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

Πτυχιακή εργασία

Ευέλικτοι Έλεγχοι Λογισμικού για JAVA - προγραμματιστές

Του φοιτητή
Μπαμπάτση Χρυσοβαλάντη
Αρ. Μητρώου 1574

Επιβλέπων καθηγητής
Σφέτσος Παναγιώτης

Θεσσαλονίκη 2013

Πρόλογος

Από την αρχή της γέννησης του πρρογραμματισμού και γενικότερα της ανάπτυξης Λογισμικού έγινε αντιληπτό ότι πέρα από τις τεχνολογίες που χρησιμοποιούνται για την επίτευξη αυτού του σκοπού μεγάλο ρόλο παίζει και ο τρόπος που χρησιμοποιούνται. Και αυτό διότι εκτός του ότι ο τρόπος που χρησιμοποιείται μια τεχνολογία μπορεί να επηρεάσει το βαθμό στον οποίο εκμεταλλεύεται αυτά που προσφέρει, είναι ικανός να διαφοροποιήσει και την απόδοση των εργαζομένων σε μια επιχείρηση. Ακόμα μπορεί και να θέσει εκτός στόχου μια ολόκληρη προσπάθεια αποπροσανατολίζοντας τα μέλη μιας ομάδας ή θέτοντας λανθασμένες προτεραιότητες. Αμέσως, λοιπόν δημιουργήθηκαν τρόποι ανάπτυξης που κατόπιν πήραν τη μορφή διαδικασιών και επισημοποιήθηκαν ως Μεθοδολογίες. Οι Μεθοδολογίες Ανάπτυξης σήμερα είναι εκτός από πολλές και τόσο διαφορετικές μεταξύ τους, που υπάρχουν ακόμα και ειδικοί σύμβουλοι που παροτρύνουν τους ενδιαφερόμενους στο πως θα εφαρμοστούν κατάλληλα. Στην αρχή και εφ'όσον οι μεθοδολογίες χρησιμοποιούνταν ως επίσημες διαδικασίες έπεφτε μεγάλη βαρύτητα έκτος από την ανάπτυξη για την παράδοση του προϊόντος στη δημιουργία ποικίλων εγγράφων και κατά τη διάρκεια σχεδόν όλης της διαδικασίας, από την αρχή των συζητήσεων περι των απαιτήσεων μέχρι και τέλος της παράδοσης. Μάλιστα κάποια από αυτά τα έγγραφα θεωρούνταν ή και ακόμα θεωρούνται μέρος του παραδοτέου. Με την πορεία των χρόνων όμως ήταν εμφανές ότι η δημιουργία αυτών των εγγράφων κόστιζε και μπορούσε αντί να διευκολύνει να δυσχεράνει την προσπάθεια για την ολοκλήρωση μιας εργασίας. Οι

Περίληψη

Τα δύο μεγάλα ρεύματα Μεθοδολογιών που μπορούμε να συναντήσουμε στην επιστήμη της Ανάπτυξης Λογισμικού είναι οι Ευέλικτες και οι Παραδοσιακές ή αλλιώς εγγγραφοκεντρικές Μεθοδολογίες. Οι πρώτες που δημιουργήθηκαν ήταν οι παραδοσιακές και ακολούθησαν οι Ευέλικτες ως εξέλιξη των πρώτων προκειμένου να αντιμετωπίσουν κάποιες συγκεκριμένες καταστάσεις οι οποίες εμφανίζονταν συχνά. Συγκεκριμένα, η αλλαγή των απαιτήσεων ή αντίληψη κατά τη διάρκεια της ανάπτυξης ότι αυτές δεν επικοινωνήθηκαν σωστά από τον πελάτη στην ομάδα

ανάπτυξης οδήγησε στην αναζήτηση Μεθοδολογιών που να μπορούν να είναι ευέλικτες σε τέτοιου είδους ανατροπές και να προσαρμόζονται σε νέα δεδομένα κατά τη διάρκεια του κύκλου ανάπτυξης. Επίσης η δημιουργία μιας μεγάλης ποικιλίας εγγράφων έγινε αποδεκτό και από τις 2 πλευρές ότι δεν εξυπηρετούσε ούτε στην καλύτερη κατανόηση και χρήση του προϊόντος από τον πελάτη αλλά ούτε και στην επιβεβαίωση για την πορεία της ανάπτυξης όπως έτειναν να χρησιμοποιούνται πολλά από αυτά, σπταταλώντας με αυτόν τον τρόπο πόρους και δημιουργώντας επιπρόσθετο κόστος. Σε αυτά τα σημεία ήρθαν κυρίως για να κάνουν τη διαφορά με τις μέχρι τότε χρησιμοποιούμενε Μεθοδολογίες οι Ευέλικτες. Χαρακτηριστικές Μεθοδολογίες της πρώτης χρονικά κατηγορίας είναι αυτή του καταρράκτη, του Πρωτότυπου καθώς και οι προσθετικές αλλά και οι επαναληπτικές Μεθοδολογίες. Από τη δεύτερη κατηγορία είναι το SCRUM,ο Ακραίος Προγραμματισμός και οι Κρυσταλλικές Μεθοδολογίες.

Μέσα από τη χρήση των Ευέλικτων Μεθοδολογιών ανακαλύφθηκε η σημαντικότητα του ελέγχου ως προς την Ποιότητα του Λογισμικού που παραδίδεται στον πελάτη. Για αυτό το λόγο ο Έλεγχος στις Ευέλικτες Μεθοδολογίες ορίζεται ως μέρος της Ανάπτυξης και όχι ως αναξάρτητη φάση στο τέλος του κύκλου ανάπτυξης. Οι διάφορες εφαρμογές ελέγχων στις Ευέλικτες Μεθοδολογίες έχουν διαχωριστεί σε 4 μεγάλες κατηγορίες ή αλλιώς τεταρτημόρια. Συνοπτικά το πρώτο περιλαμβάνει ελέγχους που καθοδηγούνται από την τεχνολογία και εφαρμόζονται ώστε να υποστηρίξουν την ομάδα, η δεύτερη ελέγχους που υποστηρίζουν την ομάδα και δημιουργούνται με βάση την επιχειρηματική λογική του προϊόντος, η τρίτη αυτούς που εφαρμόζονται στο προϊόν για να αξιολογήσουν από πλευράς επιχειρηματικής λογικής του προϊόντος και η τελευταία ελέγχους τεχνολογικούς που εφαρμόζονται στο προϊόν για να αξιολογηθούν οι μη λειτουργικές απαιτήσεις.

Περίληψη στα Αγγλικά (Abstract)

The two main strams of methodologies we can meet in the science of Software Develpoment are those of Agile and tranditional or document oriented Methodologies. Tranditional are the first ones that have been created and Agile came as an evolution in order to overcome some special situations that are usual. Specifically change of requirements or the realization of non-well defined of those by the customers during development's phase leaded to the quest of

Methodologies being agile against such inversions and capable to adapt to new data during the development's lifecycle. Also, creation of various documents was accepted by both sides (meaning customers and developers' team) that many times didn't actually help comprehension of projects or did not even confirm development of those, since they were used for this propotion and finally they were creating an overcost. Those are the main points where Agile Methoodlogies targeted to make the difference. Some characteristic of the first chronically appeared Methodologies are Waterfall, Prototype and some incremental and iterative Methodologies. On the other hand there are SCRUM, Extreme Programming and Crystal Family Methodologies.

Over usage of Agile Methodologies it turned clear the role of testing and how important it is in the Quality of Software that is delivered to the customer. This is the reason why testing in Agile Methodologies is considered a part of development phase and not an independent phase that comes at the end of development's lifecycle. Agile testing and their applications have been categorized into 4 big categories or quadrants. The first one includes technology guided testing which are applied to support the team, second one has tests that support the team again but they are business oriented, the third has tests which are applied against the product to verify its business correctness and finally the forth defines tests that are technology oriented against the product to verify non functional requirements.

.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή μου και υπεύθυνο της πτυχιακής κ.Παναγιώτη Σφέτσο για τη βοήθεια που μου παρείχε όποτε χρειάστηκα αλλά και για την υπομονή που υπέδειξε.

Ευρετήριο Περιεχομένων

Πρόλογος 2

Περίληψη 2

Περίληψη στα Αγγλικά (Abstract) 3

Ευχαριστίες 4

Εισαγωγή 7

1. Γενική περιγραφή-Ιστορική αναδρομή 8
1.1 Εισαγωγή 8
1.2 Ιστορική αναδρομή 8
1.3 Παραδοσιακές προσεγγίσεις 9
1.3.1 Το μοντέλο καταρράκτη (Waterfall) 10
1.3.2 Το μοντέλο Πρωτοτύπου (Prototyping) 11
1.3.3 Οι προσθετικές και οι επαναληπτικές μέθοδοι 13
1.4 Περιγραφή Ευέλικτων Μεθοδολογιών και Πρακτικών 15
1.4.1 Γνωστές Ευέλικτες Μεθοδολογίες 16
1.4.1.1 SCRUM 16
Πρακτικές SCRUM 18
1.4.1.2 Ακραίος Προγραμματισμός(Extreme Programming) 19
Πρακτικές Ακραίου Προγραμματισμού 21
1.4.1.3 Οι Κρυσταλλικές Μέθοδοι(Crystal Methods) 22
1.5 Επίλογος 24
2. Ευέλικτος Έλεγχος και Ποιότητα Λογισμικού – Χρήσιμα Εργαλεία 25
2.1 Εισαγωγή 25
2.2 Ποιότητα Λογισμικού 25
2.2.1 Χαρακτηριστικά Ποιότητας 26
2.3 Ευέλικτος Έλεγχος 27
2.3.1 Τεταρτημόρια Ευέλικτων Ελέγχου 28
2.4 Εργαλεία Ευέλικτου Ελέγχου 30
2.4.1 Έλεγχοι Μονάδων και Στοιχείων 31
2.4.1.1 JUnit 31
2.4.1.2 TestNG 32
2.4.1.3 Cactus 33
2.4.2 Λειτουργικοί Έλεγχοι και Προσομοιώσεις 36
2.4.2.1 HttpUnit 36
2.4.2.2 Abbot 37
2.4.2.3 Mock Objects 38
2.4.2.3.1 EasyMock 40

2.4.2.3.2 JMock 42
2.4.3 Διερευνητικοί και Χρηστικοί Έλεγχοι 43
2.4.3.1 FIT 43
2.4.3.2 FitNesse 44
2.4.3.3 Concordion 45
2.4.3.4 Arbiter 46
2.4.4 Έλεγχοι Επιδόσεων και Ασφάλειας 47
2.4.4.1 Apache JMeter 47
2.4.4.2 DBMonster 50
2.5 Επίλογος 51
Αναφορές 51
Βιβλιογραφία 52
Παράρτημα Α 53
1. Introduction 54
2. Background 56
3. Results 62
4. Conclusions 69
5. Discussion 69
6. References 69

Ευρετήριο Εικόνων

Εικόνα 1 Waterfall model 10
Εικόνα 2 Prototyping model 12
Εικόνα 3 Incremental model 13
Εικόνα 4 Iterative model 14
Εικόνα 5 SCRUM 17
Εικόνα 6 Extreme Programming 20
Εικόνα 7 Crystal Methods 23
Εικόνα 8 Τεταρτημόρια Ευέλικτου Ελέγχου 28
Εικόνα 9 Αρχιτεκτονική Cactus 34
Εικόνα 10 Διεπιφάνεια JMeter 49
Εικόνα 11 JMeter Γράφημα 50

Εισαγωγή

Το παρόν πόνημα αποτελεί το βασικό μέρος της πτυχιακής εργασίας του φοιτητή Χρυσοβαλάντη Μπαμπάτση το οποίο έχει ως στόχο να παραθέσει μία σχετικά σύντομη περιγραφή από τις κυριότερες υπάρχουσες ευέλικτες μεθοδολογίες προγραμματισμού (Agile Methodologies). Πέραν τούτου όμως του γενικού πλαισίου, επικεντρώνεται σε κάποια άλλα στοιχεία που μπορούν να προσδιορίσουν συγκεκριμένα μία τέτοιου είδους μεθοδολογία όπως οι πρακτικές που εφαρμόζει (Agile practices) και ακόμα πιο συγκεκριμένα τις μεθόδους ελέγχου που εφαρμόζονται ή μπορούν να εφαρμοστούν σε αυτήν/ές. Ο σκοπός φυσικά όλων αυτών είναι ο τελικός προσδιορισμός του κατα πόσο αποδοτικές είναι αυτές οι μεθοδολογίες, η σύγκρισή τους με τις παραδοσιακές εγγραφο-κεντρικές μεθοδολογίες (document-driven methodologies) και τέλος το κατα πόσο οι έλεγχοι βοηθούν στη βελτιστοποίηση της ποιότητας του λογισμικού που παραδιδεται στον πελάτη.

Στο πρώτο κεφάλαιο θα γίνει μία περιγραφή των ευέλικτων μεθοδολογιών και μία μίκρη ιστορική αναδρόμη από το πότε και πώς ξεκίνησαν και η εξέλιξη που ακολούθησε στην πορεία του χρόνου. Κατόπιν θα συνεχίσει με περαιτέρω ανάλυση των ευέλικτων μεθοδολογιών και ως επί των πλείστων θα παρουσιαστούν τα κύρια συστατικά που απαρτίζουν αυτές και είναι κάποιες πρακτίκες που εφαρμόζονται και το ρόλο που παίζουν στον κύκλο ζωής ανάπτυξης λογισμικού. Επίσης μία σύντομη περιγραφή των παραδοσιακών προσεγγίσεων κρίνεται απαραίτητη αφού όπως έχει προγραφεί αποτελούν έναν από τους δύο όρους στη σύγκριση που θα ακολουθήσει.

Ακολουθεί το δεύτερο κεφάλαιο όπου γίνεται μία προσπάθεια να αναλυθεί ο ρόλος που παίζουν οι έλεγχοι στις ευέλικτες μεθοδολογίες και πόσο επηρεάζουν αυτοί την ποιότητα λογισμικού. Κατόπιν παρουσιάζεται ένα μοντέλο που κατηγοριοποιεί σε τέσσερα τεταρτημόρια τους ελέγχους στις Ευέλικτες Μεθπδολογίες. Επίσης μία σειρά από εργαλεία που χρησιμοποιούνται στις ευέλικτες μεθοδολογίες περιγράφονται και αξιολογούνται τόσο για την αποδοτικότητά τους, όσο και για την χρησιμότητά τους και την δημοτικότητά τους στους κόλπους της βιομηχανίας λογισμικού.

