο πλαίσιο για τη δημιουργία ελέγχων σε ένα μεγάλο αριθμό προγραμματιστικών γλωσσών όπως Java, C#, Groovy, Perl, PHP, Python και Ruby.

- **Watir** [29]

Το Watir (προφέρεται σαν “water”) είναι ένα εργαλείο ανοιχτού λογισμικού με όλες τις οικογένειες βιβλιοθηκών της Ruby για την αυτοματοποίηση των περιηγητών του διαδικτύου (web browsers). Υποστηρίζει όλες τις εφαρμογές διαδικτύου ανεξάρτητα από την τεχνολογία με την οποία έχουν αναπτυχθεί. Ενώ το Watir υποστηρίζει αποκλειστικά τον Internet Explorer στο λειτουργικό των Windows, το Watir-Web Driver υποστηρίζει πλέον και τους Chrome, Firefox και Opera. Το εργαλείο παρέχει τη δυνατότητα συνδέσης του έλεγχου με βάσεις δεδομένων, διαβάζει αρχεία

δεδομένων και spreadsheets, εξάγει XML και δομεί τον κώδικα μέσω των διαθέσιμων βιβλιοθηκών.

- **QTP και UFT [30]**

Το Quick Test Professional (QTP) μαζί με το Service Test (ST) αποτελούν πλέον ένα ενοποιημένο πλαίσιο ελέγχου λογισμικού, το HP Unified Functional Testing (UFT), ένα εμπορικό εργαλείο που ανήκει στην εταιρία Hewlett Packard.

Παρέχει αυτοματοποιημένο λειτουργικό έλεγχο λογισμικού και έλεγχο παλινδρόμησης για εφαρμογές λογισμικού και περιβάλλοντα. Το ενοποιημένο εργαλείο επιτρέπει στους προγραμματιστές να ελέγξουν από μία κονσόλα και τις τρεις βαθμίδες του λογισμικού. Στη μεσαία βαθμίδα ακολουθεί μια υβριδική τεχνική, υποστηρίζοντας τις μεθόδους βάσει δεδομένων και βάσει λέξεων-κλειδιών. Χρησιμοποιεί τη γλώσσα σεναρίων[6] Visual Basic Scripting Edition (VB Script) για να προσδιορίσει τα σενάρια ελέγχου και να διαχειριστεί τα αντικείμενα της εφαρμογής υπό έλεγχο.

- **Rational Functional Tester [31]**

Το αυτοματοποιημένο πλαίσιο ελέγχου Rational Functional Tester είναι εμπορικό εργαλείο και ανήκει στην IBM. Επιτρέπει λειτουργικό έλεγχο και έλεγχο παλινδρόμησης στο γραφικό περιβάλλον, ενώ υποστηρίζει τις τεχνικές «καταγραφής/επανάληψης» με δυνατότητα επεξεργασίας των παραγόμενων σεναρίων. Επιπλέον παρέχει τεχνική βάσει δεδομένων για την εισαγωγή δεδομένων στο υπό έλεγχο αντικείμενο. Το Rational Function Tester υποστηρίζει ένα μεγάλο φάσμα εφαρμογών, όπως τις εφαρμογές διαδικτύου, .Net, Java, Siebel, SAP, τερματικές εφαρμογές βάσει εξομοιωτή, PowerBuilder, Ajax, Adobe Flex, Dojo Toolkit, GEF, Adobe PDF αρχεία, zSeries, iSeries, και pSeries.

- **Galen Framework [32]**

Τα τελευταία χρόνια στα αυτοματοποιημένα εργαλεία ελέγχου γραφικού περιβάλλοντος του χρήστη προστίθονται και εργαλεία για τον έλεγχο του responsive web design, δηλαδή για τον έλεγχο των γραφικών περιβαλλόντων σε μικρότερες οθόνες όπως π.χ. σε smartphones και tablets.

Το πλαίσιο ελέγχου Galen χρησιμοποιεί την τεχνική έλεγχου βάσει λέξεων-κλειδιών και ελέγχει τις αποστάσεις των αντικειμένων γραφικού περιβάλλοντος σε σχέση με το μέγεθος της σελίδας ή της οθόνης. Με απλά λόγια, ανοίγει ένα περιηγητή διαδικτύου

[6]Γλώσσες σεναρίων, εκ του scripting languages, θεωρούνται οι low-level γλώσσες προγραμματισμού των οποίων ο πηγαίος κώδικας δεν χρειάζεται μεταγλώττιση σε γλώσσα μηχανής από ένα μεταγλωττιστή (compiler) αλλά χρειάζεται κάποιο πρόγραμμα για να τρέξει. Πολλές απ'αυτές τις γλώσσες είναι interpreted ή υβριδικές γλώσσες.

(μέσω του εργαλείου Selenium), αναπροσαρμόζει το μέγεθος του και μετά ελέγχει τα αντικείμενα σύμφωνα με τις προδιαγραφές. Για τον ορισμό των προδιαγραφών χρησιμοποιεί ένα ειδικό συντακτικό με κατανοητούς όρους και κανόνες. Τέλος, παράγει HTML αναφορές με screenshots των οθονών και υπογραμμισμένα όλα τα στοιχεία που δε συμφωνούν με τις προδιαγραφές.

Πλαίσια αυτοματοποιημένου μη λειτουργικού ελέγχου για ελέγχους επίδοσης, φόρτου και έντασης:

- **JMeter** [33]

Το Apache JMeter είναι μια εφαρμογή ανοικτού λογισμικού γραμμένη σε Java που ανάμεσα σε άλλες λειτουργίες μπορεί να φορτώνει το υπό έλεγχο αντικείμενο και να μετρά τις επιδόσεις του. Μπορεί λοιπόν να εκτελέσει ελέγχους επιδόσεων, φόρτου και έντασης.

Αρχικά, προοριζόταν μόνο για εφαρμογές διαδικτύου αλλά έχει πλέον επεκταθεί και σε άλλα αντικείμενα ελέγχου, υποστηρίζοντας πολλά διαφορετικά πρωτόκολλα: HTTP, HTTPS, SOAP, FTP, Database via JDBC, LDAP, MongoDB (NoSQL), TCP κ.ά..

- **HP LoadRunner** [34]

Το LoadRunner είναι ένα εμπορικό αυτοματοποιημένο πλαίσιο ελέγχου επιδόσεων και φόρτου που ανήκει στην Hewlett-Packard. Η λογική του πλαισίου είναι να δημιουργεί εικονικούς χρήστες που παίρνουν τη θέση των πραγματικών χρηστών του προϊόντος λογισμικού. Μπορεί να προσομοιώσει χιλιάδες χρήστες που λειτουργούν ταυτόχρονα σε μια εφαρμογή, συλλέγοντας πληροφορίες από τις μονάδες συστήματος (web servers, database servers κτλ.) και καταγράφοντας τα αποτελέσματα, τα οποία μπορούν να αναλυθούν σε λεπτομέρεια για την ανίχνευση των προβλημάτων.

Το LoadRunner αποτελείται από τέσσερις μονάδες:

- *VuGen (Virtual User Generator)* για τη δημιουργία και επεξεργασία των σεναρίων. Υποστηρίζει την τεχνική της «καταγραφής/επανάληψης» και επιτρέπει την εισαγωγή δομημένου κώδικα. Τα σενάρια γράφονται σε ANSI-C, μια ειδική γλώσσα που μοιάζει πολύ με τη C.
- *Load generator* για την παραγωγή του φόρτου στην εφαρμογή.
- *Controller* για τη δημιουργία των περιπτώσεων ελέγχου. Εκεί ορίζεται ποια σενάρια, για πόσους χρήστες και για πόση ώρα θα τρέξει ο κάθε load generator.
- Analysis για την συγκομιδή των *αρχείων* (*logs*) από τους load generators και τη δημιουργία reports για την οπτικοποίηση των αποτελεσμάτων.

4

Έλεγχος λογισμικού στο «πρόβλημα των τριγώνων»

4.1. Το «πρόβλημα των τριγώνων» [15] [16]

Το «Πρόβλημα των τριγώνων», γνωστό επίσης ως πρόβλημα τριγώνων των Weinberg-Myers, είναι ένα κλασικό πρόβλημα ελέγχου που τέθηκε από τον Jerry Weinberg και δημοσιεύτηκε για πρώτη φορά στο βιβλίο «Η τέχνη του Ελέγχου Λογισμικού» του Glenford Myers, το 1979. Ο ίδιος το χρησιμοποίησε ως παράδειγμα μίας απλής εφαρμογής που όμως χρειάζεται πολλές δοκιμές ελέγχου. Ο Myers υπήρξε ο πρώτος που αντιμετώπισε τον έλεγχο λογισμικού ως ένα εντελώς ανεξάρτητο κομμάτι στην ανάπτυξη λογισμικού και προκάλεσε τους αναγνώστες του να γράψουν περιπτώσεις ελέγχου για το πρόβλημα με τις παρακάτω συνθήκες:

«Το πρόγραμμα διαβάζει 3 ακέραιες τιμές από ένα διάλογο εισόδου. Αυτές αντιπροσωπεύουν το μήκος των πλευρών ενός τριγώνου. Το πρόγραμμα εμφανίζει μήνυμα που δηλώνει αν το τρίγωνο είναι σκαληνό, ισοσκελές ή ισόπλευρο.»[15]

Στην αρχική του μορφή το πρόβλημα χρησιμοποιήθηκε για προγράμματα ανάγνωσης διάτρητων καρτών μηχανής (IBM punch cards). Η εξέταση αυτή ελέγχει την ικανότητα σκέψης και δημιουργίας περιπτώσεων ελέγχου για μια δεδομένη κατάσταση και παρουσιάζει τη διαδικασία σχεδιασμού ελέγχου (test design procedure).

Στο βιβλίο του, επίσης, ο Myers περιγράφει τις 14 δοκιμές που χρειάζονται για να ελεγχθεί επαρκώς ένα τρίγωνο. Στη συνέχεια, στο βιβλίο «Έλεγχος Λογισμικού: Μια πρόχειρη προσέγγιση», ο Paul Jorgensen καταγράφει 185 περιπτώσεις ελέγχου. Ο Bob Binder παίρνει την σκυτάλη του προβλήματος στην εισαγωγή του βιβλίου του «Έλεγχος Αντικειμενοστραφών Συστημάτων» όπου περιγράφει ένα περίτεχνο σύστημα αντικειμένων που κάνουν ουσιαστικά την ίδια δουλειά. Παρότι δεν έχει τις ίδιες προϋποθέσεις με το αρχικό πρόβλημα του Myers ισχυρίζεται ότι χρειάζεται 65 δοκιμές για να το ελέγξει. Απ' την άλλη, ο Kent Beck ισχυρίζεται ότι χρειάζεται μόνο 6 για την λύση του προβλήματος ακολουθώντας την *τεχνική ανάπτυξης λογισμικού βάσει ελέγχου* (Test-driven Development).