1. Γενική περιγραφή-Ιστορική αναδρομή

1.1 Εισαγωγή

Από τα μέσα της δεκαετίας του 90' αναπτύχθηκαν κάποιες εναλλακτικές μεθοδολογίες ανάπτυξης λογισμικού οι οποίες ονομάστηκαν ευέλικτες. Φυσικά αυτές δεν ήλθαν στο προσκήνιο, ούτε αναπτύχθηκαν αναίτια. Επίσης προκειμένου να φτάσουν στο σημείο που βρίσκονται σήμερα υπήρξε εξέλιξη κατόπιν είτε μελετών είτε και εμπειριών των προγραμματιστών και των ανθρώπων που εμπλέκονται στη δημιουργία κώδικα. Σε αυτό το κεφάλαιο θα αναφερθούμε στις πιο σημαντικές και ευρέως χρησιμοποιούμενες ευέλικτες μεθοδολογίες που αναπτύχθηκαν μέσα από το πέρασμα του χρόνου. Μία συντομή περιγραφή θα δοθεί ακόμα και για τις παραδοσιακές μεθοδολογίες δίχως όμως να εμβαθύνουμε στην κάθε μεθοδολογία από αυτές ξεχωριστά αλλά κυρίως αναφέροντας κάποια από τα κύρια χαρακτηριστικά τους που εμφανίζονται καθολικά σε αυτές. Έτσι μία σύγκριση μεταξύ αυτών θα ακολουθήσει και κατόπιν θα αναλυθούν λεπτομερέστερα οι ευέλικτες και πως αυτές εφαρμόζονται.

1.2 Ιστορική αναδρομή

Οι ευέλικτοι τρόποι προγραμματισμού υπήρξαν ως μία αντίδραση στις παραδοσιακές μεθοδολογίες οι οποίες είναι αυστηρές, «βαριές» και γενικότερα μη προσαρμόσιμες στις τυχόν αλλαγές που παραουσιάζονται κατα την δημιουγία ενός έργου. Η εφαρμογή των παραδοσιακών τακτικών ξεκινά με την εκμαίευση και την καταγραφή των απαιτήσεων του έργου ακολουθουμένων της αρχιτεκτονικής και της γενικής ανάλυσης και σχεδιασμού. Στα μέσα της δεκαετίας του 90' κάποιοι προγραμματιστές βρήκαν αυτά όλα τα βήματα αρκετά απογοητετυτικά, μη αποδοτικά για κάποιες περιπτώσεις και ίσως ακόμα και απίθανα για κάποιες άλλες. Η ταχύτητα με την οποία εξελίσσοταν και ακόμα εξελίσσεται ο τομέας της βιμηχανίας της τεχνολογίας αλλά και η ραγδαία δημιουργία νέων αναγκών καθιστούσε πολλές φόρες σχεδόν αδύνατον κάποιος πελάτης να γνωρίζει επακριβώς τι επιθυμεί να πραγματοποιεί το προϊόν που παραγγέλνει. Αυτό το γεγονός οδήγησε κάποιους προγραμματιστές αλλά και δειυθυντές έργων να αναπτύξουν ξεχωριστά και σε πρώιμο στάδιο μη καθορισμένα, κάποιες τακτικές ώστε να αντιπαρέλθουν σε αυτά τα εμπόδια που αντιμετώπιζαν. Οι Ευέλικτες

Μεθοδολογίες λοιπόν είναι κάποιες πρακτικές που μοιράζονταν τον ίδιο στόχο και συγκεντρώθηκαν ώστε να δημιουργήσουν κάτι πιο συγκρκριμένο. Κάποιες από αυτές τις πρακτικές μάλιστα ήταν τόσο παλιές που εμφανίζοταν από τα μέσα της δεκαετίας του 1970.

Οι Ευέλικτες Μεθοδολογίες γενικότερα θεωρούνται ως η εξέλιξη των προσθετικών μεθόδων ανάπυξης Λογισμικού και οι οποίες εμφανίστηκαν από τη δεκαετία του 50 και συγκεκριμένα το 1957. Το 1974 ο E.A. Edmonds δημοσίευσε για πρώτη φορά μία προσαρμόσιμη διαδικασία. Παράλληλα στο τηλεφωνικό κέντρο της Νέας Υόρκης εφαρμόζοταν μία τέτοιου είδους διαδικασία ακολουθώντας πρακτικές που ήταν οι πρόγονοι των Ευέλικτων Πρακτικών. Στα τέλη της δεκαετίας του 70’ ο Gielan έδινε διαλέξεις στην Αμερική πάνω σε αυτές τις μεθοδολογίες που άρχισαν να δημιουργούνται, τις πρακτικές που χρησιμοποιούνταν και τα προτερήματά τους.

Έτσι φτάσαμε στα μέσα της δεκαετίας του 90’ όπου ο αντικειμενοστρεφής προγραμματισμός γίνεται ο πιο διαδεμένος τρόπος υλοποίησης Λογισμικού καλιεργώντας το έδαφος για να μπορέσουν οι θεωρίες και κάποιες πρακτικές εφαρμοσμένες μεμονωμένα όπως περιγράφηκαν και πιο πάνω να συγκεκριμενοποιηθούν και από απλώς ένα ρεύμα, μία ιδέα να γίνουν μία Μεθοδολογία. Σε αυτό το χρονικό σημείο εμφανίζεται για πρώτη φορά η μεθοδολογία SCRUM η οποία είναι και μία από τις πιο διαδεδομένες ακόμα και σήμερα Ευέλικτη Μεθοδολογία και ακολούθησαν πολλές άλλες.

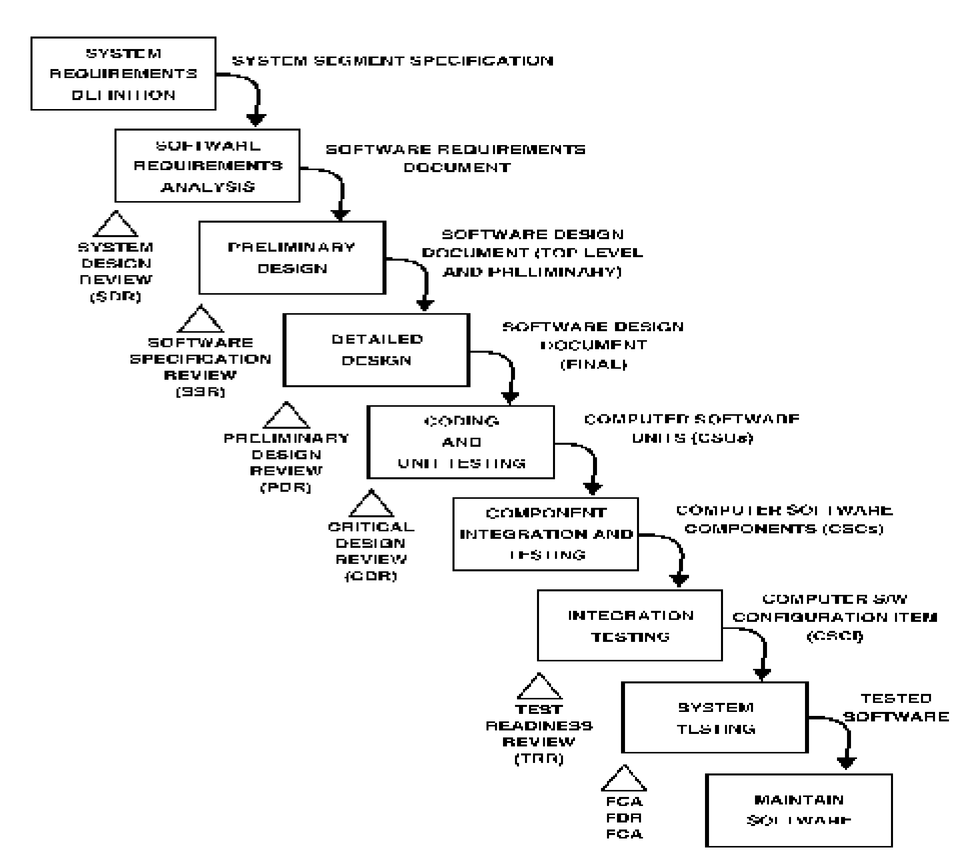
1.3 Παραδοσιακές προσεγγίσεις

Στο σημείο αυτό όπως έχει προγραφεί και προκειμένου να υπάχει μία πιο σφαιρική άποψη των Ευέλικτων Μεθοδολογιών θα περιγραφούν οι παραδοσιακές προσεγγίσεις στις οποίες βασίστηκαν οι παραδοσιακές μεθοδολογίες, έτσι ώστε να να γίνει πιο αντιληπτό στον αναγνώστη τι ακριβώς θέματα προέκυπταν και ακόμα προκύπτουν στην εφαρμογή των παραδοσιακών μεθόδων ποκαι τοα οποία προσπαθούν να αντιμετωπίσουν οι Ευέλικτες Μεθοδολογίες. Επίσης αυτή η αντιπαραβολή θα βοηθήσει στο να αναβρεθούν κατόπιν τα κοινά στοιχεία μεταξύ των 2 ρευμάτων (παραδοσιακές - Ευέλικτες Μεθοδολογίες) αλλά και οι διαφορές τους. Τέλος θα πρέπει να αναφερθεί ότι οι Ευέλικτες Μεθοδολογίες δεν ήρθαν για να αντικαταστήσουν πλήρως, να εξαλείψουν τις παραδοσιακές Μεθοδολογίες αλλά κυρίως να εμπλουτίσουν τους τρόπους δράσης των ανθρώπων που εμπλέκονται

στη δημιουργία Λογισμικού. Έτσι πρέπει να τονισθεί ότι οι παραδοσιακές μέθοδοι εφαρμόζονται κατα κόρον και εξελίσσονται και αυτές.

1.3.1 Το μοντέλο καταρράκτη (Waterfall)

Το μοντέλο καταρράκτη είναι το πρώτο που εμφανίστηκε στην ανάπτυξη Λογισμικού και σκοπό είχε να αξιολογήσει και να ικανοποιήσει τις ανάγκες των χρηστών. Αναφέρεται επίσης και γραμμικό-συνεχές μοντέλο (linear-sequential lifecycle model). Σε αυτό κάθε φάση πρέπει να αρχίζει αφού έχει ολοκληρωθεί η προηγούμενη. Στο τέλος κάθε φάσης γίνεται ένας έλεγχος αν η πραγματοποίηση κινείται σωστά ώστε να συνεχιστεί ή όχι το έργο. Σε αυτό το μοντέλο οι φάσεις δεν αλληλοκαλύπτονται. Στο παρακάτω σχήμα φαίνονται οι διάφορες φάσεις.



Εικόνα 1 Waterfall model

Όπως φαίνεται και παραπάνω το οντέλο αυτό ξεκινά με την ανάλυση των απαιτήσεων όπου αυτές γίνεται προσπάθεια να καθοριστούν μετά από εκτενείς συζητήσεις μεταξύ των πελατών, ειδικών αναλυτών απαιτήσεων (Business Analysts), αρχιτεκτόνων αλλά και έμπειρων μηχανικών Λογισμικού. Όταν οι συζητήσεις αυτές καταλήξουν σε κάποια συγκεκριμένα χαρακτηριστικά

καταγράφονται λεπτομερώς και οι λειτουργικές αλλά και οι μη λειτουργικές απαιτήσεις και ακολουθεί η επόμενη φαση.

Επόμενη φάση είναι ο σχεδιασμός και αυτός είναι σαφώς κάτι πολύ πιο συγκεκριμένο από την προηγούμενη φάση, για την ακρίβεια είναι η φάση που συγκεκριμενοποιουνται και επιμερίζονται τα αποτελέσματα της προηγούενης φάσης. Στο στάδιο αυτό οι προδιαγραφές που προέρχονται από τα έγγραφα μελετώνται από μηχανικούς Λογισμικού και σε συνεργασία με αναλυτές Βάσεων Δεδομένων και άλλων ειδικών δίνονται λύσεις με σκοπό να προκύψει η αρχιτεκτονική του συστήματος. Με λιγά λόγια σε αυτή τη φάση δίνονται οι λύσεις στα επιμέρους ερωτήματα –προβλήματα (απαιτήσεις) που είχαν δημιουργηθεί στην προηγούμενη.

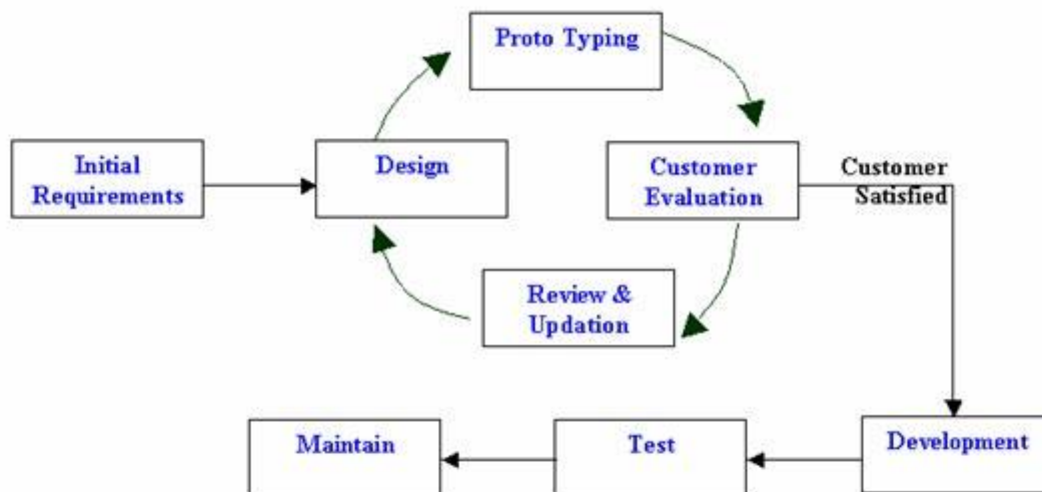
Κατόπιν ακολουθεί η φάση της πραγματοποιήσης και του ειδικού ελέγχου. Σε αυτό το σημείο είναι που γράφεται ο κώδικας όπως είχαν καθορίσει οι ειδικοί στη προηγούμενη φάση, δημιουργούνται οι βάσεις δεδομένων σύμφωνα με τα σχήματα που έιχαν επίσης προβλεφθεί και δοκιμάζονται σε επίπεδο απλής λειτουργίας (unit testing).

Επόμενο στάδιο είναι ο ολοκληρωμένος έλεγχος των υποστημάτων αλλά και ολόκληρου του συστήματος. Αυτό το στάδιο απαιτεί άτομ-ελεγκτές (testers) που αναπαράγουν σενάρια τα οποία προσομοιώνουν την καθημερινή λειτουργία που λαμβάνει μέρος για τον πελάτη. Έτσι εκτός από την συνολική λειτουργία του συστήματος ελέγχονται και οι επιδόσεις συνήθως με διάφορα εργαλεία. Και τα σενάρια αλλά και οι απαιτήσεις είναι προδιαγεγραμμένα από το στάδιο της ανάλυσης των απαιτήσεων και όπως έχει προειπωθεί αφού αυτό το στάδιο έχει τερματιστεί τίποτα από τα παραπάνω δεν μπορεί να αλλάξει από τη στιγμή που αρχίζει ο σχεδιασμός του συστήματος, πόσω μάλλον όταν έχει ξεκινήσει η εκπλήρωσή του.

Αφού λοιπόν ολοκληρωθεί και αυτή η φάση ακολουθεί η παράδοση του προϊόντος στον πελάτη με ότι αυτή εχεί προδιαγραφεί να υποστηρίζεται όπως τα έγγραφα χρήσεως (manuals), έγγραφα λειτουργικότητας και σχεδιασμού, η εκπαίδευση κάποιων χρηστών για την ομαλή λειτουργία του συστήματος ή και την παρακολούθηση αυτού και τέλος η υποστήριξη του πελάτη.

1.3.2 Το μοντέλο Πρωτοτύπου (Prototyping)

Πρόκειται για έναν τρόπο δημιουργίας Λογισμικού όπου αρχικά υλοποιούνται πρωτότυπες, μη ολοκληρωμένες εφαρμογές με κυρίαρχο σκοπό να αξιολογηθούν από τον πελάτη. Τα πρωτότυπα εκτός του ότι είναι ατελείς εφαρμογές μπορούν ακόμα και να διαφοροποιούνται σε σχέση με το τελικό προϊόν εκτός από το μέγεθος και το πλήθος των λειτουργιών που προσφέρουν ακόμα και στη φύση της λειτουργίας που επείδικνύουν στον πελάτη. Αυτό σημαίνει ότι μπορεί στο τέλος ένα πρωτότυπο να έχει ελάχιστες ομοιότητες στις λειτουργίες που αυτό υποστηρίζει με αυτό που κατέληξε να έχει το προϊόν. Στο παρακάτω σχήμα φαίνονται τα στάδια ανάπτυξης σύμφωνα με την πρεσέγγιση του Πρωτοτύπου.



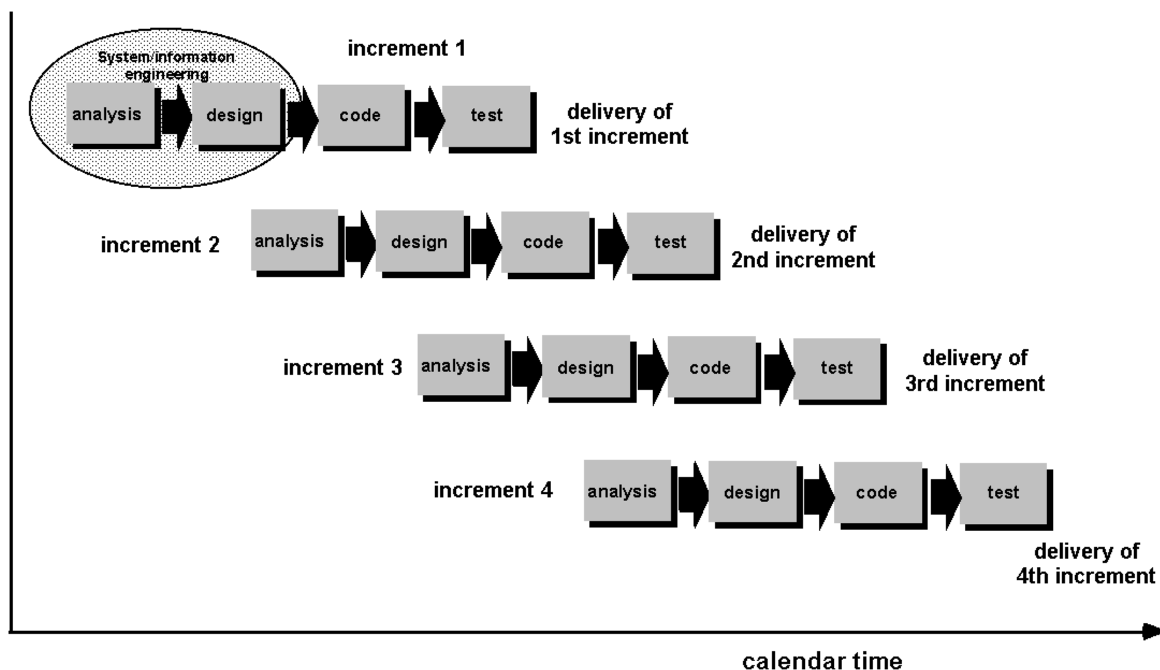
Εικόνα 2 Prototyping model

Το μοντέλο υπαγορεύει ότι ο κύκλος ζωής της δημιουργίας ξεκινά με τις αρχικές απιτήσεις. Γενικά το στάδιο αυτό δε διαφέρει από την ανάλυση απαιτήσεων που βρίσκουμε και σε άλλου είδους προσεγγίσεις και η λόγος ύπαρξης αυτής της φάσης είναι να καθοριστεί η είσοδος και έξοδος του συστήματος και όχι τόσο κάποιες λεπτομέρειες όπως για παράδειγμα η ασφάλεια. Κάπου εδώ έρχονται και οι διαφορές σε σχέση με την προηγούμενη προσέγγιση, το μοντέλο πρωτότυπου ορίζει ότι αμέσως αφού οι απαιτήσεις (οι λειτουργικές κυρίως) ολοκληρωθούν δημιουργείται ένα πρωτότυπο. Οι σχεδιαστές συστημάτων μαζί με τους άλλους ειδικούς οπώς τους αρχιτέκτονες, τους σχεδιαστές-ειδικούς βάσεων δεδομένων, τους έμπειρους προγραμματιστές δημιουργούν ένα πρωτότυπο αρχικά σχεδιάζοντάς το και κατόπιν υλοποιώντας το. Αυτό που ενδιαφέρει σε αυτό το στάδιο δεν είναι τόσο η ποιότητα ή η εκπλήρωση των μη λειτουργικών προδιαγραφών, αλλά κυρίως να φτιαχτεί μία λειτουργική διεπαφή με

το χρήστη. Πάνω σε αυτή θα προχωρήσει έτσι και η αξιολόγηση του πρωτότυπου και κατόπιν συζητούνται με τον πελάτη οι διαφοροποιήσεις που πρέπει να λάβουν χώρα. Έτσι αφού καταλήξουν σε κάποια συμπερασμάτα και συμφωνήσουν οι δύο μεριές το πρωτότυπο ξαναπερνά από τη φάση της σχεδίασης και βελτιώνεται για να παραδοθεί μία καλύτερη εκδοχή του (version) στον πελάτη για εκ νέου αξιολόγηση. Ο κύκλος αυτός συνεχίζεται μέχρι ο πελάτης να ωι΄ψσει ικανοποιημένος με το προϊόν και να δωθεί έτσι το έναυσμα για την τελική ανάπτυξη. Μετά την ανάπτυξη ακολουθεί όπως πάντα ο έλεγχος ολοκλήρωσης και τελικά η παράδοση του λογισμικού με ότι αυτή μπορεί να συμπεριλαμβάνει (έγγραφα,εκπαίδευση κ.α.).

1.3.3 Οι προσθετικές και οι επαναληπτικές μέθοδοι

Σύμφωνα με αυτές τις μεθόδους ακολουθείται μία τακτική που εμφανίζει πολλές ομοιότητες με τη μέθοδο waterfall. Η βασική όμως διαφορά είναι ότι εφαρμόζεται στα επιμέρους κομμάτια που σπάει το λογισμικό που θα πραδοθεί. Αυτή η τακτική δημιουργήθηκε όταν έγινε αντιληπτό ότι κάποιες από τις λειτουργίες-απιτήσεις ενός παραδοθέντος συστήματος δεν σχετίζοταν καθόλου με κάποιες άλλες. Έτσι μπορούσε να μην εφαρμοστεί μία αλυσίδα ή ενα μονοδιάστατο μοντέλο ανάπτυξης όπως δείχνει και το σχήμα παρακάτω.

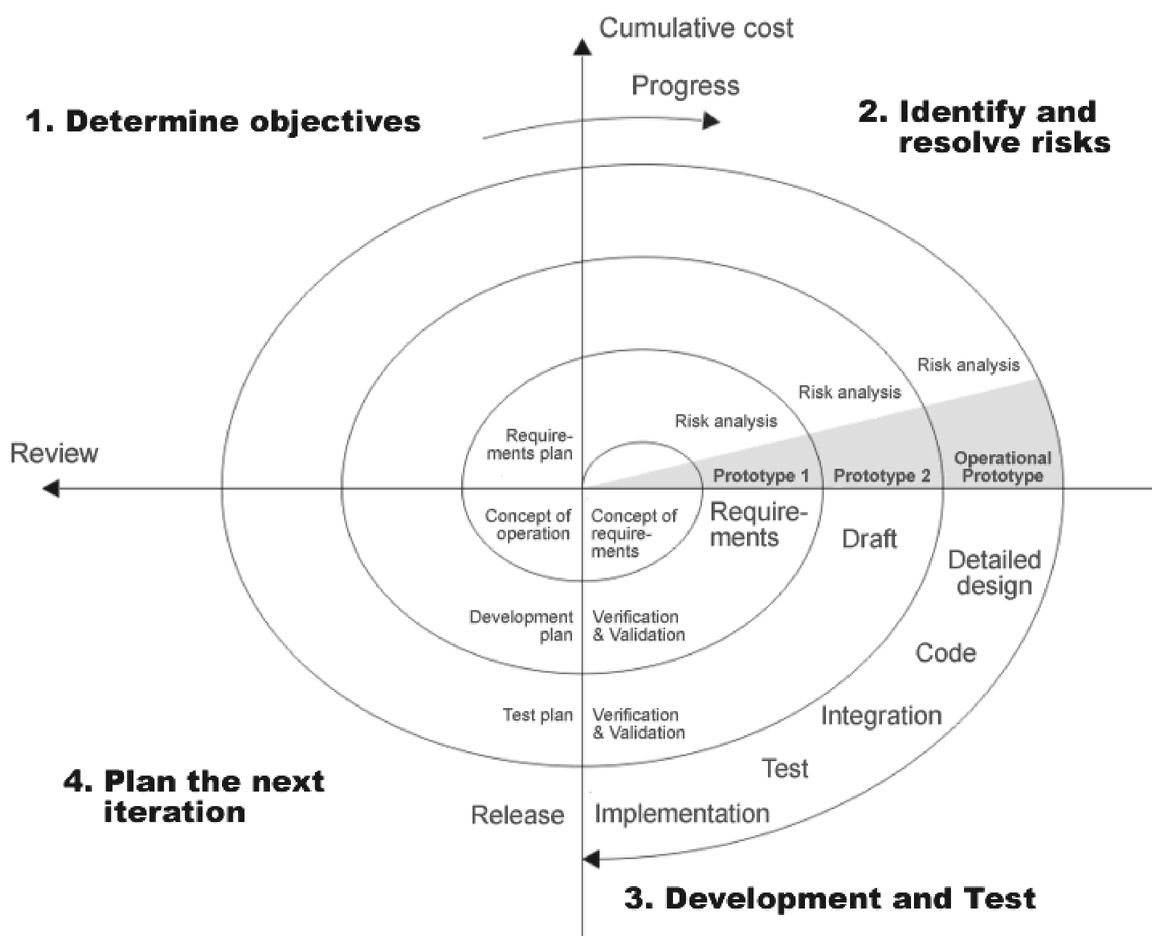


Εικόνα 3 Incremental model

Το προϊόν κατακερματίζεται σε μικρότερα και ο λόγος αυτής της διάσπασης είναι ότι με αυτόν τον τρόπο εξοικονομείται χρόνος και συνεπώς υπάρχουν και

οικονομικές ωφέλειες. Όπως φαίνεται και από το σχήμα οι διάφορες επαυξήσεις μπορούν να βρίσκονται παράλληλα σε παραγωγή και η κάθε μία να είναι στο ικό της στάδιο όπως αυτό περιγράφεται από το μοντέλο Waterfall.

Μία άλλη προσέγγιση που μοιάζει φαινομενικά με αυτή των αυξητικών μεθόδων είναι οι επαναληπτικές μέθοδοι. Και σε αυτήν έχουμε την κατάτμηση του μελλοντικού προϊόντος όμως εδώ οι λόγοι που ώθησαν στη δημιουργία αυτών των μεθόδων δεν είναι η εξοικονόμηση πόρων. Οι μέθοδοι αυτές πρωτοεμφανίστηκαν όταν έγινε αντιληπτό ότι ο κυριότερος παράγοντας ενός έργου ήταν η προσαρμοστικότητα αυτού στις αλλαγές απαιτήσεων. Αυτός ίσως είναι και ο λόγος που οι Ευέλικτες Μεθοδολογίες θεωρούνται η εξέλιξη των επαναληπτικών μεθόδων. Στο σχήμα 4 φαίνεται ο κύκλος ζωής ενός έργου στον οποίο ασκείται μία επαναληπτική μέθοδος.



Εικόνα 4 Iterative model

Η ανάπτυξη σε αυτήν τη μέθοδο βασίζεται στη διάσπαση του κύκλου παραγωγής σε επαναλήψεις διαφόρων χρονικών διαστημάτων. Η πρώτη είναι και

η κυριότερη δηλαδή αυτή που περιέχει τις βασικές λειτουργίες. Στο τέλος κάθε επανάληψης παραδίδεται ένα ολοκληρωμένο και λειτουργικό μέρος του προϊόντος. Η σειρά της κάθε επανάληψης που θα παραδοσθεί ορίζεται σύμφωνα με τη λειτουργικότητα και για αυτόν τον λόγο η πρώτη είναι και αυτή με την κυριότερη λειτουργικότητα. Καθ'αυτόν τον τρόπο έχουμε στο τέλος κάθε επανάληψης ένα πλήρως λειτουργικό προϊόν, ελεγμένο και ικανοποιώντας όλες τις επιμέρους ποιοτικές απαιτήσεις και βελτιωμένο σε σχέση με το προηγούμενο παραδοθέν.