Το πρόβλημα των τριγώνων είναι ιδιαίτερα δημοφιλές στην κοινότητα των testers, καθώς πολλά βιβλία και ιστολόγια έχουν πολλάκις ασχοληθεί. Στην συγκεκριμένη

εργασία επιλέχθηκε η web 2.0 υλοποίηση της Elizabeth Hendrickson[7] για το κλασικό πρόβλημα ελέγχου τριγώνων, καθώς ο κώδικας είναι γραμμένος σε JavaScript και συνεπώς είναι διαθέσιμος για έλεγχο. Η επιλογή αυτής της εφαρμογής διαδικτύου καθορίστηκε απ' την ανάγκη να γίνει ανεξάρτητος και αντικειμενικός έλεγχος μιας εφαρμογής που δεν έχει υλοποιηθεί στο παίσιο της διπλωματικής εργαίας, ικανοποιώντας έτσι το βασικό κανόνα του Ελέγχου Λογισμικού περί ανεξαρτησίας των ομάδων ανάπτυξης λογισμικού και ελέγχου.

4.2. Έλεγχος λογισμικού στην εφαρμογή των τριγώνων [16]

Για τον έλεγχο λογισμικού του προβλήματος των τριγώνων, όπως και κάθε προβλήματος ελέγχου, και τη δημιουργία περιπτώσεων ελέγχου δεν υπάρχει σωστή θεωρητική απάντηση και ιδανικός τρόπος εκτέλεσης. Η απάντηση εξαρτάται από τον tester, τις υποθέσεις που θα πάρει και τις μεθόδους που θα επιλέξει.

Για αρχή ο tester οφείλει να καταλάβει το πρόβλημα και την εφαρμογή που πρέπει να ελέγξει μελετώντας τις απαιτήσεις του προβλήματος. Οι συνθήκες της εφαρμογής των τριγώνων ορίζονται από την δημιουργό της εφαρμογής παρακάτω:

«*Παίρνει ως είσοδο τρεις αριθμούς που αντιπροσωπεύουν το μήκος των τριών πλευρών του τριγώνου. Αυτόματα*[8] *το πρόγραμμα σχεδιάζει μια εικόνα του τριγώνου με το μέγεθος των πλευρών σε αναλογία και δηλώνει τον τύπο του τριγώνου.*»[19]

Οι συνθήκες που δόθηκαν θεωρούνται οι απαιτήσεις του προβλήματος, και όπως συμβαίνει στις περισσότερες εφαρμογές ή συστήματα του πραγματικού κόσμου, οι πληροφορίες που έχουμε για το λογισμικό είναι είτε ελλιπής είτε ασαφής. Παρατηρούμε ότι η Hendrickson, σε αντίθεση με τον Myers, δεν αναφέρει αν τα δεδομένα εισόδου περιορίζονται στους ακεραίους αριθμούς και δεν προσδιορίζει ποιοι είναι οι τύποι τριγώνων που αναγνωρίζει και εμφανίζει το σύστημα. Συνεπώς, περαιτέρω ανάλυση των απαιτήσεων είναι αναγκαία για την κάλυψη αυτών των κενών και τη δημιουργία *υποθέσεων* (*assumptions*) πάνω στις οποίες θα βασιστεί ο έλεγχος λογισμικού.

Κατά τον έλεγχο λογισμικού που πραγματοποιείται «χειρωνακτικά» (δηλαδή χωρίς καμία αυτοματοποίηση αλλά με μία απλή αλληλεπίδραση με την εφαρμογή από τη θέση του τελικού χρήστη) θα πραγματοποιήσουμε δύο είδη διερευνητικών ελέγχων. Αρχικά θα εκτελέσουμε ένα «έλεγχο καπνού» για να την εξέταση των βασικών περιπτώσεων και στη συνέχεια ένα πιο ενδελεχή έλεγχο βάσει απαιτήσεων, με τη μέθοδο του black box.

[7] http://practice.agilistry.com/triangle

[8] Στις αρχικές συνθήκες της εφαρμογής αναφερόταν ότι το πρόγραμμα σχεδιάζει την εικόνα του τριγώνου μετα από πάτημα του κουμπιού "Draw". Μετά την αναβάθμιση της εφαρμογής του 2010 , το τρίγωνο σχεδιάζεται αυτόματα χωρίς τη χρήση κουμπιού.

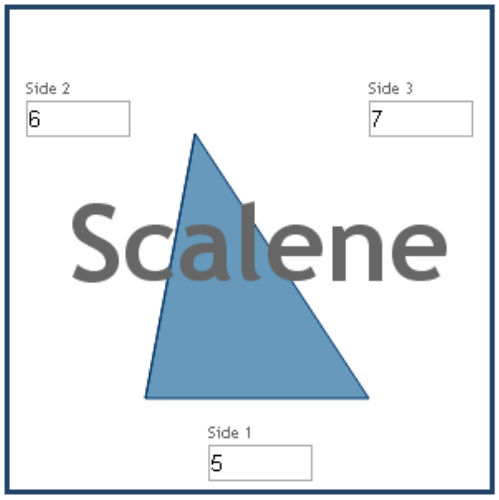
4.2.1 Έλεγχος καπνού της εφαρμογής

Εν αρχή θα κάνουμε ένα «έλεγχο καπνού» για να εξοικειωθούμε με την εφαρμογή και να ελέγξουμε αν οι πολύ βασικές περιπτώσεις ελέγχου λειτουργούν σύμφωνα με τις απαιτήσεις. Στον έλεγχο αυτό θα ελέγξουμε ότι η εφαρμογή δέχεται τρεις τιμές εισόδου, που αναπαριστούν τις τρεις πλευρές του τριγώνου, (δεδομένα εισόδου) και επιστρέφει τον τύπο του τριγώνου και την γραφική αναπαράσταση του σε αναλογία (δεδομένα εξόδου).

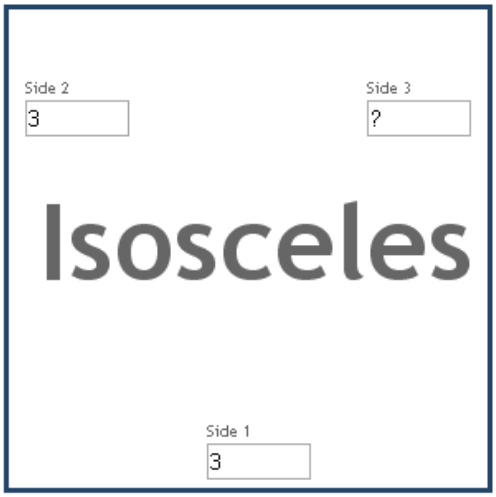
Για τον «έλεγχο καπνού» της εφαρμογής των τριγώνων αρκεί να εξεταστούν οι βασικοί τύποι τριγώνων. Επομένως, πρέπει να ελεγχθούν οι περιπτώσεις όπου τα δεδομένα εισόδου σχηματίζουν ισόπλευρο, ισοσκελές και σκαληνό τρίγωνο. Σε αυτό το σημείο αναρωτιόμαστε αν υπάρχει και άλλος τύπος τριγώνου που δεν καλύπτεται από τις προδιαγραφές του Myers. Όντως, υπάρχει επίσης το ορθογώνιο τρίγωνο, το οποίο πρέπει να λάβουμε υπόψη μας. Σε όλες τις περιπτώσεις ελέγχουμε αν η εφαρμογή επιστρέφει το σωστό τύπο τριγώνου (“Equilateral”, “Isosceles” και “Scalene”), αν σχηματίζει σωστά το τρίγωνο και αν αυτό είναι σε αναλογία με τις τιμές που δώσαμε για την κάθε πλευρά. Επιπλέον, θα προσθέσουμε και μια μη έγκυρη περίπτωση ελέγχου, όπου τα δεδομένα εισόδου δεν είναι σωστά, για να εξετάσουμε πως ανταποκρίνεται η εφαρμογή.

Οι περιπτώσεις του «ελέγχου καπνού» είναι:

- Για τα δεδομένα εισόδου **(1, 1, 1)** ο τύπος τριγώνου είναι: **Equilateral.**
- Για τα δεδομένα εισόδου **(2, 2, 3)** ο τύπος τριγώνου είναι: **Isosceles.**
- Για τα δεδομένα εισόδου **(5, 6, 7)** ο τύπος τριγώνου είναι: **Scalene.**
- Για τα δεδομένα εισόδου **(3, 4, 5)** ο τύπος τριγώνου είναι: **Right.**
- Για τα δεδομένα εισόδου **(3, 3, ?)** ο τύπος τριγώνου είναι: **Isosceles.**



Σχήμα 4.1: Γραφική αναπαράσταση τριγώνου για τις τιμές (5, 6, 7).



Σχήμα 4.2: Γραφική αναπαράσταση τριγώνου για τα μη έγκυρα δεδομένα (3, 3, ?).

Εκτελούμε τον «έλεγχο καπνού» και παρατηρούμε ότι οι τέσσερις έγκυρες περιπτώσεις *«πέρασαν»* τον έλεγχο (*pass*), ενώ η μη έγκυρη περίπτωση *δεν «πέρασε»* (*fail*). Κατά την μη έγκυρη περίπτωση ελέγχου το σύστημα επέστρεψε ως τύπο τριγώνου "Isosceles" και δεν σχημάτισε γραφικά καθόλου το τρίγωνο. Η εφαρμογή θα έπρεπε να μην επιτρέπει στο χρήστη την εισαγωγή ειδικού χαρακτήρα, επιστρέφοντας ίσως κάποιο *μήνυμα λάθους* (*error message*) ή επιστρέφοντας ως τύπο τριγώνου "Invalid".

Παρόλο που ο «έλεγχος καπνού» βρήκε ένα ελάττωμα στο λογισμικό, υπάρχουν πολλά ελαττώματα που δε θα φανερωθούν με αυτές τις πέντε περιπτώσεις ελέγχου και για αυτό απαιτείται ένας πιο ενδελεχής έλεγχο. Η ερώτηση «πόσες περιπτώσεις αρκούν για να ελέγξουμε καλά» την εφαρμογή μπορεί να απαντηθεί μόνο με βάση την επικινδυνότητα της εφαρμογής και την προσπάθεια που χρειάζεται να καταβληθεί για να αποφευχθεί αποτυχία (effort). Αν η επικινδυνότητα είναι σχετικά χαμηλή τότε μπορεί αυτές οι πέντε περιπτώσεις ελέγχου να αρκούν. Στην περίπτωση μας, το επίπεδο επικινδυντότητας θεωρείται υψηλό και για το λόγο αυτό θα εκτελεστεί *εκτενής έλεγχο (thorough testing).*

4.2.2 Black box έλεγχος της εφαρμογής

Αφού έχουμε μια πρώτη εικόνα για την εφαρμογή από τον «έλεγχο καπνού», το επόμενο βήμα είναι ο έλεγχος λογισμικού βάσει απαιτήσεων με τη μέθοδο του black box. Δηλαδή χωρίς να λάβουμε υπόψη μας τον κώδικα της εφαρμογής και με μόνη γνώση τις απαιτήσεις θα εκτελέσουμε τον έλεγχο λογισμικού. Απαιτείται να αναλύσουμε τις προδιαγραφές που μας έχουν δοθεί και ταυτόχρονα να διεξάγουμε διερευνητικό έλεγχο για την συμπλήρωση αυτών.

Από τις απαιτήσεις της εφαρμογής υπό έλεγχο προκύπτουν οι παρακάτω ερωτήσεις:

- Τι είναι τρίγωνο και πως ορίζεται;
- Πόσοι τύποι τριγώνων υπάρχουν και πώς ορίζονται αυτοί;
- Τι είδους αριθμούς δέχεται ως εισόδους η εφαρμογή;
- Οι αριθμοί αυτοί έχουν περιορισμένο πεδίο ορισμού (ακρότατες τιμές);
- Τρεις οποιοδήποτε αριθμοί φτιάχνουν ένα τρίγωνο;

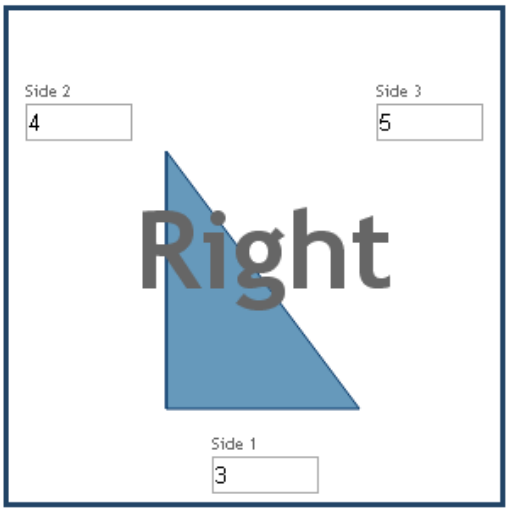
Για την απάντηση αυτών των ερωτήσεων πρέπει να γίνουν κάποιες υποθέσεις, αφού οι απαιτήσεις έχουν κενά και δεν εκτελούμε white box έλεγχο για να ξέρουμε πως ο προγραμματιστής έφτιαξε τον κώδικα. Οι υποθέσεις είναι μέγιστης σημασίας για το σχεδιασμό του ελέγχου διότι πάνω σε αυτές θα βασιστούν οι περιπτώσεις ελέγχου. Επιπλέον πρέπει πάντα να καταγράφονται και να επιθεωρούνται από την υπόλοιπη ομάδα ελέγχου έτσι ώστε να υπάρχει μια κοινή αντίληψη για το αντικείμενο υπό έλεγχο.

Στην περίπτωση του προβλήματος των τριγώνων, η πιο βασική υπόθεση που πρέπει να γίνει είναι αυτή για τον ορισμό του τριγώνου. Οι υποθέσεις λαμβάνουν πάντα υπόψη της τον πελάτη ή τον τελικό χρήστη που απευθύνεται η εφαρμογή. Ως εκ τούτου, ένας διδακτορικός φοιτητής Μαθηματικών ή ένα παιδί δημοτικού έχουν μια πολύ διαφορετική αντίληψη του τριγώνου από ένα κοινό ορισμό[9]. Αν αναζητήσουμε τον ορισμό σε λεξικό, το τρίγωνο ορίζεται ως «το σχήμα που περιφράζεται από τρεις γραμμές»[17] ή «το πολύγωνο με τρεις πλευρές»[18], όπου «πολύγωνο είναι ένα σχήμα του δισδιάστατου χώρου περιφραγμένο με ευθείες γραμμές»[18]. Στην παρούσα ανάλυση θα βασιστούμε στο δεύτερο ορισμό που είναι πιο αυστηρός.

Επίσης, από την Ευκλείδεια Γεωμετρία ορίζουμε τους βασικούς τύπους τριγώνων ως προς τις πλευρές: Το *ισόπλευρο τρίγωνο* ως το τρίγωνο που έχει τρεις ίσες πλευρές, το *ισοσκελές* ως το τρίγωνο που έχει ακριβώς δύο πλευρές ίσες μεταξύ τους και το *σκαληνό* ως αυτό που καμία από τις πλευρές δεν είναι ίση μεταξύ τους. Υπάρχουν δύο είδη σκαληνών τριγώνων, το *οξυγώνιο* με τρεις γωνίες οξείες και το *αμβλυγώνιο* με μια γωνία αμβλεία. Επίσης, *ορθογώνιο τρίγωνο* ορίζεται το τρίγωνα με μια γωνία ορθή.

Οι πλευρές που περιέχουν την ορθή γωνία ενός ορθογωνίου λέγονται κάθετες πλευρές και η απέναντι της λέγεται υποτείνουσα. Κριτήριο για να είναι ένα τρίγωνο ορθογώνιο είναι το Πυθαγόρειο θεώρημα, δηλαδή «το τετράγωνο της υποτείνουσας ενός ορθογώνιου τριγώνου ισούται με το άθροισμα των τετραγώνων των δύο κάθετων πλευρών».

[9] Ένας διδακτορικός φοιτητής Μαθηματικών γνωρίζει επίσης και άλλες γεωμετρίες, όπως του Riemman, όπου οι πλευρές του τριγώνου μπορούν να είναι καμπύλες πάνω επιφάνειες. Σε αυτές τις γεωμετρίες μπορεί π.χ. να υπάρξει ένα τρίγωνο με πλευρές (1, 10 (πάνω σε επιφάνεια), 1)



Σχήμα 4.3: Γραφική αναπαράσταση ορθογωνίου τριγώνου με τιμές (3, 4, 5).

Ένας πολύ μεγάλος αριθμός σφαλμάτων προέρχεται από τις ίδιες τις απαιτήσεις και τις προδιαγραφές του συστήματος, έτσι δεν μπορούμε να υποθέσουμε ότι οι απαιτήσεις που μας δόθηκαν για το πρόβλημα είναι ολοκληρωμένες και σωστές. Στην περίπτωση μας, οι προδιαγραφές δεν αναφέρουν τι είδους είναι τα δεδομένα εισόδου και ποιό είναι το πεδίο ορισμού τους. Οπότε είναι απαραίτητο να ελεγχθεί αν τα δεδομένα ελέγχου μπορούν να περιέχουν δεκαδικό μέρος ή είναι απλώς ακέραιοι.

- Για τα δεδομένα εισόδου **(3.2, 5.2, 6.2)** ο τύπος τριγώνου είναι: **Equilateral**

Συνεπώς οι δεκαδικοί αριθμοί είναι αποδεκτοί και πρέπει να δοκιμαστούν διεξοδικά για την εύρεση ελαττωμάτων που προκαλούνται από *εσωτερικές αριθμητικές υπερχειλίσεις (internal arithmetic overflow)* και λάθη στρογγυλοποίησης.

Επίσης, πρέπει να ελέγξουμε την εφαρμογή για το ποιο είναι το μεγαλύτερο και μικρότερο επιτρεπτό μήκος των δεδομένων εισόδου, αφού αυτό δεν προσδιορίζεται από τις απαιτήσεις. Ξέρουμε ότι οι αρνητικοί αριθμοί και το μηδέν δεν είναι επιτρεπτά δεδομένα για πλευρές τριγώνου. Δοκιμάζουμε άμεσα:

- Για τα δεδομένα εισόδου **(0.1, 0.1, 0.1)** ο τύπος τριγώνου είναι: **Degenerate.**
- Για τα δεδομένα εισόδου **(0.9, 1, 1)** ο τύπος τριγώνου είναι: **Degenerate.**
- Για τα δεδομένα εισόδου **(100, 100, 100)** ο τύπος τριγώνου είναι: **Equilateral.**
- Για τα δεδομένα εισόδου **(10.000, 10.000, 10.000)** ο τύπος τριγώνου είναι: **Equilateral.**
- Για τα δεδομένα εισόδου **(1.000.000, 1.000000, 1.000.000)** ο τύπος τριγώνου είναι: **Equilateral.**

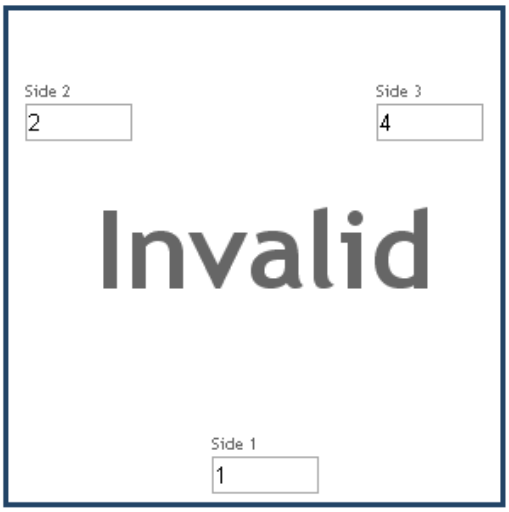
Αφού δοκιμάσουμε τις παραπάνω τριάδες αριθμών παρατηρούμε ότι το σύστημα δεν δέχεται αριθμούς μικρότερους της μονάδας, παρόλο που δέχεται δεκαδικούς αριθμούς, χωρίς όμως να παρουσιάζει κανένα περιορισμό για το άνω όριο των τιμών

εισόδου. Ενδεχομένως αυτή η έλλειψη άνω ορίου να προκαλεί πολλά σφάλματα στην εφαρμογή, ειδικά για την κατά αναλογία γραφική αναπαράσταση πολύ μεγάλων τριγώνων. Υποθέτουμε λοιπόν για ακρότατες τιμές εισόδου το 1 και το 999 .