Μία παραλλαγή αυτών των μεθόδων είναι το σπειροειδές μοντέλο (Spiral model). Η προτεραιότητα αυτού του μοντέλου είναι η επισφάλεια που φέρει το προϊόν στην ολοκλήρωσή του αλλά και στον χρόνο παράδοσής του. Αυτή η βαρύτητα που δίνεται στο ρίσκο είναι και που καθορίζει το πως θα διασπαστεί σε επαναλήψεις ο κύκλος εργασίας της ομάδας ανάπτυξης. Ενώ λοιπόν στις επαναληπτικές μεθόδους η πρώτη επανάληψη περιλαμβάνει το πιο λειτουργικό μέρος ενος συστήματος και ακολουθείται από λιγότερο λειτουργικά, στο σπειροειδές μοντέλο η πρώτη επανάληψη έχει ως στόχο την ολοκλήρωση του πιο «επικίνδυνου» υποσυστήματος.

1.4 Περιγραφή Ευέλικτων Μεθοδολογιών και Πρακτικών

Σε αυτό το σημείο θα περιγραφούν οι Ευέλεκτες Μεθοδολογίες. Βέβαια το να περιγραφούν όλες οι Ευέλικτες Μεθοδολογίες θα ήταν σχεδόν αδύνατον αφού αυτές ολοένα και διαδίδονται πιο πολύ και δημιουργούνται είτε τέλειως καινούριες είτε νέες υβριδικές που συνδυάζουν διάδορα στοιχεία από προγενέστερες. Παρ'όλα αυτά σε αυτό το κεφάλαιο θα περιγραφούν οι πλέον γνωστές Ευέλικτες Μεθδολογίες.

Πότε όμως μία Μεθοδολογία θεωρείται ότι ανήκει στην οικογένεια των Ευέλικτων Μεθοδολογιών; Κατά βάση μία Μεθοδολογία θεωρείται Ευέλικτη όταν μπορεί και προσαρμόζεται στις αλλαγές που προκύπτουν στη διάρκεια της ανάπτυξης ή ακόμα και όταν μπορεί να ανταποκριθεί σε περιπτώσεις που οι απαιτήσεις δεν είναι πλήρως γνωστές από την αρχή. Κατα τα άλλα οι Ευέλικτες Μεθοδολογίες όπως θα φανεί και πιο κάτω έχουν κάποια κοινά χαρακτηριστικά. Μέσα σε αυτά τα κοινά χρακτησιστικά είναι συχνά και η κοινή χρήση κάποιων πρακτικών πολλές εκ των οποίων προϋπήρχαν από τις Ευέλικτες Μεθοδολογίες και άλλες χρησιμοποιούνταν και από τις εγγραφοκεντρικές Μεθοδολογίες, άλλες ήταν τελείως παραγκωνισμένες. Μέσα απο την περιγραφή των Ευέλικτων Μεθοδολογιών θα προσπαθήσουμε να δώσουμε και μία περιγραφή αυτών των

συστατικών τους που αποτελούν ίσως το πιο κύριο χαρακτηριστικό τους στοιχείο. Όμως ας επανέλθουμε στο βασικό ερώτημα που θίξαμε, τι είναι Ευέλικτη Μεθοδολογία πέραν από την προφανή απάντηση που εξ'αλλού φανερώνει και το όνομα τους. Φυσικά αυτό το ερώτημα δεν τίθεται για πρώτη φορά και για την ακρίβεια απασχόλησε πολλά άτομα που εμπλέκονται στην παραγωγή Λογισμικού. Στην προσπάθεια να καθοριστεί το τι πρέπει να σέβεται μια Ευέλικτη Μεθοδολογία, ποιες είναι οι βασικές αρχές της, το 2001 συνευρέθησαν άτομα από διάφορους οργανισμούς που χρησιμοποιούσαν Ευέλικτες Μεθοδολογίες ώστε να συγκεντρώσουν τα βασικά τους χαρακτηριστικά. Το αποτέλεσμα αυτής της συνεύρεσης ήταν το Agile Manifesto, ένα κείμενο γραμμένο το οποίο εν ολίγοις κατέληξε να ορίζει το ποια Μεθοδολογία ήταν και θα θεωρείται από εκείνο το σημείο και έπειτα ως Ευέλικτη. Θα ήταν ανούσιο να παρατεθεί ολόκληρο το κείμενο εδώ, πάντως οι 12 αρχές που κατέληξαν μπορούν να βρεθούν επισήμως εδώ: http://agilemanifesto.org/iso/el/principles.html . Το απόσταγμα όλων αυτών που κατέληξαν και που ορίζουν τις Μεθοδολογίες ειναι οι εξής 4 προτάσεις:

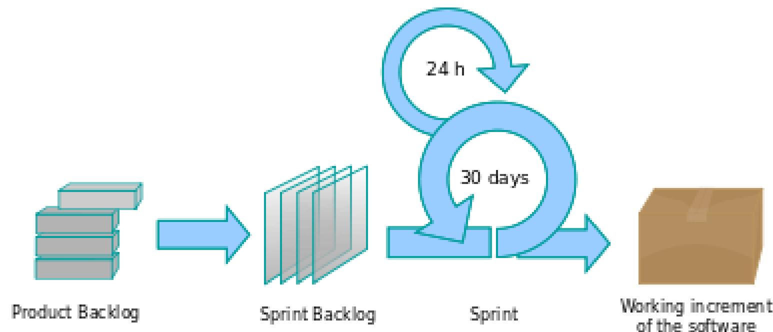
1)Τα άτομα και τις αλληλεπιδράσεις πάνω από τις διαδικασίες και τα εργαλεία.
2)Το λογισμικό που λειτουργεί πάνω από την εκτενή τεκμηρίωση.
3)Την συνεργασία με τον πελάτη πάνω από τις συμβατικές διαπραγματεύσεις.
4)Την ανταπόκριση στην αλλαγή πάνω από την τήρηση ενός προδιαγεγραμμένου σχεδίου.

1.4.1 Γνωστές Ευέλικτες Μεθοδολογίες

1.4.1.1 SCRUM

Είναι η πρώτη Ευέλικτη μεθοδολογία που εμφανίστηκε και σε γενικές γραμμές είναι μία επαναληπτική μεθοδολογία. Βασίζεται σε 3 ρόλους που λαμβάνουν μέρος. Ο πρώτος ειναι ο κάτοχος του προϊόντος και καθορίζει τι πρέπει να παραδοθεί στις επόμενες 30 μέρες ή και ακόμα λιγότερο. Ο δεύτερος ρόλος είναι οι ομάδες ανάπτυξης οι οποίες παραδίδουν και παρσουσιάζουν αυτό που πρέπει σε 30 ή λιγότερες μέρες και πάνω σε αυτό είναι που κρίνεται από τον κάτοχο του προϊόντος το τι πρέπει να παραδοθεί στην επόμενη επανάληψη. Τρίτος ρόλος είναι ο SCRUM ειδικός (Master) ο οποίος διαβεβαιώνει ότι η διαδικασία προχωράει ομαλά αλλά συγχρόνως βοηθαέι στο να βελτιωθεί η ίδια η διαδικασία, η ομάδα ανάπτυξης και το προϊόν.

Στο σχήμα απεικονίζονται τα επιμέρους στάδια που αποτελείται η SCRUM διαδικασία. Πέρα από τους ρόλους η SCRUM μεθοδολογία περιγράφει όπως είναι φυσικό τον τρόπο ανάπτυξης. Αυτή αποτελείται από 3 φάσεις.



Εικόνα 5 SCRUM

Η πρώτη φάση είναι η προ παιχνιδιού και σε αυτή συμπεριλαμβάνονται δύο μικρότερες φάσεις, ο σχεδιασμός και η σχεδίαση αρχιτεκτονικής/υψηλού επιπέδου. Ο σχεδιασμός περιλαμβάνει τον καθορισμό του συστληματος που πρόκειται να δημιουργηθεί. Μία λίστα εκκρεμοτήτων προϊόντος δημιουργείται που περιλαμβάνει τις γνωστές σε εκείνο το σημείο απαιτήσεις (Product Backlog List). Οι απαιτήσεις ορίζονται οπως πάντα από αναλυτές, τεχνικούς, άτομα του Marketing. Κάθε φορά γίνεται προσπάθεια αυτή η λίστα να ανανεώνεται και να τίθονται εκ νέου εκτός από τις απαιτήσεις τις ίδιες, η προτεραιότητα της κάθε μίας αλλά και οι εκτιμήσεις του χρόνου επίτευξης. Ο σχεδιασμός τέλος περιλαμβάνει και την κατάρτιση των ομάδων-ρόλων, τον προσδιορισμό των εργαλείων που θα χρησιμοποιηθούν, την αξιολόγηση του ρίσκου και όλα αυτά βεβαίως επανεξετάζονται στο τέλος κάθε επανάληψης. Η σχεδίαση της αρχιτεκτονικής από την άλλη μεριά περιλαμβάνει την γενική σχεδίαση του έργου σύμφωνα με τις απαιτήσεις που υπάρχουν τη δεδομένη στιγμή στη λίστα και όταν αυτή αλλάζει φυσικά επανεξετάζεται και η αρχιτεκτονική του συστήματος. Αυτή η διαδικασία της επανεξέτασης συνήθως γίνεται σε συνδριάσεις και μέσα από αυτές προκύπτουν επίσης και πλάνα του τι πρόκειται να παραδοθεί.

Η δεύτερη φάση είναι αυτή της υλοποίησης (η οποία καλείται επίσης και φάση παιχνιδιού στην τακτική SCRUM). Αυτή η φάση είναι και που μπορεί και μετατρέπει την SCRUM μεθοδολογία σε ευέλικτη καθώς τίποτα δεν είναι προκαθορισμένο ως προς τον χρόνο, τις απαιτήσεις, τα εργαλεία που χρησιμοποιούνται, τους πόρους αλλά όλες αυτές οι παράμετροι

επαναπροσδιορίζονται μέσα από τις πρακτικές που χρησιιμοποιούνται σε κάθε Sprint. Το Sprint πρακτικά είναι η κάθε επανάληψη κατά τη διάρκεια της δημιουργίας. Κάθε Sprint περιλαμβάνει όλες τις φάσεις της ανάπτυξης όπως απαιτήσεις, ανάλυση, σχεδιασμό, εξέλιξη και παράδοση και συνήθως το διάστημα που διαρκεί ποικίλει από μία εβδομάδα ως ένα μήνα. Το αποτέλεσμα είναι στο τέλος κάθε Sprint να βγαίνει μία νεότερη έκδοση του προϊόντος παρέχοντας περισσότερες λειτουργίες.

Τελευταίο στάδιο είναι η μετα-παιχνιδιού φάση στην οποία περιλαμβάνεται η οριστικοποίηση του τελικού παραδοθέντος μετά από συμφώνια που έχει προκύψει μεταξύ των δύο πλευρών. Σε αυτή τη φάση εκτός από την παράδοση περιλαμβάνονται ο έλεγχος ολοκλήρωσης, ο συστημικός έλεγχος και η παράδοση των εγγράφων που έχει συμφωνηθεί.

Πρακτικές SCRUM

Sprint: Είναι η διαδικασία προσαρμογής σε παραλλαγμένες παραμέτρους του έργου όπως ο χρόνος, η τεχνολογία, οι πόροι, οι απαιτήσεις και το αποτέλεσμα αυτού είναι, στο τέλος του κύκλου ο οποίος συνήθως διαρκεί 30 μέρες, να παραδίνεται επιπρόσθετη λειτουργικότητα.

Συνεδρίαση Sprint Σχεδίασης(Sprint Planning Meeting): Περιλαμβάνει δύο φάσεις. Η πρώτη οργανώνεται από τον ειδικό SCRUM και μετέχουν όλοι όσοι εμπλέκονται στο έργο με σκοπό να αποφασιστεί το παραδοθέν του επόμενου Sprint και η δεύτερη η οποία επίσης οργανώνεται υπό τον ειδικό SCRUM και μετέχει η ομάδα ανάπτυξης για να καθοριστούν πως θα υλοποιηθεί το νέο λειτουργικό παραδοθέν με πιο τεχνικές λεπτομέρειες.

Εκκρεμότητες Sprint(Sprint Backlog): Είναι η αφετηρία στην ουσία του κάθε Sprint και είναι επιλογή μίας πλειάδας εκκρεμοτήτων που έχουν επιλεγεί από τη λίστα του προϊοντος κατα την πρώτη φάση της Συνεδρίασης Σχεδίασης Sprint. Σε αντίθεση με τη λίστα εκκρεμοτήτων του προϊόντος αυτή η λίστα αφού καταρτιστεί δεν αλλάζει μέχρι το τέλος του Sprint. Όταν όλες οι εκκρεμότητες διευθετηθούν τότε το Sprint λαμβάνει τέλος και μία νέα έκδοση του προϊόντος παραδίδεται.

Συνεδρίαση Ελέγχου Sprint(Sprint Review Meeting): Την τελευταία μέρα του Sprint ο ειδικός Sprint μαζί με την ομάδα ανάπτυξης παρουσιάζουν στον κάτοχο του προϊόντος, διάφορους χρήστες, τον πελάτη το νέο παραδοθέν με σκοπό να γίνει έλεγχος αυτού και να αποφασιστεί το μέλλον του προϊόντος με τυχόν αλλαγές.

Καθημερινή SCRUM Συνεδρίασγ(Daily SCRUM Meeting): Μία από τις διασημότερες πρακτικές του SCRUM είναι οι καθημερινές συνεδριάσεις που συμβαίνουν το πρωί διαρκούν σχετικά λίγο (περίπου 10-20 λεπτά) και λειτουργούν ως βραχυπρόθεσμοι σχεδιασμοί μέχρι την επόμενησυνεδρίαση και παρακολούθηση των όσων έχουν συμβεί από την προηγούμενη.

*Εκκρεμότητες Προϊόντος(Product Backlog):*Είναι η προσωρινή εκτίμηση των αναγκών του τελικού προϊόντος και σύμφωνα με αυτή ορίζεται η δουλειά που οφείλεται να γίνει. Αυτή η κατάρτιση εκτός από νέες απαιτήσεις μπορεί να περιλαμβάνει νεές τεχνολογίες, διόρθωση λαθών, προσθέσεις νέων χαρακτηριστικών. Στην πρακτική αυτή μπορεί να λαμβάνει μέρος οποιοσδήποτε εμπλέκεται στο έργο.