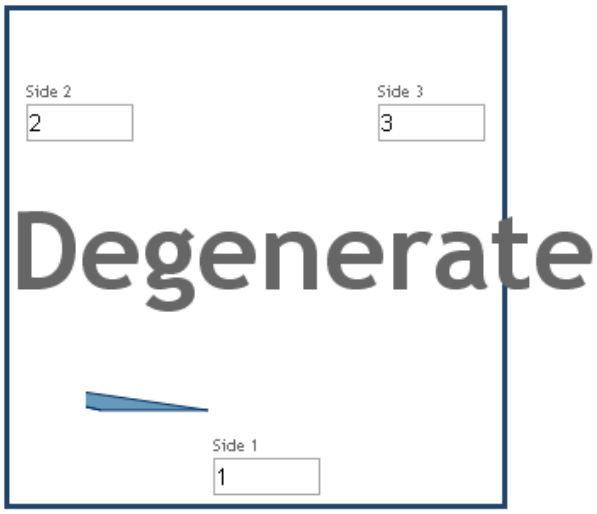
Τέλος, κατά την ανάλυση των απαιτήσεων παρατηρούμε ότι δεν επιστέφουν όλες οι τριάδες δεδομένων εισόδου τρίγωνο. Αν δοκιμάσουμε:

- Για τα δεδομένα εισόδου **(1, 2, 4)** ο τύπος τριγώνου είναι: **Invalid.**
- Για τα δεδομένα εισόδου **(1, 2, 3)** ο τύπος τριγώνου είναι: **Degenerate.**

Ως συνέπεια της υπόθεσης που πήραμε για τον ορισμό του τριγώνου, οι πλευρές του τρίγωνου πρέπει να περικλείουν μια περιοχή για να ορίζεται αυτό. Στην πρώτη περίπτωση που δοκιμάσαμε οι τρεις τιμές είναι έγκυρες αλλά δεν σχηματίζεται τρίγωνο γιατί δεν κλείνουν ποτέ, ενώ στη δεύτερη περίπτωση οι τιμές εισόδου είναι στην ίδια ευθεία.



Σχήμα 4.1: Γραφική αναπαράσταση τριγώνου όπου το άθροισμα των δύο μικρότερων πλευρών είναι μικρότερο από την μεγαλύτερη.



Σχήμα 4.2: Γραφική αναπαράσταση τριγώνου όπου το άθροισμα των δύο μικρότερων πλευρών είναι ίσο από την μεγαλύτερη.

Συμβουλευόμαστε πάλι την Ευκλείδεια Γεωμετρία και βλέπουμε ότι αν α, β, γ τρεις θετικοί αριθμοί, η ανίσωση $|\beta-\gamma| < \alpha < \beta + \gamma$ είναι ικανή και αναγκαία συνθήκη για να αποτελούν πλευρές τριγώνου. Επομένως, αν δεν ισχύει η τριγωνική ανισότητα τότε δεν σχηματίζεται τρίγωνο.

Από την ανάλυση των απαιτήσεων της εφαρμογής και τον διερευνητικό έλεγχο αποκτήσαμε μια πιο ολοκληρωμένη εικόνα για την εφαρμογή, τι είναι αυτό που θα ελέγξουμε και συλλέξαμε πληροφορίες που θα μας φανούν χρήσιμες για τον σχεδιασμό των περιπτώσεων ελέγχου. Συνοψίζοντας, καταλήξαμε στις παρακάτω υποθέσεις και παρατηρήσεις:

- Για να σχηματίζουν οι τρεις τιμές εισόδου τρίγωνο πρέπει το άθροισμα των τριών μικρότερων πλευρών να είναι μεγαλύτερο από την τρίτη πλευρά.
- Έχουμε τέσσερις τύπους τριγώνων: ισόπλευρο, ισοσκελές, σκαληνό και ορθογώνιο.
- Υπάρχει τουλάχιστον μια περίπτωση ελέγχου που «πέρασε» για όλους τους βασικούς τύπους τριγώνου.
- Για να είναι ένα τρίγωνο ορθογώνιο πρέπει να ικανοποιείται το Πυθαγόρειο θεώρημα.
- Οι αρνητικοί αριθμοί δεν είναι αποδεκτοί ως πλευρές του τριγώνου.
- Το σύστημα δέχεται τιμές με εύρος από το 1 μέχρι το 999.
- Το σύστημα δέχεται αριθμούς με δεκαδικό μέρος.
- Η εφαρμογή δεν επικυρώνει αν τα δεδομένα εισόδου είναι αριθμοί ή όχι.

Ο διερευνητικός έλεγχος βασίζεται περισσότερο στην εμπειρία και στις ικανότητες του tester και προσφέρει άμεση πληροφορία για το αντικείμενο υπό έλεγχο, αλλά δεν αρκεί για ένα ολοκληρωμένο έλεγχο λογισμικού. Για το λόγο αυτό θα

ακολουθήσουμε μια πιο φορμαλιστική μέθοδο ελέγχου από αυτές που αναλύσαμε στο κεφάλαιο 2.

Ο έλεγχος της εφαρμογής των τριγώνων είναι κυρίως έλεγχος των δεδομένων εισόδου, οπότε κατάλληλη black box μέθοδος είναι αυτή του διαχωρισμού σε κλάσεις ισοδυναμίας σε συνδυασμό με την ανάλυση οριακών τιμών. Αρχικά θα διαχωρίσουμε τα δεδομένα εισόδου σε έγκυρα και μη έγκυρα, δηλαδή θα διαχωρίσουμε τα δεδομένα σε αυτά που παράγουν τρίγωνα και σε δεδομένα που δεν παράγουν, στη συνέχεια η κλάση των έγκυρων θα διαχωριστεί για κάθε τύπο τριγώνου και η κλάση των μη έγκυρων δεδομένων σε αριθμητικά και μη αριθμητικά δεδομένα. Στη συνέχεια θα κάνουμε μια ανάλυση οριακών τιμών για να προσδιορίσουμε τις ακρότατες τιμές των δεδομένων εισόδου. Επομένως, σύμφωνα με τη μέθοδο διαχωρισμού σε κλάσεις ισοδυναμίας τα δεδομένα εισόδου s1, s2, s3 που αντιστοιχούν στις πλευρές Side 1, Side 2 και Side 3 της εφαρμογής υπό έλεγχο διαχωρίζονται σε:

- **Έγκυρα δεδομένα εισόδου**

  - Τα δεδομένα εισόδου s1, s2, s3 σχηματίζουν ισόπλευρο τρίγωνο.
  - Τα δεδομένα εισόδου s1, s2, s3 σχηματίζουν ισοσκελές τρίγωνο.
    - με s1 = s2
    - με s2 = s3
    - με s1 = s3
  - Τα δεδομένα εισόδου s1, s2, s3 σχηματίζουν σκαληνό τρίγωνο.
    - Οξυγώνιο
    - Αμβλυγώνιο
  - Τα δεδομένα εισόδου s1, s2, s3 σχηματίζουν ορθογώνιο τρίγωνο.
    - με s1 υποτείνουσα
    - με s2 υποτείνουσα
    - με s3 υποτείνουσα

- **Μη έγκυρα δεδομένα εισόδου**

  - Το δεδομένο εισόδου s1 είναι **μη αριθμητικό** και είναι:
    - Γράμμα αλφαβήτα
      - Κεφαλαίο γράμμα (π.χ. Α).
      - Μικρό γράμμα (π.χ. α).
    - Αλφαριθμητικό (π.χ. 9Α9).
    - Επιστημονικός συμβολισμός (π.χ. $1.3^{E}6$).
    - Ειδικού χαρακτήρα (π.χ. ? , $).
    - Χαρακτήρας με ειδική σημασία για το περιβάλλον του συστήματος (π.χ. Enter, Space, Escape, Backspace κ.ά.).

  - Το δεδομένο εισόδου είναι s1 **αριθμητικό** και είναι:
    - Το μηδέν.

- Αρνητικός αριθμός (π.χ. -1).
- Ακρότατη τιμές (1 και 999).

Για τις ακρότατες τιμές των δεδομένων εισόδου θα εκτελέσουμε ανάλυση οριακών τιμών. Σκοπός μας είναι να βεβαιωθούμε ότι η εφαρμογή διαχειρίζεται σωστά τα πολύ μεγάλα τρίγωνα λόγω έλλειψης περιορισμού άνω ορίου στα δεδομένα εισόδου. Συνεπώς δεν μας αφορά η κάτω οριακή τιμή αλλά και ούτε και οι γειτονικές τιμές των κλάσεων, αφού το μηδέν ξέρουμε ότι δεν είναι επιτρεπτό και το άνω όριο τέθηκε από εμάς στις υποθέσεις. Συνεπώς πρέπει να ελέγξουμε τις παρακάτω οριακές τιμές:

- s1 = s2 = s3 = 999 για το μεγαλύτερο ισόπλευρο τρίγωνο.
- s1 = s2 = 999 και s3 = 998 για να εξετάσουμε πώς διαχειρίζεται το σύστημα τις μεγάλες τιμές και αν διακρίνει ότι το τρίγωνο είναι ισοσκελές.
- s1 = s2 = 999 και s3 = 99 για να εξετάσουμε πώς διαχειρίζεται το σύστημα τιμές διαφορετικής τάξης και αν διακρίνει ότι το τρίγωνο είναι ισοσκελές.
- s1 = s2 = 999 και s3 = 1 για να εξετάσουμε πώς διαχειρίζεται το σύστημα το συνδυασμό των ακραίων τιμών και αν διακρίνει ότι το τρίγωνο είναι ισοσκελές.

Στις ερωτήσεις πόσες μη έγκυρες περιπτώσεις αρκούν για έναν έλεγχο, αν υπάρχουν μη έγκυρες περιπτώσεις ελέγχου που δεν ελέγχθηκαν παραπάνω ή αν οι περιπτώσεις ελέγχου που θέσαμε είναι πάρα πολλές (*overkill*), η απάντηση εξαρτάται από δύο παράγοντες: το ρίσκο που φέρει το αντικείμενο υπό έλεγχο, αν η λειτουργία του τριγώνου επηρεάζει σημαντικά και άλλα κομμάτια του συστήματος τότε το ρίσκο είναι υψηλό, και την πιθανότητα ο τελικός χρήστης να βάλει μη έγκυρες τιμές εισόδου είναι μεγάλη. Για παράδειγμα, η εφαρμογή των τριγώνων που εξετάζουμε δεν περιορίζει τον χρήστη στο άνω όριο των τιμών εισόδου, όμως η πιθανότητα ένας χρήστης να βάλει για πλευρές πολύ μεγάλες τιμές, όπως 999.999.999, είναι ιδιαίτερα απίθανη, οπότε δε χρειάζεται να ελεγχθεί για τόσο ακραίες τιμές και για αυτό επιλέχθηκε ως άνω όριο το 999 που αρκεί για τον έλεγχο των μεγάλων τριγώνων.

Η απάντηση συνεπώς είναι ότι ο κατάλληλος συνδυασμός έγκυρων και μη έγκυρων περιπτώσεων ελέγχου εξαρτάται από της ανάγκες του αντικειμένου υπό εξέταση. Εμείς θέλουμε να πετύχουμε υψηλή αξιοπιστία και σταθερότητα στο σύστημα μας οπότε θα ελέγξουμε αρκετές μη έγκυρες περιπτώσεις.

Ένας άλλος πολύ σημαντικός παράγοντας για την επιλογή των κατάλληλων μη έγκυρων περιπτώσεων ελέγχου είναι «πόσο» έλεγχο πρέπει να κάνουμε για τις τιμές εισόδου s2 και s3. Αν είχαμε στη διάθεση μας τον κώδικα της εφαρμογής, δηλαδή αν εκτελούσαμε white box έλεγχο, τότε θα γνωρίζαμε εάν ο προγραμματιστής ακολούθησε την ίδια λογική υλοποίησης για κάθε μία από τις τιμές εισόδου. Σε αυτή την περίπτωση δε θα ήταν αναγκαίο να επαναλάβουμε όλες τις περιπτώσεις ελέγχου και για τις τρεις πλευρές του τριγώνου. Αυτό δε σημαίνει άμεσα ότι εμείς που εκτελούμε black box έλεγχο πρέπει να το κάνουμε, αρκεί μια περίπτωση για την τιμή

εισόδου s2 και μια για την s3 για να ελέγξουμε αν ο προγραμματιστής έπραξε αναλόγως για όλες τις εισερχόμενες τιμές.

Μια ακόμα καλύτερη προσέγγιση είναι αντί να διεξάγουμε όλες τις μη έγκυρες περιπτώσεις ελέγχου στην s1 ακολουθούμενες από μία περίπτωση για την s2 και μια για την s3, να διαμοιράσουμε τις μη έγκυρες περιπτώσεις ισόποσα στις τρεις πλευρές. Αναλόγως, για τις έγκυρες περιπτώσεις ελέγχου, αν υποθέσουμε ισοδυναμία στον τρόπο υλοποίησης των τριών τιμών εισόδου s1, s2, s3 τότε η σειρά των δεδομένων εισόδου δεν μας απασχολεί. Για παράδειγμα, η τριάδα (3, 3, 2) που παράγει ισοσκελές τρίγωνο, το ίδιο συμβαίνει και με τις τριάδες (3, 2, 3) και (2, 3, 3). Η μόνη διαφορά στα παραπάνω τρίγωνα είναι η γραφική απεικόνιση τους. Στην περίπτωση της εφαρμογής μας, με δοκιμές παρατηρούμε ισοδυναμία όποτε δε χρειάζεται να επαναλάβουμε τις μη έγκυρες περιπτώσεις ελέγχου για όλες τις πλευρές. Όμως τις έγκυρες περιπτώσεις ελέγχου θα τις εκτελέσουμε επί τρία, διότι βασικό στοιχείο των έγκυρων δοκιμασιών της εφαρμογής είναι ο έλεγχος των γραφικών αναπαραστάσεων των τριγώνων.

Από την ανάλυση των απαιτήσεων και με την χρήση black box μεθόδων καταλήγουμε στο παρακάτω σύνολο 28 περιπτώσεων ελέγχου που αποτελεί το σενάριο ελέγχου (test suite) της εφαρμογής των τριγώνων:

Περίπτωση ελέγχου #	**Κατάσταση υπό έλεγχο**	**Δεδομένα εισόδου**			**Αναμενόμενο αποτέλεσμα**
		s1	**s2**	**s3**	
1	Ισόπλευρο τρίγωνο	5	5	5	Equilateral
2	Ισοσκελές τρίγωνο με s1 = s2	10	10	15	Isosceles
3	Ισοσκελές τρίγωνο με s2 = s3	10	15	10	Isosceles
4	Ισοσκελές τρίγωνο με s1 = s2	15	10	10	Isosceles
5	Οξυγώνιο	4	5	6	Scalene
6	Αμβλυγώνιο	2	5	6	Scalene
7	Ορθογώνιο με s1 υποτείνουσα	5	3	4	Right
8	Ορθογώνιο με s2 υποτείνουσα	3	5	4	Right
9	Ορθογώνιο με s3 υποτείνουσα	3	4	5	Right

10	Πλευρές με ένα δεκαδικό ψηφίο	1.1	1.9	1.5	Isosceles
11	Πλευρές με έξι δεκαδικά ψηφία	3.330050	3.329951	3.330050	Equilateral
12	Όλες οι πλευρές με τη μέγιστη ακρότατη τιμή	999	999	999	Equilateral
13	Δύο πλευρές με τη μέγιστη ακρότατη τιμή και η τρίτη με μια γειτονική	999	999	998	Isosceles
14	Δύο πλευρές με τη μέγιστη ακρότατη τιμή και η τρίτη άλλης τάξης	999	99	999	Isosceles
15	Δύο πλευρές με τη μέγιστη ακρότατη τιμή και η τρίτη με την ελάχιστη	1	999	999	Isosceles
16	Μια πλευρά με κεφαλαίο γράμμα	A	2	3	Invalid
17	Μια πλευρά με μικρό γράμμα	1	b	1	Invalid
18	Μια πλευρά με αλφαριθμητικό	3	3	3c	Invalid
19	Μια πλευρά με επιστημονικό συμβολισμό	1.2E6	1	1	Invalid
20	Μια πλευρά με ειδικό χαρακτήρα	3	@	6	Invalid
21	Η πλευρά s1 με 0	0	2	3	Invalid
22	Οι πλευρές s1 και s2 με 0	0	0	5	Invalid
23	Όλες οι πλευρές με 0	0	0	0	Invalid
24	Η πλευρά s2 με αρνητικό αριθμό	2	-2	2	Invalid

25	Οι πλευρές s2 και s3 με αρνητικό αριθμό	3	-1	-1	Invalid
26	Όλες οι πλευρές με αρνητικό αριθμό	-5	-5	-5	Invalid
27	Οι κορυφές του τριγώνου είναι στην ίδια ευθεία	1	2	3	Degenerate
28	Οι πλευρές δεν σχηματίζουν ποτέ τρίγωνο	1	2	4	Invalid

4.3. Αυτοματοποιημένος ελέγχος της εφαρμογής των τριγώνων

4.3.1 Επιλογή εργαλείου αυτοματοποίησης

Το σενάριο ελέγχου που καταλήξαμε μπορεί να εκτελεστεί μη αυτοματοποιημένα, δηλαδή απλά πηγαίνοντας στην εφαρμογή και συμπληρώνοντας όλες τις τιμές εισόδου που επιλέχθηκαν για την κάθε περίπτωση. Τότε, για το κάθε τέλος περίπτωσης ελέγχου το αποτέλεσμα του ελέγχου πρέπει να σημειώνεται και να συγκρίνεται με το αναμενόμενο αποτέλεσμα. Επίσης σε περίπτωση αναβάθμισης της εφαρμογής το σενάριο πρέπει να εκτελείται συστηματικά ως έλεγχος παλινδρόμησης. Παρόλο που η εφαρμογή υπό έλεγχο είναι μικρή και ο έλεγχος είναι πιο γρήγορος αν γίνει «χειρωνακτικά», επιλέγεται η αυτοματοποίηση για την δυνατότητα επανάληψης του ελέγχου.

Για τη δημιουργία και την εκτέλεση του αυτοματοποιημένου σεναρίου ελέγχου πρέπει να επιλεχθεί ένα εργαλείο κατάλληλο για την εφαρμογή. Στόχος του ελέγχου είναι η σωστή λειτουργία της εφαρμογής διαδικτύου, συνεπώς στόχος είναι η εύρεση ενός εργαλείου μεσαίας βαθμίδας για δοκιμασίες αποδοχής. Επίσης το αντικείμενο του ελέγχου είναι η δοκιμή των δεδομένων ελέγχου, οπότε ένα εργαλείο με βάση την τεχνική βάσει δεδομένων θα ήταν ίσως το καταλληλότερο. Εν τούτοις, και ένα εργαλείο τεχνικής βάσει λέξεων-κλειδιών, η οποία επιτρέπει τη δόμηση περιπτώσεων ελέγχου σε μορφή λέξεων-κλειδιών, μπορεί να καλύψει τις ανάγκες της εφαρμογής υπό έλεγχο. Επίσης, το εργαλείο πρέπει να δημιουργεί αυτόματα αναφορές αποτελεσμάτων των περιπτώσεων ελέγχου και να είναι γενικά εύκολο στη χρήση, έτσι ώστε να υπάρξει μεγαλύτερο κέρδος από την αυτοματοποίηση. Επίσης αναζητείται ανοιχτό ή ελεύθερο λογισμικό ώστε να επιτρέπεται η χρήση του στα

πλαίσια της διπλωματικής εργασίας. Από τα εργαλεία που παρουσιάσαμε στο υπόκεφάλαιο 3.3, επιλέχθηκε να χρησιμοποιηθεί το Robot Framework της τεχνικής βάσει λέξεων-κλειδιών, το οποίο παρέχει επίσης και λειτουργίες της τεχνικής βάσει δεδομένων.

4.3.2 Εκτέλεση αυτοματοποιημένου ελέγχου [25]

Το Robot Framework είναι υλοποιημένο σε Python, αλλά μπορεί να τρέξει και με Jython ή IronPython, που είναι συμβεβλημένες με πλατφόρμες της Java και .NET αντίστοιχα. Για την εγκατάσταση του εργαλείου επιλέχθηκε η εγκατάσταση της Python, και στη συνέχεια εγκαταστάθηκε το Robot Framework. Επειδή το εργαλείο που εξελέγει είναι για δοκιμασίες αποδοχής, χρειάζεται ένα επιπλέον εργαλείο ή μια βιβλιοθήκη του εργαλείου για την πλοήγηση του σεναρίου ελέγχου στην διεπαφή της εφαρμογής.

Το Robot είναι πλούσιο σε βιβλιοθήκες που παρέχουν έτοιμες λέξεις-κλειδιά, κάποιες από αυτές είναι άμεσα διαθέσιμες (standard libraries), κάποιες είναι *εξωτερικές* και πρέπει να εγκατασταθούν επιπλέον (*external libraries*) και άλλες μπορούν να φτιαχτούν από τον χρήστη (custom libraries). Η βιβλιοθήκη Selenium2Library είναι εξωτερική βιβλιοθήκη και παρέχει την πλοήγηση του γραφικού περιβάλλοντος που χρειαζόμαστε.

Το εργαλείο παρέχει επιπλέον ένα ειδικό περιβάλλον για τη συγγραφή των σεναρίων ελέγχου, το RIDE (Robot integrated Development Environment), το οποίο πρέπει να εγκατασταθεί επιπλέον. Το πλαίσιο δέχεται όμως και άλλου είδους αρχεία με τις περιπτώσεις ελέγχου, όπως .txt, .html. και .tsv, τα οποία γράφονται ξεχωριστά και εκτελούνται άμεσα. Για να εκτελεστεί ένα από αυτά του είδους αρχεία στο Robot αρκεί η κατάλληλη εντολή στη γραμμή εντολών του υπολογιστή. Στην παρούσα εργασία επιλέχτηκε να γραφτούν τα σενάρια ελέγχου σε μορφή απλού κειμένου.