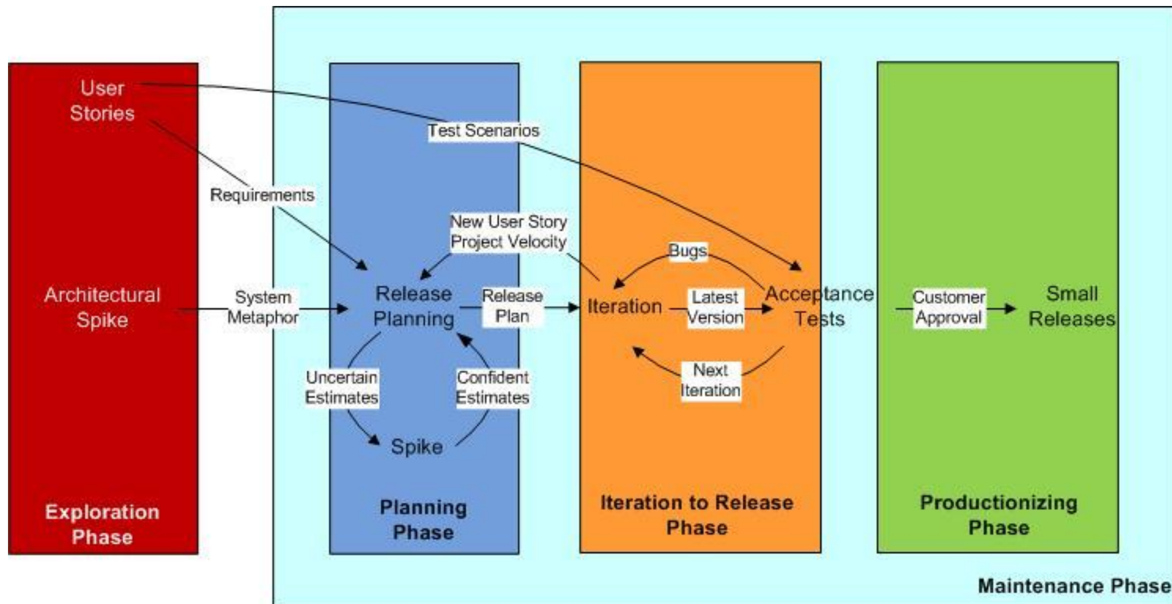
Εκτίμηση Προσπάθειας(Effort Estimation) :Με αυτή την πρακτική ο κάτοχος του προϊόντος μαζί με την ομάδα ανάπτυξης επαναπροσδιορίζουν με μεγαλύτερη ακρίβεια μία εκκρεμότητα όταν η λίστα αυτών ανανεώνεται και παρέχεται σε αυτούς νέα πληροφορία.

1.4.1.2 Ακραίος Προγραμματισμός(Extreme Programming)

Είναι μία Ευέλικτη Μεθοδολογία η οποία ξεκίνησε με το σκεπτικό να ικανοποιηθεί ο πελάτης ότι και να γίνει, εξ’άλλου είναι και μία από τις βασικές αρχές των Ευέλικτων Μεθοδολογιών και αυτή τη στιγμή είναι η πιο διάσημη. Χαρακτηριστικό του είναι οι μίκρες επαυξανόμενες παραδόσεις με στόχο την επανένταξη νέων απαιτήσεων μετά το τέλος αυτών και πριν την αρχή των επόμενων παραδόσεων. Πρόκειται για την πιο «χαλαρή» Μεθοδολογία μεταξύ των Ευέλικτων που πρακτικά σημαίνει ότι είναι μία δίχως όρια και δεσμά Μεθοδολογία η οποία αφήνει εντελώς ανοιχτά τα περιθώρια στην ομάδα να προσαρμόσει τον τρόπο που θα δουλέψει. Στην ουσία πρόκειται για μία συνάθροιση προυπάρχοντων πρακτικών και αρχών με το σκεπτικό ότι τις εξωθούνε ώστε να αποδώσουν τα μέγιστα. Σε αυτό οφείλεται και το όνομα της Μεθοδολογίας.

Στο σχήμα φαίνεται ο κύκλος ζωής της Μεθοδολογίας. Σε αυτόν υπάρχουν διάφοροι ρόλοι που συναντώνται σχεδόν σε κάθε Μεθοδολογία όπως αυτός του Προγραμματιστή, του Ελεγκτή(tester), του Πελάτη, του Διευθυντή που λίγο ή πολύ είναι υπεύθυνοι για θέματα όπως πάντα. Υπάρχουν όμως και ρόλοι όχι τόσο συχνα εμφανιζόμενοι όπως του Συμβούλου. Αυτός έχει ως στόχο να δίνει λύσεις σε συγκεκριμένα τεχνικά προβλήματα και συνήθως είναι ένα εξωτερικό μέλος της ομάδας. Επίσης υπάρχει ο προπονητής (coach) που σκοπό έχει να αντιληφθεί

ακριβώς το πως η ομάδα θα δουλεύει και να κανονίζει να τηρείται αυτός ο τρόπος λειτουργίας. Τέλος υπάρχει ο ανιχνευτής που σκοπό έχει να επιβλέπει τις εκτιμήσεις και να δίνει αναφορά το κατα πόσο ακριβείς είναι αυτές με απώτερο σκοπό αυτές να βελτιώνονται σταδιακά. Επίσης ελέγχει τη διαδικασία της υλοποίησης σε μια επανάληψη και κρίνει αν είναι εφικτή η επίτευξη του στόχου.



Extreme Programming (XP) lifecycle

Εικόνα 6 Extreme Programming

Ο κύκλος ζωής ξεκινά με τη φάση της εξερεύνησης και σε αυτήν την περίοδο ο πελάτης γράφει σε κάρτες ιστοριών (card stories) τα χαρακτηριστικά που θέλει να υπάρχουν στην πρώτη έκδοση. Παράλληλα η ομάδα δημιουργεί ένα πρωτότυπο με εκτός τον ξεκάθαρο σκοπό να αρχίζει να παράγει να προσπαθεί να εξοικειωθεί με τις τεχνολογίες και την ίδια τη διαδικασία της ανάπτυξης. Τη φάση αυτή ακολουθεί η φάση σχεδιαμού όπου σε αυτό εδώ το σημείο επικεντρώνεται η προσπάθεια στην ανεύρεση των πιο σημαντικών καρτών από αυτές που παραδόθηκαν στην πρώτη φάση (όσο το δυνατόν λιγότερες) και η συμφωνία στο χρόνο που αυτές θα πραγματοποιηθούν. Από τη μεριά των συμμετεχόντων στη διαδικασία της ανάπτυξης αυτές οι κάρτες διασπώνται σε εργασίες (tasks) και κάθε μία από αυτές γράφονται σε κάρτες εργασιών που μπορεί να είναι οποιασδήποτε μορφής ανεπίσημο έγγραφο ακόμα και ένα σχέδιο γραμμένο στο χέρι. Η επόμενη φάση που είναι η επανάληψη για κυκλοφορία ή όπως αλλιώς αναφέρεται οι επαναλήψεις κατασκευής ή ακόμα και απλά επαναλήψεις είναι η κυριότερη φάση όσον αφορά την ανάπτυξη καθώς εδώ λαμβάνουν χώρα τα στάδια του σχεδιασμού του μοντέλου, του προγραμματισμού, του ελέγχου και της ολοκλήρωσης. Τέλος είναι η φάση της προϊοντοποίησης όπου εκτελούνται

κάποιες διαδικασίες προκειμένου να πιστοποιηθεί ότι το προϊόν μπορεί να βγει στην παραγωγή. Αυτές μπορεί να είναι ο έλεγχος του συστήματος, ο έλεγχος της εγκατάστασης, ο έλεγχος της φόρτωσης (system, installation, load testing). Φυσικά σες αυτή τη φάση η διαδικασία παραγωγής συνεχίζεται αλλά μεσαφώς μικρότερο ρυθμό και μία εκτίμηση πάντα γίνεται για το σε ποιο παραδοθέν θα επικολληθούν τα αποτελέμσατα παραγώγής αυτής της φάσης , δηλαδή στο τρέχον ή σε κάποιο επόμενο. Τέλος πάντα κάποια συμφωνηθέντα έγγραφα παραδίδονται στον πελάτη (είτε εντελώς νέα είτε αναπροσαρμοσμένα).

Πρακτικές Ακραίου Προγραμματισμού

Ο σχεδιασμός παιχνιδιού (planning game): Στην αρχή κάθε επανάληψης πελάτες, υπεύθυνοι και προγραμματιστές συγκεντρώνονται προκειμένου να αναλύσουν, να ακτιμήσουν και να θέσουν προτεραιότητες στις απαιτήσεις για την επόμενη παράδοση. Οι απαιτήσεις αυτές ονομάζονται «ιστορίες χρήστη» και περιγράφονται σε «κάρτες ιστορίας» με τρόπο αντιληπτό από όλους τους συμμετέχοντες.
Μικρές Παραδόσεις: Μία αρχική έκδοση παραδίδεται με το πέρας κάποιων επαναλήψεων. Κατόπιν νέες εκδόσεις παραδίδονται ανα τακτά χρονικά διαστήματα που μπορεί να έιναι από μερικές μέρες μέχρι μερικές εβδομάδες.
Έλεγχοι: Οι προγραμματιστές ακολουθούν την τακτική του πρώτα-ο-έλεγχος (test-first) που σημαίνει ότι πρώτα γράφουν τους ελέγχους αποδοχής και μετά τον ίδιο τον κώδικα. Οι πελάτες γράφουν τους λειτουργικούς ελέγχους και στο τέλος της κάθε επανάληψης όλοι αυτοί οι έλεγχοι πρέπει να εκτελούνται πετυχημένα.
Προγραμματισμός ανα ζευγάρια: Ίσως μία από τις πιο διάσημες πρακτικές του ακραίου προγραμματισμού όπου υποδεικνύεται 2 προγραμματιστές να κάθονται στο ίδιο μηχάνημα για να γράψουν κώδικα αλληλοβοηθόντας ο ένας τον άλλον και αλλάζοντας τακτικά θέσεις (ο ένας γράφει ο άλλος επιβλέπει).
Μεταφορά: Οι πελάτες μαζί με τους υπεύθυνους και τους προγραμματιστές δημιουργούν ένα φανταστικό σύστημα αποτελούμενο από ένα σύνολο μεταφορών πάνω στο οποίο βασίζεται η δημιουργία του μοντέλου.
Παρών πελάτης: Ο πελάτης δουλεύει καθ'όλη τη διάρκεια της μέρας με την ομάδα ανάπτυξης και απαντάει σε ερωτήσεις (κυρίως σχετιζόμενες με τις απαιτήσεις), εκτελεί ελέγχους αποδοχής και διαβεβαιώνει ότι η διαδικασία ανάπτυξης προχωράει ομάλα και σύμφωνα με τα χρονοδιαγράμματα.
Απλός σχεδιασμός: Οι προγραμματιστές γράφουν με γνώμονα την απλότητα στο σχεδιασμό αποφεύγοντας πολύπλοκες δομές όσο το δυνατόν.

Επανασχεδιασμός: Καθώς οι προγραμματιστές συνχεχίζουν να δουλεύουν ο σχεδιασμός του συστήματος θα πρέπει να εξελίσσεται ώστε να παραμένει απλός.

Συνεχής ολοκλήρωση: Οι προγραμματιστές πρέπει να προσαρμόζουν τον κώδικα τους όσο το δυνατόν πιο συχνά στο σύστημα. Όλα οι λειτουργικοί έλεγχοι θα πρέπει να ολοκληρώνονται επιτυχώς αλλιώς ο κώδικας να απορρίπτεται.

Συλλογική ιδιοκτησία: Ο κώδικας μπορεί να τροποποιηθεί από όλους τους προγραμματιστές ανα πάσα στιγμή.

Ανοιχτό περιβάλλον: Οι προγραμματιστές δουλεύουν σε ανοιχτό κοινό περιβάλλον απαρτιζόμενο από ξεχωριστά τερματικά περιφερειακά και κοινούς σταθμούς ανάπτυξης στο κέντρο.

40 ώρες/εβδομάδα: Οι απαιτήσεις θα πρέπει να επιλέγονται για κάθε επανάληψη έτσι ώστε ο κάθε προγραμματιστής να δουλεύει 40 ώρες την εβδομάδα χωρίς υπερωρίες.