Τα σενάρια ελέγχου στο Robot Framework γράφονται με λέξεις-κλειδιά τα οποία ορίζονται αναλόγως μέσα στο σενάριο. Εκτός από τις λέξεις-κλειδιά του χρήστη (user keywords), μέσα στο σενάριο πρέπει να οριστούν και οι λέξεις-κλειδιά της βιβλιοθήκης. Οι πρώτες ορίζονται κάτω από τον τίτλο `*** Settings ***`, ενώ οι δεύτερες κάτω από το `*** Keywords ***`. Οι λέξεις-κλειδιά της βιβλιοθήκης στην ουσία καλούν τις βιβλιοθήκες, είτε αυτές είναι του χρήστη είτε εξωτερικές, π.χ. `Library Selenium2Library`. Επιπλέον εκεί προσδιορίζονται λέξεις-κλειδιά που εκτελούνται πριν ή μετά από κάθε περίπτωση ελέγχου ή στην έναρξη και το τέλος ολόκληρου του σεναρίου π.χ. με τη χρήση των εντολών `Suite Setup` και `Suite Teardown`.

Για το σενάριο ελέγχου των τριγώνων χρειάστηκε να οριστούν πέντε λέξεις-κλειδιά με τη βοήθεια μεταβλητών:

- **Enter**: καλεί εντολές της βιβλιοθήκης Selenium2Library για το άνοιγμα του πλοηγητή διαδικτύου στη διεύθυνση ιστοσελίδας που είναι η εφαρμογή υπό εξέταση.
- **Exit**: κλείνει τον πλοηγητή διαδικτύου.
- **Input values**: αναθέτει τα δεδομένα εισόδου στις πλευρές του τριγώνου μέσω των μεταβλητών `${side1}, ${side2}, ${side3}`.
- **Verify triangle is identified as:** ορίζει τη μεταβλητή `${actual}` ως το λεκτικό που επιστρέφει η εφαρμογή στην οθόνη για τύπο τριγώνου και τη μεταβλητή `${expected}` ως το αναμενόμενο τύπο τριγώνου που θα σχηματίσουν τα δεδομένα εισόδου. Οι δύο μεταβλητές πρέπει να είναι ίσες μεταξύ τους.
- **Verify triangle is drawn inside canvas:** ελέγχει αν το σχήμα του τριγώνου είναι μέσα στο πλαίσιο. Ορίζει τις μεταβλητές `${coord_as_string},` η οποία παίρνει σαν εισόδους τις συντεταγμένες των κορυφών του τριγώνου, και `${in_range}`, η οποία καλεί την μέθοδο `CoordinateCheck` [10] και επιτστρέφει True ή False, αν το σχήμα είναι μέσα στο πλαίσιο ή όχι αντίστοιχα.

Αφού οριστούν οι λέξεις-κλειδιά, υπάρχουν όλα τα απαραίτητα εργαλεία για να γραφτούν οι περιπτώσεις ελέγχου, οι οποίες ορίζονται συνήθως στην αρχή του σεναρίου κάτω από τον τίτλο `*** Test Cases ***`. Οι περιπτώσεις ελέγχου χωρίζονται μεταξύ τους με μια κενή σειρά. Σε κάθε περίπτωση ελέγχου, η πρώτη σειρά της αποτελεί τον τίτλο της περίπτωσης και οι υπόλοιπες καλούν από μια λέξη-κλειδί παρέχοντας τις τιμές των μεταβλητών που έχουν οριστεί. Για παράδειγμα, η πρώτη περίπτωση ελέγχου από το σενάριο ελέγχου που επιλέχθηκε παραπάνω και ελέγχει ένα απλό ισόπλευρο τρίγωνο, έχει την μορφή:

```
Handles a simple Equilateral

Input values 5, 5, 5

Verify triangle is identified as "Equilateral"
```

Αφού γραφτούν όλες οι περιπτώσεις ελέγχου σε αποδεκτή μορφή σεναρίου του Robot, το σενάριο μπορεί να αποθηκευτεί και είναι έτοιμο για εκτέλεση. Η εκτέλεση του είναι ιδιαίτερα απλή, ανοίγοντας τη γραμμή εντολών του υπολογιστή αρκεί να εισάγουμε την εντολή:

```
pybot test_triangles.txt
```

Με την εντολή αυτή εκτελείται αυτόματα το σενάριο. Κατά την λήξη του σεναρίου, το εργαλείο παράγει άμεσα τρία αρχεία: την αναφορά των αποτελεσμάτων του ελέγχου (report.html), το αρχείο με όλες τις καταστάσεις που πέρασε το σενάριο (logs.html) και όλα τα δεδομένα εξόδου σε μορφή xml αρχείου (output.xml).

[10] Η μέθοδος αυτή υλοποιήθηκε σε Python για τις ανάγκες ελέγχου του σχεδίου του τριγώνου που σχηματίζεται στην οθόνη

4.3.3 Αποτελέσματα αυτοματοποιημένου ελέγχου

Τα εξαγόμενα αποτελέσματα του αυτοματοποιημένου ελέγχου από το Robot Framework αναφέρουν ότι οι 22 από τις 28 περιπτώσεις «πέρασαν τον έλεγχο», ενώ οι υπόλοιπες απέτυχαν.

Οι 6 περιπτώσεις που απέτυχαν πρέπει να εξεταστούν περισσότερο εις βάθος για να αποσαφηνιστεί αν η αποτυχία οφείλεται σε ελάττωμα της εφαρμογής ή σε λάθος του σεναρίου ελέγχου, του περιβάλλοντος ελέγχου ή της συνδεσιμότητας μεταξύ των δύο. Αναλυτικά οι περιπτώσεις που δεν πέρασαν τον έλεγχο σύμφωνα με τον τίτλο της περίπτωσης ελέγχου στο σενάριο είναι:

- Handles an Isosceles when all sides have one decimal digit

Από το ελάττωμα αυτό παρατηρούμε ότι το σύστημα δεν επεξεργάζεται σωστά την στρογγυλοποίηση. Θεωρεί ότι οι αριθμοί 1.1 και 1.9 στρογγυλοποιούνται και οι δύο στο 1, οπότε το τρίγωνο είναι ισόπλευρο.

- Handles Invalid with an upper-case letter side
- Handles Invalid with a lower-case letter side
- Handles Invalid with a alphanumerical side
- Handles Invalid with a special character side

Και οι τέσσερις περιπτώσεις ελέγχου ανήκουν στην ίδια κλάση, αφού τα τέσσερα ελλατώματα που ανιχθεύθηκαν έχουν κοινή ρίζα. Τα αποτελέσματα του ελέγχου υποδηλώνουν ότι το σύστημα δεν κάνει επικύρωση των δεδομένων εισόδων με αποτέλεσμα να δέχεται τιμές γράμματα και ειδικούς χαρακτήρες.

- Handles Invalid with all sides zero

Η περίπτωση όλες οι πλευρές να είναι μηδέν δεν «περνάει» τον έλεγχο γιατί επιστρέφει “Equilateral” αντί για “Invalid”. Το σύστημα δεν επικυρώνει την περίπτωση του μηδενός, επομένως οι περιπτώσεις ελέγχου 21 και 22 «πέρασαν» τον έλεγχο, δηλαδή επέστρεψαν “Invalid”, λόγω της μη ικανοποίησης της τριγωνικής ανισότητας και όχι λόγω του μηδενός.

Συνολικά, παρατηρούμε ότι το αυτοματοποιημένο σενάριο ελέγχου βρήκε τρεις βασικές κλάσεις ελαττωμάτων, αυτές της στρογγυλοποίησης, υπερχείλισης και απουσίας επικύρωσης δεδομένων εισόδου. Επίσης, παρατηρούμε ότι το σύστημα δεν επιστρέπει κανένα μήνυμα λάθους στο χρήστη ώστε να τον ενημερώνει, συμπεραίνοντας ότι ο παράγοντας χρησικανότητας δεν είναι ικανοποιητικός και με ανάλογες προδιαγραφές αυτό θα μπορούσε να υποδηλώνει άλλο ένα ελάττωμα.

5

Επίλογος

5.1. Σύνοψη και συμπεράσματα

Η συμβολή του ελέγχου λογισμικού στην ποιότητα ενός προϊόντος έχει αναδείξει τον έλεγχο λογισμικού σε απαραίτητο και αναπόσπαστο κομμάτι του κύκλου ζωής του. Η διαδικασία του ελέγχου αφενός εξετάζει την ορθότητα του προϊόντος σε κάθε φάση του κύκλου σύμφωνα με τη διαδικασία της επαλήθευσης, και αφετέρου εκτιμάει κατά πόσο το λογισμικό ικανοποιεί τις απαιτήσεις που έχουν τεθεί με την διαδικασία της επικύρωσης.

Η αυτοματοποίηση του ελέγχου λογισμικού σε όλα τα επίπεδα παρέχει τρία βασικά πλεονεκτήματα:

- Καλύτερη κάλυψη για ανεύρεση ελαττωμάτων και μείωση κόστους αποτυχίας
- Μείωση χρόνου εκτέλεσης, και αντιστοίχως κόστους, και παράδοσης στον πελάτη μέσω της επανάληψης
- Πλεονέκτημα για παραγωγικότητα

Για να μπορέσει η αυτοματοποίηση να αποφέρει τα μέγιστα στους παραπάνω τομείς θα πρέπει να εφαρμόζεται με γνώμονα την επικινδυνότητα της κάθε εφαρμογής και τους διαθέσιμους πόρους. Υπάρχουν περιπτώσεις που ο αυτοματοποιημένος έλεγχος δε συνίσταται, όπως για παράδειγμα σε ασταθείς εφαρμογές λόγω κακής αρχιτεκτονικής ή όταν η αναμενόμενη συμπεριφορά του αντικειμένου υπό έλεγχο δεν μπορεί να προβλεφθεί.

Από την άλλη πλευρά, τα σενάρια αυτοματοποιημένου ελέγχου, προερχόμενα είτε από τεχνική βάσει λέξεων-κλειδιών είτε από άλλη τεχνική αυτοματοποίησης, είναι και αυτά προϊόντα λογισμικού και χρήζουν ελέγχου. Αυτό το παράδοξο δεν πρέπει να παραλείπεται και πρωταρχικός στόχος του ειδικού σε αυτοματοποιημένο έλεγχο οφείλει να είναι ο έλεγχος των σεναρίων ελέγχου πριν από κάθε νέα έκδοση λογισμικού εξασφαλίζοντας υψηλό βαθμό αξιοπιστίας του ελέγχου.

Ο έλεγχος αποδοχής της εφαρμογής των τριγώνων που πραγματοποιήθηκε στην παρούσα εργασία παρουσιάζει αναλυτικά τη διαδικασία σχεδιασμού και κατασκευής περιπτώσεων ελέγχου αναδεικνύοντας τη σημασία της επιλογής περιπτώσεων

ελέγχου. Παρόλο που η εκτέλεση του ελέγχου λογισμικού γίνεται με τεχνική αυτοματοποιημένου ελέγχου, η επιλογή των περιπτώσεων ελέγχου προς εκτέλεση έγινε μη αυτοματοποιημένα, με τη χρήση ανάλυσης και φορμαλιστικών μεθόδων. Συνεπώς, η διαδικασία επιλογής περιπτώσεων σε αυτή την περίπτωση είναι επιρρεπής σε λάθη, καθώς βασίζεται στις ικανότητες του tester και στην κατανόηση του αντικειμένου υπό έλεγχο ενώ συχνά δεν παρέχει εγγυήσεις ότι το σύστημα έχει ελεγχθεί επαρκώς.

Η εκτέλεση του ελέγχου αποδοχής της εφαρμογής τριγώνων με το Robot Framework απέδειξε ότι αυτοματοποίηση δε σημαίνει απαραίτητα κώδικας, αφού χρειάστηκε επιπλέον η υλοποίηση μιας μόνο μεθόδου. Ταυτόχρονα ανέδειξε την ευκολία κατασκευής και εκτέλεσης σεναρίων ελέγχου από το συγκεκριμένο πλαίσιο ελέγχου. Οι περιπτώσεις ελέγχου εκφράζονται σε φυσική γλώσσα, έτσι ακόμα και κάποιος που δε γνωρίζει το συγκεκριμένο εργαλείο μπορεί να κατανοήσει το αντικείμενο που ελέγχουν. Επίσης, η αυτόματη δημιουργία αναφορών με περιγραφική παρουσίαση των αποτελεσμάτων ελέγχου, σε μορφή που μπορούν να επεξεργαστούν οι testers και να αποθηκευτούν για μελλοντική χρήση, συμβάλει σημαντικά στην εξαγωγή συμπερασμάτων για τον έλεγχο λογισμικού.

Συνοψίζοντας τα αποτελέσματα του ελέγχου που πραγματοποιήθηκε, το σύνολο των 28 περιπτώσεων το οποίο επιλέχθηκε για τον έλεγχο της εφαρμογής είναι αποτέλεσμα της ανάλυσης των απαιτήσεων και των black box μεθόδων διαχωρισμού σε κλάσεις ισοδυναμίας και ανάλυσης οριακών τιμών. Τα 6 ελαττώματα που ανιχνεύτηκαν από το σενάριο ελέγχου θεωρούνται ότι καλύπτουν τις πιο σημαντικές κλάσεις ελαττωμάτων που παράγουν σφάλματα, χωρίς αυτό να σημαίνει ότι δεν υπάρχουν και άλλα.

5.2. Μελλοντικές προεκτάσεις

Η παρούσα εργασία παρέχει μια καλή βάση πάνω στην οποία μπορεί να στηριχτεί περαιτέρω έρευνα στο πεδίο του αυτοματοποιημένου ελέγχου σε συνδυασμό με τη *διαχείριση επικινδυνότητας του λογισμικού (software risk management)*, όπως αυτή ορίζεται στις πιο δημοφιλείς μεθοδολογίες ανάπτυξης. Μέσα στο πλαίσιο αυτό, θα μπορούσε να πραγματοποιηθεί σύγκριση διαφορετικών τεχνικών αυτοματοποίησης, και αντίστοιχα εργαλείων, ως προς την επικινδυνότητα λογισμικού με τη χρήση κατάλληλων μετρικών για τους κινδύνους αποτυχίας. Σημαντική κρίνεται επίσης και η έρευνα πάνω στην επιτευξη κατάλληλου επίπεδου αξιοπιστίας της τεχνικής αυτοματοποίησης μέσω του ορισμού *κριτηρίων κάλυψης (coverage criterion)* του ελέγχου, και εφαρμογής τους στις τεχνικές αυτοματοποιημένου ελέγχου.

Μια άλλη περιοχή που θα μπορούσε να εστιάσει η μελλοντική έρευνα είναι το πρόβλημα της επιλογής περιπτώσεων ελέγχου με αυτοματοποιημένο τρόπο. Ο έλεγχος βάσει μοντέλου (model-based testing), δηλαδή η δημιουργία ενός μοντέλου

που αναπαριστά την επιθυμητή συμπεριφορά του αντικειμένου υπό έλεγχο, επιτρέπει την αλγοριθμική παραγωγή περιπτώσεων ελέγχου εντελώς αυτοματοποιημένα.

Τέλος, όλες οι εξελίξεις στον τομέα του ελέγχου λογισμικού, αυτοματοποιημένου και μη, οδήγησαν σε νέες μεθοδολογίες που προωθούν τον έλεγχο να προηγείται του σχεδιασμού και την υλοποίησης κώδικα. Η *ανάπτυξη λογισμικού βάσει ελέγχου TDD (Test-driven development)* είναι μία τέτοια μέθοδολογία που βασίζεται σε μικρούς κύκλους όπου ο προγραμματιστής πρώτα γράφει μια αυτοματοποιημένη περίπτωση ελέγχου μονάδας, η οποία αποτυγχάνει, και μετά υλοποιεί τον κώδικα με σκοπό να περάσει τον έλεγχο. Μια μελλόντική προέκταση της εργασίας θα μπορούσε να είναι στο πεδίο της *ανάπτυξης λογισμικού βάσει ελέγχου αποδοχής ATDD (Acceptance Test-driven development)* με την χρήση πάλι του εργαλείου Robot Framework.

6

Βιβλιογραφία

[1] A. Spillner, T. Linz, H. Schaefer, Software Testing Foundations: A Certified Guide for the Certified Tester Exam, February 4, 2011

[2] D. Graham, E. Veenendaal, I. Evans, R. Black, Foundations of Software Testing, International Software Testing Qualifications Board (ISTQB) Certification

[3] E. Veenendaal & «Glossary Working Party», Standard glossary of terms used in Software Testing, International Software Testing Qualifications Board (ISTQB)

[4] A.B. AL Badareen, M. H. Selamat, M. A. Jabar, J. Din, S. Turaev, Software Quality Models: A Comparative Study, In Proc. 2nd Intl. Conference Software Engineering and Computer Systems, (ICSECS 2011), Kuantan, Pahang, Malaysia, June 27-29, 2011

[5] R. Fitzpatrick, Software Quality: Definitions and Strategic Issues, Staffordshire University, School of Computing report, April 1996

[6] G. Bath, P. Jorgensen, J. Mitchell, Certified Tester Advanced Level Syllabus Technical Test Analyst, International Software Testing Qualifications Board (ISTQB), 2010-2012

[7] J. Hinz, M. Gijsen, Fifth Generation Scriptless and Advanced Test Automation Technologies, TESTars Test Competence Centre

[8] J. Kent, Test Automation: From Record/Playback to Frameworks V1.0 , ©Simply Testing Ltd

[9] D. Hoffman, Test Automation Architectures: Planning for Test Automation, © 1999, Software Quality Methods, LLC, International Quality Week 1999

[10] G.Ukkuru, Test Automation Best Practices, November 7, 2013

[11] K.Martin, Automated testing – The Advantages and Disadvantages (QA for Software Development), June 20, 2012

[12] B. Marick, When should a test be automated?, Reliable Software Technlogies, 1998

[13] S. Hebbar, A Structured Approach to Software Test Automation, Adobe Systems India Pvt. Ltd., Bangalore, 25 December 2008

[14] L. G. Hayes, Automated testing handbook, Software Testing Inst; 2nd edition (March 1, 2004)

[15] G. J Myers, Revised and Updated by T. Badgett, T. M. Thomas, C. Sandler, The Art of Software Testing, 2nd Edition, John Wiley & Sons, Inc., Hoboken, New Jersey, 2004

[16] R. Collard, An introduction to Software Test Case Design, Workshop on Teaching Software Testing (WTST) 3, available at http://www.testingeducation.org/conference/wtst3_collard1.pdf , 2004

[17] Stevenson, Oxford Dictionary of English, Oxford University Press, USA, 3rd Revised edition edition, August 1, 2010

[18] Merriam-Webster, Collegiate Dictionary, Merriam-Webster, Inc., 11th edition, July 30, 2003

[19] R. Potts, Euclid's Elements of Geometry, HardPress Publishing, August 1, 2012

Ηλεκτρονική Βιβλιογραφία

[20] http://testobsessed.com/2007/03/testing-triangles-a-classic-exercise-updated-for-the-web/

[21] http://www.velocitypartners.net/blog/2014/01/28/agile-testing-the-agile-test-automation-pyramid/

[22] http://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[23] http://www.slideshare.net/pekkaklarck/introduction-to-test-automation

[24] http://junit.org/

[25] http://www.nunit.org/

[26] http://robotframework.org/

[27] http://www.fitnesse.org/

[28] https://cukes.info/

[29] http://www.seleniumhq.org/

[30] http://watir.com/

[31] http://www8.hp.com/us/en/software-solutions/unified-functional-automated-testing/

[32] http://www-03.ibm.com/software/products/en/functional

[33] http://galenframework.com/
[34] http://jmeter.apache.org/
[35] http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/