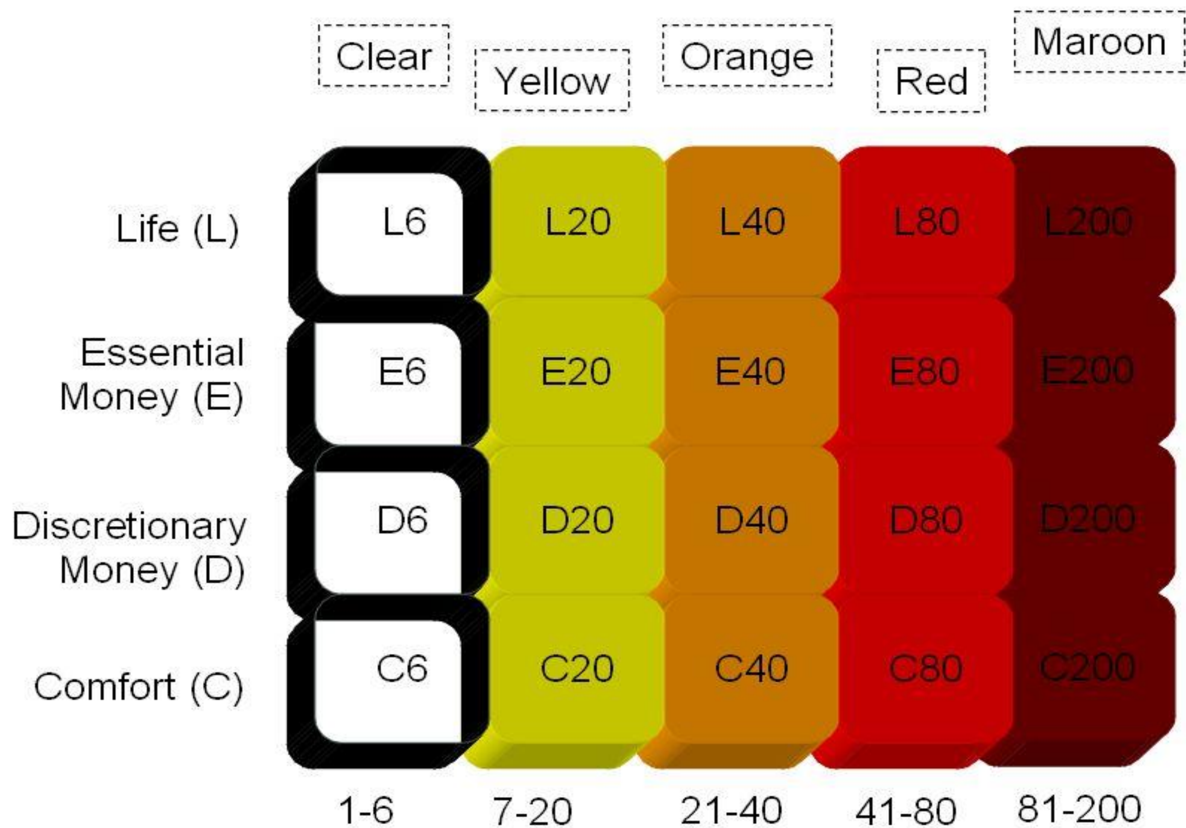
1.4.1.3 Οι Κρυσταλλικές Μέθοδοι(Crystal Methods)

Οι Κρυσταλλικές Μέθοδοι δημιουργήθηκαν στις αρχές του 90' με σκοπό να ξεπεραστούν οι αντιξοότητες που προέκυπταν από την έλλειψη επικοινωνίας. Συγκεκριμένα μία από τις βασικές αρχές που βασίστηκαν οι Μέθοδοι αυτές είναι ότι όσο πιο συχνά παραδίδεις κομμάτια του προϊόντος τόσο εξαλείφεται η εξάρτηση για «υποσχετικά» έγγραφα και με αυτόν τον τρόπο βελτιώνονται οι πιθανότητες επιτυχούς παράδοσης.

Στην πραγματικότητα οι Μέθοδοι αυτές αποτελούνται από μία οικογένεια ευέλικτων μεθοδολογιών όπως η Καθαρή, η Κίτρινη, η Πορτοκαλί και άλλες των οποίων τα μοναδικά χαρακτηριστικά μπορούν να ορισιτκοποιηθούν και να επιλεγούν σύμφωνα με κάποιους παράγοντες που επηρρεάζουν κάθε προϊόν. Τέτοιοι παράγοντες είναι το μέγεθος, ο αριθμός των μελών δηλαδή, που απαρτίζει μία ομάδα ανάπτυξης λογισμικού, η κρισιμότητα του συστήματος το οποίο πρόκειται να δημιουργηθεί και οι προτεραιότητες του εγχειρήματος. Στις Μεθόδους αυτές μπόρεσε και εμφυτεύτηκε η λογική ότι κάθε προϊόν, κάθε εργασία που επιχειρείται από μία ομάδα ανάπτυξης είναι μοναδική και διαφέρει από τις άλλες. Έτσι αυτές αφήνουν το περιθώριο σε κάθε ομάδα ανάπτυξης και ειδικότερα σε κάθε υπεύθυνο που χαράσσει την γραμμή ανάπτυξης να προσαρμόσει τις πολιτικές, τις πρακτικές και τις διαδικασίες ούτως ώστε να εξυπηρετηθούν αυτά τα μοναδικά χαρακτηριστικά που προαναφέρθηκαν.

Πολλά από τα αξιώματα των Κρυσταλλικών Μεθόδων περιλαμβάνουν λέξεις όπως ομαδικότητα, επικοινωνία, απλότητα αλλά και φράσεις όπως στόχευση στην συχνή προσαρμογή και βελτίωση της διαδικασίας. Όπως όλες οι άλλες Ευέλικτες Μεθοδολογίες έτσι και οι Κρυσταλλικές προωθούν την έγκαιρη και συχνή παράδοση λειτουργικού προϊόντος, την ενασχόληση του χρήστη σε μεγάλο βαθμό, την προσαρμοστικότητα και την εξάλειψη γραφειοκρατίας και άλλων περισπασμών από τον τελικό στόχο.

Όπως αναφέρθηκε και παραπάνω κάθε όψη που αντιπροσωπεύεται από ένα χρώμα δηλώνει και διαφορετική εκδοχή των διαδικασιών που παρ’όλα αυτά όλες τους βασίζονται σε έναν ίδιο πυρήνα. Στο σχήμα 7 αντικατοπρίζονται οι διαφοροποιήσεις των Κρυσταλλικών Μεθόδων σύμφωνα με τους παράγοντες που αναγράφησαν και πάνω.



Εικόνα 7 Crystal Methods

Οι πιο ευέλικτες Μέθοδοι όπως φαίνονται στο παραπάνω σχήμα είναι αυτές που βρίσκονται στα αριστερά και κάτω και όσο μεγαλώνει το μέγεθος του εγχειρήματος τόσο πιο δεξιά κινείται η Μεθοδολογία, που πρέπει να ακολουθηθεί, στο σχήμα και συνεπώς τόσο πιο αδιαφανής γίνεται. Επίσης όσο πιο κρίσιμο είναι το προϊόν που δημιουργείται τόσο πιο πάνω κινείται η Μεθοδολογία και γίνεται πιο «σκληρή».

1.5 Επίλογος

Σε αυτό το κεφάλαιο περιγράφηκαν κάποιες από τις γνωστότερες παραδοσιακές μεθοδολογίες ανάπτυξης λογισμικού και ακολούθησε ανάλογα η περιγραφή των γνωστότερων Ευέλικτων. Σε αυτές δόθηκε έμφαση στον κύκλο ζωής της ανάπτυξης που αυτές ακολουθούν και έγινε μία προσπάθεια να περιγραφούν συντόμως τα βασικά συστατικά που απαρτίζουν μία ευέλικτη μεθοδολογία, άλλα όπως διευκρινίστηκε χρησιμοποιούνται και από τις παραδοσιακές Μεθοδολογίες, και είναι οι Πρακτικές.

Στο επόμενο κεφάλαιο θα γίνει λόγος για την ποιότητα λογισμικού, πως αυτή καθορίζεται από τις πρακτικές που χρησιμοποιούνται στις Ευέλικτες Μεθοδολογίες και θα παρουσιαστούν κάποια εργαλεία – λογισμικά που χρησιμοποιούνται για αυτόν τον σκοπό.

2. Ευέλικτος Έλεγχος και Ποιότητα Λογισμικού – Χρήσιμα Εργαλεία

2.1 Εισαγωγή

Ένας πολυσυζητημένος παράγοντας που δίνεται έμφαση στον τομέα της Ανάπτυξης Λογισμικού είναι η ποιότητά του. Όπως είναι λογικό σε μία τόσο ακαθόριστη έννοια είναι φυσικό επακόλουθο να έχουν δοθεί αρκετοί ορισμοί και να έχουν χρησιμοποιηθεί αρκετά μετρικά συστήματα προκειμένου να τον προσδιορίσουν ποσοτικά και εν τέλει να χρησιμοποιηθεί σε συγκρίσεις. Έτσι πολλά μοντέλα έχουν προκύψει με το κάθε ένα από αυτά να περιλαμβάνει διαφορετικού είδους και πλήθος παραμέτρων. Αυτό λοιπόν το ατελείωτο θέμα συζήτησης και προβληματισμού θα ήταν αδύνατο να περιγραφεί σε μία εργασία. Για αυτό το λόγο ο κάθε ενδιαφερόμενος που θέλει να ασχοληθεί με την Ποιότητα Λογισμικού επιλέγει και μία οπτική από την οποία θα αναλύσει τα δεδομένα που που έχει διαθέσιμα. Παρακάτω θα προσδιορίσουμε την οπτική υπό την οποία σε αυτό το έγγραφο εξετάζουμε την ποιότητα και θα παρουσιαστούν κάποια εργαλεία που ενίοτε βελτιώνουν αυτήν.

2.2 Ποιότητα Λογισμικού

Η ποιότητα λογισμικού αναφέρεται σε δύο διαφορετικές έννοιες και αυτές μπορούν να βρεθούν όπου εν τέλει γίνεται λόγος για ποιότητα. Έτσι μπορούμε να πούμε ότι υπάρχει η λειτουργική ποιότητα και καταγίνεται με το πόσο ένα κομμάτι κώδικα ανταποκρίνεται στις απαιτήσεις και τις προδιαγραφές. Επίσης μπορεί αυτή η έννοια να περιλαμβάνει και τη σύγκριση του κώδικα με τα ανταγωνιστικά προϊόντα που υπάρχουν στην αγορά και το πόσο αυτό ταιριάζει στις ανάγκες της αγοράς. Η δεύτερη έννοια σχετίζεται με τη δομή του κώδικα και το πόσο εξυπηρετούνται οι λεγόμενες μη λειτουργικές απαιτήσεις. Αυτές με τη σειρά τους μπορούν να είναι η σταθερότητα, η ευκολία συντήρησης και φυσικά το πόσο ο κώδικας παράχθηκε σωστά.

Η δομική ποιότητα ενός Λογισμικού αξιολογείται μέσα από την ανάλυση του κώδικα που χρησιμοποιεί σε επίπεδο μονάδας, τεχνολογίας και συστήματος. Από την άλλη η λειτουργική ποιότητα αξιολογείται και κατ'επέκταση εξασφαλίζεται μέσω του Ελέγχου (Software Testing). Εμείς σε αυτό το κεφάλαιο θα ασχοληθούμε με τη

λειτουργική εξασφάλιση Ποιότητας και πως αυτή επιτυγχάνεται στις Ευέλικτες Μεθοδολογίες. Προτού όμως προχωρήσουμε στην ανάλυση του Ευέλικτου Ελέγχου ας επεκταθούμε λίγο περισσότερο στο τι είναι ή μπορεί να λογιστεί,μιας και όπως είπαμε είναι αρκετά γενικός ο όρος, ως Ποιότητα.

Έχουν υπάρξει πολλοί διαφορετικοί ορισμοί για το τι είναι Ποότητα Λογισμικού. Ένας πολύ σύντομος ο οποίος έχει διατυπωθεί κατά το ISO 9001 είναι η ικανότητα του προϊόντος Λογισμικού να εναρμονίζεται στις απαιτήσεις. Το γεγονός ότι πλήθος ορισμών έχουν δοθεί (και συνεχίζει να δίνονται) για την Ποιότητα Λογισμικού μπορεί σε ένα μόνο ασφαλές συμπέρασμα να μας οδηγήσει: «Η Ποιότητα Λογισμικού είναι κάτι το υποκειμενικό και μπορεί να αξιολογηθεί υπο πολλές διαφορετικές οπτικές». Αυτό το αντιλήφθησαν και στην Κοινοπραξία της Ποιότητας Λογισμικού (CISQ) και προκειμένου να δώσουν ακόμα έναν ορισμό διαπίστωσαν ότι θα ήταν πιο χρήσιμο και ίσως πιο διευκρινιστικό να παραθέσουν τα χαρακτηριστικά της Ποιότητας δηλαδή τους στόχους που μέσω αυτής επιτυγχάνονται.

2.2.1 Χαρακτηριστικά Ποιότητας

Το πρώτο χαρακτηριστικό είναι η αξιοπιστία, ένα γνώρισμα της προσαρμοστικότητας και της δομικής σταθερότητας. Η αξιοπιστία μετρά την πιθανότητα αποτυχιών του συστήματος και το επίπεδο κινδύνου. Επίσης μετρά τα ελαττώματα που προήλθαν από τροποποιήσεις στο Λογισμικό. Ο στόχος της μέτρησης και παρακολούθησης της αξιοπιστίας είναι να προλαμβάνει και να μειώνει το χρόνο «εκτός λειτουργίας» καθώς και τα λάθη που επηρρεάζουν άμεσα τους χρήστες. Δεύτερο χαρακτηριστικό είναι η αποτελεσματικότητα. Ο κώδικας και η αρχιτεκτονική είναι τα στοιχεία αυτά που προσδίδουν απόδοση στο τελικό προϊόν. Η αποδοτικότητα είναι ιδιαίτερα σημαντικός παράγοντας ειδικά σε εφαρμογές που εκτελούνται σε περιβάλλοντα απαιτητικά από άποψης ταχύτητας. Τέτοιες εφαρμογές είναι οι αλγοριθμικές και οι εφαρμογές δοσοληψιών που έχουν να κάνουν με παράλληλη πρόσβαση στις πηγές της εφαρμογής στις οποίες η απόδοση και η δυνατότητα ανάπτυξης θεωρούνται πρώτιστες απαιτήσεις. Η ανάλυση του κώδικα από πλευράς αποτελεσματικότητας και αναπτυξιμότητας μπορεί να δώσει συν τοις άλλοις μία εικόνα για το ρίσκο της εφαρμογής και να αξιολογηθεί ο χρόνος τελειοποίησης, παράγοντα σηματντικού για την ικανοποίηση του πελάτη. Άλλο χαρακτηριστικό είναι η δυνατότητα συντήρησης του προϊόντος. Αυτή με τη σειρά της περικλείει τις έννοιες της προσαρμοστικότητας, της

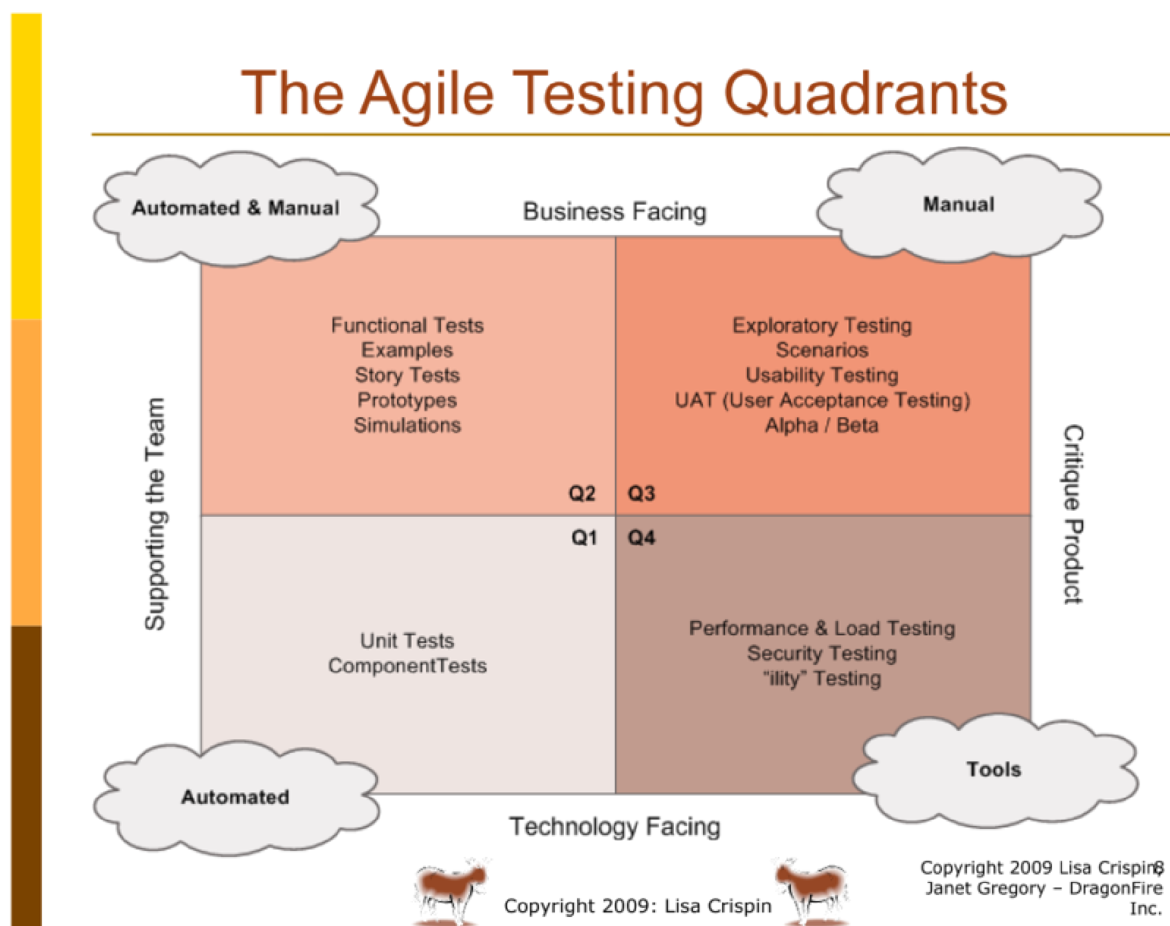
μεταφερσιμότητας (σε άλλα τρέχοντα περιβάλλοντα) και της δυνατότητας να χρησιμοποιηθεί και από άλλες ομάδες ανάπτυξης. Αυτό το χαρακτηριστικό είναι σημαντικό κυρίως σε εφαρμογές όπου το κυρίως ζητούμενο είναι η γρήγορη τοποθέτησή τους στην αγορά και η προσαρμογή τους στης αλλαγές που αυτή προστάζει. Ακόμα ένα χαρακτηριστικό είναι βέβαια η ασφάλεια. Ο έλεγχος αυτής γίνεται προκειμένου να μειωθούν οι πιθανότητες ζημιάς της εφαρμογής από κακόβουλους χρήστες. Τέλος, πέμπτο χαρακτηριστικό είναι το μέγεθος, το οποίο αν και δεν θεωρείται δεδομένα ένας ποιοτικός παράγοντας, το μέγεθος του κώδικα είναι κύριο χαρακτηριστικό διότι επηρρεάζει τα άλλα χαρακτηριστικά στο βαθμό που αυτά μπορούν να επιτευχθούν και κυρίως αυτό της συντήρησης. Συνδυαζόμενο με τα άλλα χαρακτηριστικά και χρησιμοποιώντας διάφορα εργαλεία το μέγεθος μπορεί να βοηθήσει στην αξιολόγηση του κώδικα και του φόρτου που πρέπει να επομισθεί η ομάδα ανάπτυξης.

2.3 Ευέλικτος Έλεγχος

Ως Ευέλικτος Έλεγχος ορίζεται η πρακτική του Ελέγχου Λογισμικού αλλά ακολουθώντας τις αρχές των Ευέλικτων Μεθοδολογιών. Ο Ευέλικτος Έλεγχος περιλαμβάνει μέλη από μία ομάδα που δουλεύουν σε κάθετο επίπεδο με ειδικούς πάνω στον Έλεγχο ώστε να διασφαλιστεί η παράδοση αξιόλογου προϊόντος στον πελάτη τακτικά και δουλεύοντας με ρυθμό ανάλογη των άλλων ομάδων. Το τελευταίο υπονοεί ότι κατά τις Ευέλικτες Μεθοδολογίες ο Έλεγχος δε θεωρείται μια φάση ξεχωριστή από αυτή της ανάπτυξης αλλά αποτελεί κομμάτι αυτής. Έτσι στις Ευέλικτες Μεθοδολογίες προβλέπονται ομάδα ή ομαδες αφοσιωμένες στον Έλεγχο για να εξασφαλιστεί η ποιότητα του Προϊόντος. Τα μέλη της ομάδας ελέγχου εφαρμόζουν την ειδικότητά τους εκμαιεύοντας παραδείγματα, δημιουργώντας υποθέσεις και σενάρια της επιθυμητής συμπεριφοράς του προϊόντος όπως αυτοί την αποκόμισαν από τον πελάτη. Σε συνεργασία με τις άλλες ομάδες οδηγούν τη δημιουργία κώδικα με γνώμονα την υλοποίηση των χαρακτηριστικών. Ο τρόπος που δουλεύουν οι ομάδες Ελέγχου είναι και αυτός όπως και αυτός που εργάζονται οι άλλες ομάδες των Ευέλικτων Μεθοδολογιών, δηλαδή προσθετικός και επαναληπτικός. Έτσι με αυτόν τον τρόπο χτίζεται σταδιακά το κάθε γνώρισμα μέχρις ότου να είναι έτοιμο για να παραχθεί. Παρακάτω θα εξηγηθεί ένα μοντέλο κατηγοριοποίησης και συγχρόνως βοήθημα των Ευέλικτων Ελέγχων, τα Τεταρτημόρια Ευέλικτων Ελέγχου (Agile Testing Quadrants).

2.3.1 Τεταρτημόρια Ευέλικτων Ελέγχου

Το μοντέλο του Τεταρτημορίου παρουσιάστηκε αρχικά στο βιβλίο Agile Testing των Lisa Crispin και Janet Gregory. Αυτό το μοντέλο σκοπό έχει να ταξινομήσει τα πεδία όπου εφαρμόζεται ο Ευέλικτος Έλεγχος αλλά και να βοηθήσει στο πως μπορεί να εφαρμοστεί και σε ποιες περιπτώσεις ο Ευέλικτος Έλεγχος. Στην εικόνα 8 φαίνεται τα Τεταρτημόρια όπως αυτό σχεδιάστηκε και παρακάτω θα προσπαθήσουμε να το αξηγήσουμε ώστε κατόπιν να μπορέσουμε να αναφερθούμε και στα εργαλεία που χρησιμοποιούνται στον Ευέλικτο Έλεγχο ανα κατηγορία.



Εικόνα 8 Τεταρτημόρια Ευέλικτου Ελέγχου

Το πρώτο τεταρτημόριο (Q1) αποτελείται από τις περιπτώσεις ελέγχου (test cases) οι οποίες καθοδηγούνται από την τεχνολογία και εφαρμόζονται ώστε να υποστηρίξουν την ομάδα. Αυτό εστιάζει κυρίως στην εσωτερική ποιότητα του κώδικα. Ο Έλεγχος Μονάδας αφού επιτευχθεί σε αυτό το τεταρτημόριο λειτουργεί ως ένα δίχτυ προστασίας για το προϊόν και βοηθά την ομάδα ανάπτυξης και τον κάθε προγραμματιστή ξεχωριστά να αντιληφθεί το σενάριο σωστά. Επίσης η εκτέλεση των ελέγχων του 1^{ου} τεταρτημορίου δίνει απ’ευθείας ανταπόκριση για την

ορθότητα του παραγόμεου συστήματος. Όταν η πρακτική του προγραμματισμού οδηγούμενου από τον έλεγχο υιοθετείται ο σχεδιασμός γίνεται ισχυρότερος και φυσικά ορθότερος. Ο Έλεγχος σε αυτό το τεταρτημόριο χτίζεται μέσα στον κώδικα και στο τέλος της υλοποίησης του κώδικα ελάχιστα λάθη θα παρουσιαστούν. Σε αυτό το τεταρτημόριο τα είδη των ελέγχων που βρίσκουμε είναι τα:

1) Έλεγχοι Μονάδας(Unit Tests) που περιλαμβάνουν τον έλεγχο ενός μέρους του κώδικα και επιβεβαιώνουν ότι πληρούνται οι απαιτήσεις.
2) Έλεγχοι Στοιχείου(Component Tests) που περιλαμβάνουν τον έλεγχο από αρχιτεκτονικής πλευράς για να επιβεβαιώσουν ότι τα στοιχεία μπορούν και λειτουργούν-συνεργάζονται μεταξύ τους.

Στο δεύτερο τεταρτημόριο βρίσκονται οι έλεγχοι που υποστηρίζουν την ομάδα και δημιουργούνται με βάση την επιχειρηματική λογική του προϊόντος. Σε αυτό το τεταρτημόριο οι έλεγχοι εστιάζουν στην απόσπαση, την εκμαίευση των απαιτήσεων. Οι προγραμματιστές εξακολουθούν να γράφουν κώδικα όσο δεν εντοπίζονται λάθη και τα αποτελέσματα είναι τα αναμενόμενα. Από τη στιγμή που οι περιπτώσεις που εξετάζονται είναι επειχηρηματικές έτσι και η ανάπτυξη κώδικα πορευέται με αυτό υπ’όψην και με σκοπό να υλοποιηθούν οι απαιτήσεις δίχως εμπόδια. Στουθς ελέχχους αυτού του τεταρτημορίου είναι επίσης πο´´υ πιθανή η συνεργασία μεταξύ προγραμματιστή και ελεγχτή. Οι τύποι ελέγχου αυτού υου τεταρτημορίου είναι οι εξής:

1) Έλεγχος παραδειγμάτων πιθανών σεναρίων και ροών.
2) Έλεγχος της Εμπειρίας Χρήστη όπως των προτότυπων.
3) Έλεγχος ζεύγους

Το επόμενο τεταρτημόριο αποτελείται από ελέγχους επιχειρηματικής λογικής που εφαρμόζονται στο προϊόν. Ο κύριος σκοπός των ελέγχων αυτού του τεταρτημορίου είναι να δώσουν αποτελέσματα προς αξιολόγηση στα τεταρτημόρια 1 και 2. Οι περιπτώσεις που εξετάζονται σε αυτού του είδους τους ελέγχους μπορούν να χρησιμοποιηθούν σαν βάση για τους αυτοματοποιημένους ελέγχους. Το προϊόν μπορεί να εκτιμηθεί επειδή σε αυτή τη φάση οι έλεγχοι γίνονται σε συνθήκες πραγματικής χρήσης. Η διαδικασία των ελέγχων σε αυτό το σημείο είναι ευμετάβλητες και μπορούν να προσαρμοστούν ανάλογα με τις ανάγκες που προκύπτουν. Ο μεγάλος αριθμός των επαναληπτικών εξετάσεων μπορεί και επιφέρει άνεση στις ομάδες και τους υπύθυνους σχετικά με την έκβαση της υλοποίησης και αυτός ο κύκλος ελέγχων σιγουρεύει τη γρήγορη επίληψη κάποιου ζητήματος. Το προϊόν επίδειξης (demo) μπορεί να δοθεί ακόμα και ανολοκλήρωτο ώστε να αποτιμηθεί εξ’αρχής η λειτουργικότητα του. Κάποιοι έλεγχοι διευκρινιστικοί μπορούν επίσης να γίνουν σε συνεργασία με τον πελάτη προς την

πλευρά του προαναφερθέντος στόχου. Τα είδη των ελέγχων που συναντόνται σε αυτό το τεταρτημόριο είναι:

1) Διερευνητικοί Έλεγχοι
2) Έλεγχοι Χρηστικότητας
3) Έλεγχοι κατά ζεύγη με τον πελάτη
4) Έλεγχοι αποδοχής από τον πελάτη
5) Έλεγχοι συνεργασίας

Το τελευταίο τεταρτημόριο περιέχει τους ελέγχους από πλευράς τεχνολογίας που εφαρμόζονται στο προϊόν.Σε αυτούς η προσοχή στρέφεται στις μη λειτουργικές απαιτήσεις όπως η απόδοση, η σταθερότητα, η ασφάλεια. Αυτοί είναι οι τελικοί έλεγχοι που είναι υπρύθυνοι πρότου παραδοθεί το τελικό προϊόν. Η εφαρμογή φτιάχνεται με σκοπό να παραδοθεί με τις αξίες που έχουν τεθεί και αυτές περιλαμβάνουν και μη λειτουργικές απαιτήσεις όπου επιτυγχάνονται με τη βοήθεια αυτών των ελέγχων οι οποίοι μπορεί να είναι:

1) Έλεγχοι μη λειτουργικών απαιτήσεων όπως αποδοσης και πίεσης.
2) Έλεγχοι ασφαλείας με κύριο προσανατολισμό την αυθεντικοποίηση και την αποτροπή παράνομης πρόσβασης.
3) Έλεγχοι υποδομής
4) Έλεγχοι μεταφοράς δεδομένων
5) Έλεγχοι ανάπτυξης
6) Έλεγχοι φόρτωσης

Συνοψίζοντας τα παραπάνω ανάλογα με τις απαιτήσεις του συστήματος, τους κινδύνους, τις προτεραιότητες και φυσικά τον σκοπό του, τα παραπάνω τεταρτημόρια πρέπει να διαλεχθούν και να υλοποιηθούν. Σε αυτήν την υλοποίηση μπορε΄υν να βοηθήσουν εργαλεία κατάλληλα τα οποία κα θα παρουσιασούν μετέπειτα. Η υλοποίηση των τεταρτημορίων βοηθάει στην ομαδική επίλυση ζητημάτω από τους ελεγκτές, τους προγραμματιστές και τους πελάτες και η γνώση αυτών των διαδικασιών που ορίζονται από το μοντέλο που περιγράψαμε και ο συνδυασμός εργαλείων και διαφορετικών ελέγχων είναι κάτι που υπόκειται στις ευθύνες της ομάδας ελέγχου.