Παράρτημα Α

1. test_triangles.txt

```
*** Test Cases ***

Handles a simple Equilateral
    Input values 5, 5, 5
    Verify triangle is identified as "Equilateral"
    Verify triangle is drawn inside canvas

Handles a simple Isosceles with first and second edges equal
    Input values 10, 10, 15
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas

Handles a simple Isosceles with second and third edges equal
    Input values 15, 10, 10
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas

Handles a simple Isosceles with first and third edges equal
    Input values 10, 15, 10
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas

Handles a simple Acute Scalene
    Input values 4, 5, 6
    Verify triangle is identified as "Scalene"
    Verify triangle is drawn inside canvas

Handles a simple Obtuse Scalene
    Input values 2, 5, 6
    Verify triangle is identified as "Scalene"
    Verify triangle is drawn inside canvas

Handles a simple Right with first side as hypotenuse
    Input values 5, 3, 4
    Verify triangle is identified as "Right"
    Verify triangle is drawn inside canvas

Handles a simple Right with second side as hypotenuse
    Input values 3, 5, 4
    Verify triangle is identified as "Right"
    Verify triangle is drawn inside canvas

Handles a simple Right with third side as hypotenuse
    Input values 3, 4, 5
    Verify triangle is identified as "Right"
    Verify triangle is drawn inside canvas

Handles an Isosceles when all sides have one decimal digit
    Input values 1.1, 1.9, 1.5
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas
```

```
Handles an Equilateral when all sides have six decimal digits
    Input values 3.330050, 3.329951, 3.330050
    Verify triangle is identified as "Equilateral"
    Verify triangle is drawn inside canvas

Handles an Equilateral when all sides have the maximum boundary
    Input values 999, 999, 999
    Verify triangle is identified as "Equilateral"
    Verify triangle is drawn inside canvas

Handles an Isosceles when two sides have the maximum boundary and the
other a close value
    Input values 999, 999, 998
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas

Handles an Isosceles when two sides have the maximum boundary and the
other a much smaller value
    Input values 999, 999, 10
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas

Handles an Isosceles when two sides have the maximum boundary and the
other the minimum boundary
    Input values 1, 999, 999
    Verify triangle is identified as "Isosceles"
    Verify triangle is drawn inside canvas

Handles Invalid with a upper-case letter side
    [Tags]  failing
    Input values A, 2, 3
    Verify triangle is identified as "Invalid"

Handles Invalid with a lower-case letter side
    [Tags]  failing
    Input values 1, b, 1
    Verify triangle is identified as "Invalid"

Handles Invalid with a alphanumerical side
    [Tags]  failing
    Input values 3, 3, 3c
    Verify triangle is identified as "Invalid"

Handles Invalid with a scientific notation side
    [Tags]  failing
    Input values 1.2E6, 1, 1
    Verify triangle is identified as "Invalid"

Handles Invalid with a special character side
    [Tags]  failing
    Input values 3, @, 3
    Verify triangle is identified as "Invalid"

Handles Invalid with a zero side
    [Tags]  failing
    Input values 0, 2, 3
    Verify triangle is identified as "Invalid"

Handles Invalid with two zero sides
    [Tags]  failing
    Input values 0, 0, 5
```

```
    Verify triangle is identified as "Invalid"

Handles Invalid with all sides zero
    [Tags]  failing
    Input values 0, 0, 0
    Verify triangle is identified as "Invalid"

Handles Invalid with a negative side
    [Tags]  failing
    Input values 2, -2, 2
    Verify triangle is identified as "Invalid"

Handles Invalid with two negative sides
    [Tags]  failing
    Input values 3, -1, -1
    Verify triangle is identified as "Invalid"

Handles Invalid with all sides negative
    [Tags]  failing
    Input values -5, -5, -5
    Verify triangle is identified as "Invalid"

Handles Invalid where the longest side is equal to the sum of the
other two sides
    [Tags]  failing
    Input values 1, 2, 3
    Verify triangle is identified as "Degenerate"

Handles Invalid where the longest side is longer than the sum of the
other two sides
    [Tags]  failing
    Input values 1, 2, 4
    Verify triangle is identified as "Invalid"



*** Settings ***
Library  Selenium2Library
Library  CoordinateCheck
Suite Setup         Enter
Suite Teardown        Exit

*** Keywords ***
Enter
    Open Browser  http://practice.agilistry.com/triangle
    Set Selenium Speed  0.75
    Maximize Browser Window

Exit
    Close Browser

Input values ${side1}, ${side2}, ${side3}
    Input Text  triangle_side1  ${side1}
    Input Text  triangle_side2  ${side2}
    Input Text  triangle_side3  ${side3}
    Click Element   //div[@id='triangle_frame']

Verify triangle is identified as "${expected}"
    ${actual} =  Get Text  triangle_type
    Should Be Equal  ${actual}  ${expected}
```

```
Verify triangle is drawn inside canvas
    ${coord_as_string} =  Get Text
//div[@id='triangles_list']/div[contains(@class,
'triangle_row')][1]/div[contains(@class, 'triangle_data_cell')][5]
    ${in_range} =  coordsInRange  ${coord_as_string}
    Should Be True  ${in_range}  Coordinates out of range
${coord_as_string}
```

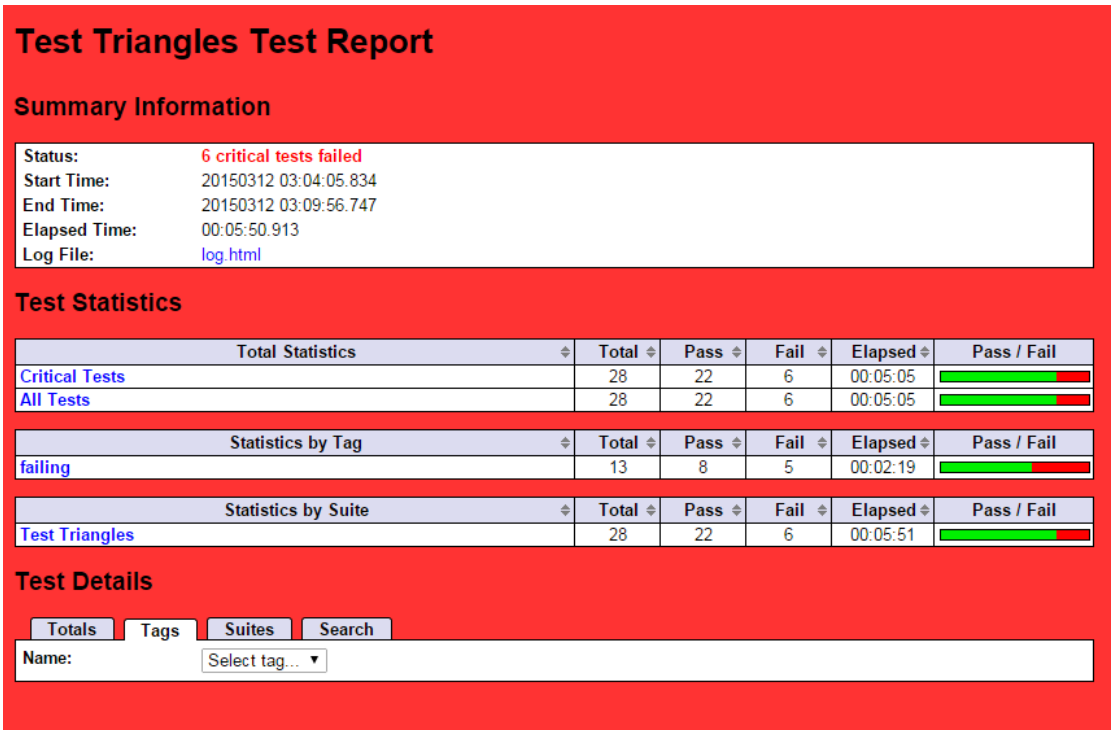
2. CoordinateCheck.py

```
import re

class CoordinateCheck:

    def coordsInRange(self,sString):
        import re
        regex = re.compile("(-*[0-9]+),(-*[0-9]+)\) \((-*[0-9]+),(-
*[0-9]+)\) \((-*[0-9]+),(-*[0-9]+)")
        r = regex.search(sString)
        coords_valid = True
        for i in range(6):
            if r.group(i+1) < 0 or r.group(i+1) > 200:
                coords_valid = False
        return coords_valid
```

Παράρτημα Β

1. test_triangles_report.html

Test Statistics:



Test Triangles Test Report

Summary Information

Status:	6 critical tests failed
Start Time:	20150312 03:04:05.834
End Time:	20150312 03:09:56.747
Elapsed Time:	00:05:50.913
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	28	22	6	00:05:05	
All Tests	28	22	6	00:05:05	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
failing	13	8	5	00:02:19	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Test Triangles	28	22	6	00:05:51	

Test Details

Totals | Tags | Suites | Search

Name: Select tag...

Test Details:

Name	Tags	Crit.	Status	Message	Elapsed
TestTriangles.Handles Invalid with a alphanumerical side	failing	yes	FAIL	Isosceles != Invalid	00:00:10.644
TestTriangles.Handles Invalid with a lower-case letter side	failing	yes	FAIL	Isosceles != Invalid	00:00:10.883
TestTriangles.Handles Invalid with a special character side	failing	yes	FAIL	Isosceles != Invalid	00:00:10.569
TestTriangles.Handles Invalid with a upper-case letter side	failing	yes	FAIL	Scalene != Invalid	00:00:10.611
TestTriangles.Handles Invalid with all sides zero	failing	yes	FAIL	Degenerate != Invalid	00:00:10.684
TestTriangles.Handles an Isosceles when all sides have one decimal digit		yes	FAIL	Equilateral != Isosceles	00:00:11.030
TestTriangles.Handles Invalid where the longest side is equal to the sum of the other two sides	failing	yes	PASS		00:00:10.592
TestTriangles.Handles Invalid where the longest side is longer than the sum of the other two sides	failing	yes	PASS		00:00:10.629
TestTriangles.Handles Invalid with a negative side	failing	yes	PASS		00:00:10.587
TestTriangles.Handles Invalid with a scientific notation side	failing	yes	PASS		00:00:10.630
TestTriangles.Handles Invalid with a zero side	failing	yes	PASS		00:00:10.579
TestTriangles.Handles Invalid with all sides negative	failing	yes	PASS		00:00:10.723
TestTriangles.Handles Invalid with two negative sides	failing	yes	PASS		00:00:10.785
TestTriangles.Handles Invalid with two zero sides	failing	yes	PASS		00:00:10.695
TestTriangles.Handles a simple Acute Scalene		yes	PASS		00:00:11.333
TestTriangles.Handles a simple Equilateral		yes	PASS		00:00:12.839
TestTriangles.Handles a simple Isosceles with first and second edges equal		yes	PASS		00:00:11.084
TestTriangles.Handles a simple Isosceles with first and third edges equal		yes	PASS		00:00:11.027
TestTriangles.Handles a simple Isosceles with second and third edges equal		yes	PASS		00:00:11.042
TestTriangles.Handles a simple Obtuse Scalene		yes	PASS		00:00:10.799
TestTriangles.Handles a simple Right with first side as hypotenuse		yes	PASS		00:00:10.859
TestTriangles.Handles a simple Right with second side as hypotenuse		yes	PASS		00:00:10.907
TestTriangles.Handles a simple Right with third side as hypotenuse		yes	PASS		00:00:10.937
TestTriangles.Handles an Equilateral when all sides have six decimal digits		yes	PASS		00:00:11.232
TestTriangles.Handles an Equilateral when all sides have the maximum boundary		yes	PASS		00:00:10.921
TestTriangles.Handles an Isosceles when two sides have the maximum boundary and the other a close value		yes	PASS		00:00:10.818
TestTriangles.Handles an Isosceles when two sides have the maximum boundary and the other a much smaller value		yes	PASS		00:00:10.582
TestTriangles.Handles an Isosceles when two sides have the maximum boundary and the other the minimum boundary		yes	PASS		00:00:10.674