2.4 Εργαλεία Ευέλικτου Ελέγχου

Σε αυτό το σημείο θα αναφερθούμε σε κάποια εργαλεία που χρησιμοποιούνται από τις ομάδες που απασχολούνται με τον Ευέλικτο Έλεγχο. Οι δύο κύριοι λόγοι που χρησιμοποιούνται τέτοιου είδους εργαλεία είναι, ο πρώτος η

καλύτερη λειτουργία της ομάδας πράγμα που σημαίνει καλύτερη απόδοση και δεύτερο προς διευκόλυνση αυτής σε ζητήματα που προκύπτουν κατά την εκτέλεση ελέγχων. Η διαδικασία του Ελέγχου όπως έγινε σαφές και στην προηγούμενη ενότητα δεν είναι καθήκον μόνο μιας ομάδας αλλά λαμβάνει χώρα σε διάφορα σημεία κατά τον κύκλο ζωής της ανάπτυξης και αυτό σημαίνει ότι άτομα από διαφορετικές ομάδες και με διαφορετικό είδος γνώσεων εφαρμόζουν κάποια από αυτά. Συνεπώς όλα τα εργαλεία που θα αναφερθούν δεν απευθύνονται στον καθένα που ελέγχει κάποιο τμήμα λογισμικού όπως και δεν εφαρμόζονται σε κάθε περίσταση που χρειάζεται να ελεγχθεί μία εφαρμογή. Μάλιστα σύμφωνα με αυτή τη λογική θα προσπαθήσουμ ένα τα κατηγοριοποιήσουμε παρακάτω και βεβαίως θα αναφερθεί και από ποια μέλη είναι φρόνιμο να χρησιμοποιούνται. Το μοντέλο που περιγράφηκε πιο πάνω είναι ιδανικό προκειμένου να διαχωριστούν λοιπόν εκτός από τα είδη των ελέγχων και τα εργαλεία που χρησιμοποιούνται για την εκτέλεση αυτών.

2.4.1 Έλεγχοι Μονάδων και Στοιχείων

Πρόκειται αναμφισβήτητα για την πολυπληθέστερη ομάδα εργαλείων. Για αυτού του είδους υπάρχουν πραγματικά αμέτρητες επιλογές για κάθε σχεδόν γλώσσα προγραμματισμού. Εμείς εδώ θα ασχοληθούμε με αυτά που προορίζονται για την JAVA. Ως έλεγχος μονάδας ορίζεται ο έλεγχος που μπορεί να γίνει στο μικρότερο λειτουργικό κομμάτι κώδικα. Πρόκειται λοιπόν για τον πιο στοιχειώδη, από πλευράς εκτέλεσης κώδικα, έλεγχο.

2.4.1.1 JUnit

Το JUnit είναι ένα πλαίσιο ελέγχου μονάδας για την JAVA. Έχει παίξει σημαντικό ρόλο στην ανάπτυξη λογισμικού και ειδικά στην τεχνοτροπία της ανάπτυξης με οδηγό τους ελέγχους (test-driven development). Ανήκει στην οικογένεια των πλαισίων τα οποία ονομάζονται xUnit και έχουν προέλθει από τα SUnit.

Το JUnit έρχεται σαν ένα συμπιεσμένο αρχείο τύπου JAR και είναι ιδιαίτερα εύκολο στη χρήση του. Η ευκολία στη χρήση του έγκειται στο γεγονός ότι μία μία κλάση ορίζεται ως κλάση ελέγχου από τη στιγμη που υποσημειωθεί (annotation) μία τουλάχιστον μέθοδός της με το εξής υπόμνημα «@Test». Το JUnit έχει πια ολοκληρωθεί με όλα τα περιβάλλοντα ανάπτυξης λογισμικού (IDEs) και μπορεί

έτσι εύκολα να εκτελεούνται οι έλεγχοι μέσα από αυτά. Μάλιστα μπορεί να χρησιμοποιηθεί σε ολόκληρες σουίτες και να δημιουργηθούν αυτοματοποιημένοι μαζικοί έλεγχοι. Με αυτόν τον τρόπο μπορεί να «τρέχουν» οι έλεγχοι κάθε φορά που είναι να τροποποιηθεί ή να προστεθεί νέος κώδικας και να ελέγχεται εκτός από τα νέα πρόσθετα χαρακτηριστικά αν μεταβλήθηκε, στην ουσία αν ζημιώθηκε η ήδη υπάρχουσα λειτουργικότητα. Έτσι με αυτόν τον τρόπο μπορεί σε ένα πρώτο επίπεδο να ελεγχθεί ο κώδικας από τον ίδιο τον προγραμματιστή.

```
public class Calc {
 public long add(int a, int b) {
  return a+b;
 }
}

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalcTest {
 @Test
 public void testAdd() {
  assertEquals(5, new Calc().add(2, 3));
 }
}
```

2.4.1.2 TestNG

Το TestNG είναι ένα ακόμα πλάισιο το οποίο έχει βασιστεί στο JUnit που όμως παρέχει σαφώς περισσότερα και πιο εξειδικευμένα χαρακτηριστικά. Για την ακριβεία το TestNG δεν είναι ένα πλαίσιο μόνο για ελέγχους μονάδων αλλά και πιο ολοκληρωμένων κομματιών κώδικα και συνεπώς μπορεί να χρησιμοποιηθεί για τον έλεγχο διαφορετικών κλάσεων και της αλλαηλεπίδρασης μεταξύ τους (έλεγχος στοιχείων), διαφόρων πακέτων, ακόμα και διαφόρων άλλων εξωτερικών πλαισίων όπως εξυπηρετητών εφαρμογών και έτσι μπορεί να επιτύχει ολοκληρωμένους ελέγχους (integrated testing).

Είναι και αυτό αρκετά εύχρηστο, χρησιμοποιώντας σε μεγάλο βαθμό τις υποσημειώσεις της JAVA και ευδιάβαστα αρχεία τύπου xml προκειμένου να προωθήσουν πληροφορία στις μεθόδους ελέγχου. Μπορεί και αυτό να συνδυαστεί με εργαλεία ανάπτυξης(Eclipse, IntelliJ) και χτισίματος (Ant, Maven) κώδικα και φυσικά να δημιουργηθούν ολόκληρα πακέτα αυτοματοποιημένων και παραμετροποιημένων ελέγχων.

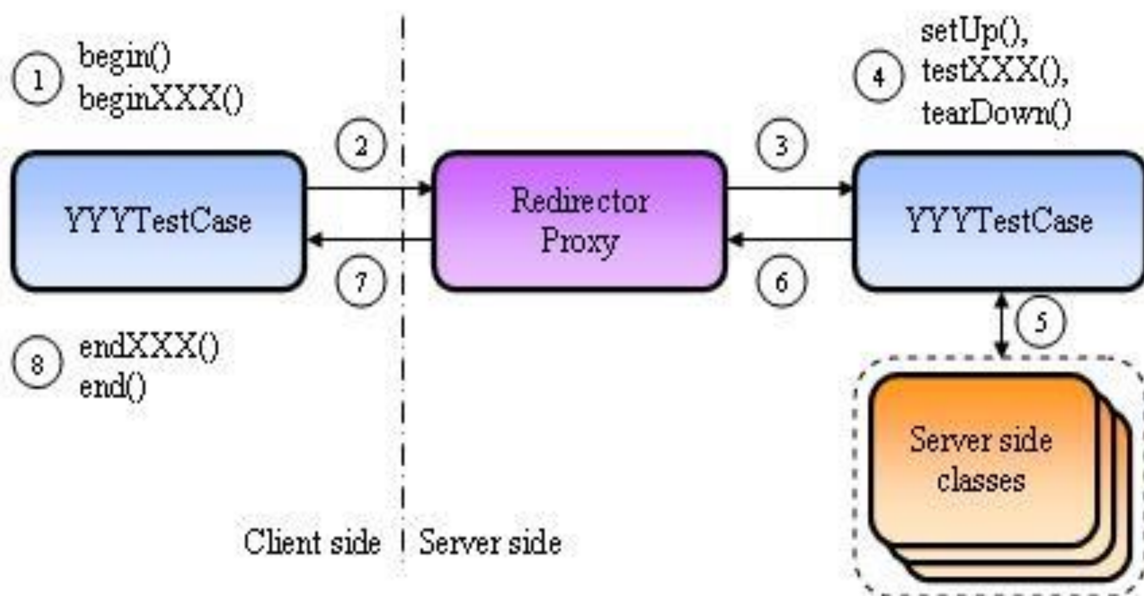
```
public class WebTestFactory {
  @Factory
  public Object[] createInstances() {
   Object[] result = new Object[10];
   for (int i = 0; i < 10; i++) {
      result[i] = new WebTest(i * 10);
    }
    return result;
  }
}

public class WebTest {
  private int m_numberOfTimes;
  public WebTest(int numberOfTimes) {
    m_numberOfTimes = numberOfTimes;
  }

  @Test
  public void testServer() {
   for (int i = 0; i < m_numberOfTimes; i++) {
     // access the web page
    }
  }
}
```

2.4.1.3 Cactus

Ακόμα ένα πλάισιο ελέγχου μονάδας είναι το Cactus το οποίο όμως αν και χρησιμοποιεί και αυτό με τη σειρά του άλλα πλαίσια ελέγχου η βασική του λειτουργία είναι να παρέχει και να βελτιώνει την υποστήριξη του πλαισίου JEE σε ελέγχους μονάδας. Το συγκεκριμένο πλαίσιο λοιπόν έχει ως στόχο την αξιολόγηση κώδικα που βρίσκεται στην πλευρά του εξυπηρετητή. Ειδικότερα το Cactus δημιουργεί ολοκληρωμένους ελέγχους μονάδας. Αν και βασίζεται επίσης στο JUnit το Cactus λειτουργεί μέσα σε περιβάλον εξυπηρετητή (JEE servers) και αυτό γιατί θέλει να ελέγξει στοίχεια (components) τα οποία διαχειρίζονται από τον container. Τέτοια στοιχεία μπορεί να είναι τα EJBs, Servlets, Tag Libs, Filters.

Το Cactus επιτυγχάνει τη δημιουργία και εκτέλεση αυτών των ελέγχων με τη χρήση της αρχιτεκτονικής προώθησης (redirection). Συγκεκριμένα είτε πρόκειται

για JSP σελίδα, είτε για απλό Servlet (Http Listener) είτε για κάποιο φίλτρο, το Cacuts αφήνει την εξυπηρετούμενη μονάδα λογισμικού,πελάτη να ανοίξει 2 συνδέσεις στην κλάση προώθησης που αυτομάτως δημιουργεί. Αυτό το κάνει διότι το ένα κανάλι σχρησιμοποιείται για να λάβει η κλάση που κάλεσε το αποτέλεσμα της μεθόδου που κάλεσε και το δεύτερο για να λάβει το αποτέλεσμα του ελέγχου. Δηλαδή διαχωρίζει τα αποτελέσματα ελέγχου και ελεγχόμενης κλάσης-μεθόδου και αυτό διότι έτσι επιτυγχάνεται η λήψη των εξειρέσεων (Exception) από τον πελάτη κάτι που σε αντίθετη περίπτωση δε θα γινόταν αφού όλα τα στοιχεία που καλούνται απομακρυσμένα σε έναν εξυπηρετητή συνήθως περικλείουν τις εξαιρετικές καταστάσεις (wrap) σε μία και μόνο δική τους. Κάτι τέτοιο δυσκολεύει τον έλεγχο σε επίπεδο διόρθωσης λαθών (debugging) κάνοντας πολύ αργότερο αν αναγκαστεί ο υπεύθυνος του ελέγχου να ανατρέξει τα πρακτικά (logs) στον εξυπηρετητή ή ακόμα και αδύνατον αν δεν έχει πρόσβαση σε αυτά κατά τη διάρκεια του ελέγχου. Τα αποτελέσματα των ελέγχων αποθηκεύονται σε μια κλάση που ονομάζεται SevletContext τα οποία λαμβάνονται από τη δεύτερη σύνδεση που αναφέραμε. Στην εικόνα που ακολουθεί φαίνεται η βασική αρχιτεκτονική του πλαισίου.



Εικόνα 9 Αρχιτεκτονική Cactus

Για την εκτέλεση των ελέγχων χρησιμοιποιούνται κάποιες συγκεκριμένες μεθόδους που ακολουθούν μία ονοματολογία (callback methods) όπως και στους JUnit ελέγχους άλλοστε. Κάποιοες βασικές είναι οι setup() και tearDown() οι οποίες είναι προαιρετικές και καλούνται πριν και μετά αντίστοιχα από τους ελέγχους, η testXXX() που είναι και η βασικότερη ‘οπου το XXX πρακτικά είναι το

όνομα της μεθόδου που θα εξεταστεί και φυσικά είναι υποχρεωτική αλλά και οι μέθοδοι όπως begin(), beginXXX(), end() και endXXX() οι οποίες καλούνται αντίστοιχα πριν και μετά την εκτέλεση κάποιου συγκεκριμένου ελέγχου ή την αρχή και το τέλος της εκτέλεσης όλου του πακέτου ελέγχου που πιθανώς να έχει δημιουργηθεί. Αυτές οι μέθοδοι διαφέρουν από τις setup και teardown στο ότι εκετελούνται στη μεριά του πελάτη και χρησιμοποιούνται για να ορίσουν παραμέτρους του Http, cookies, κεφαλίδες του Http, URL τοποθεσίες και στη συνέχεια να ελεγχθούν και αυτές με τη σειρά τους στο τέλος του ελέγχου.

```
package org.apache.cactus.sample.ejb;

import javax.naming.*;
import javax.rmi.*;
import junit.framework.*;

import org.apache.cactus.*;

public class ConverterTest extends ServletTestCase
{
   private Converter converter;

   public ConverterTest(String name)
   {
      super(name);
   }

   public static Test suite()
   {
      return new TestSuite(ConverterTest.class);
   }

   public void setUp()
   {
      Context ctx = new InitialContext();
      ConverterHome home = (ConverterHome)
         PortableRemoteObject.narrow(ctx.lookup("java:comp/ejb/Converter"),
         ConverterHome.class);
      this.converter = home.create();
   }

   public void testConvert() throws Exception
   {
      double dollar = this.converter.convertYenToDollar(100.0);
      assertEquals("dollar", 1.0, dollar, 0.01);
   }
}
```

2.4.2 Λειτουργικοί Έλεγχοι και Προσομοιώσεις

Κατά τους ελέγχους αυτούς το ζητούμενο είναι να αξιολογηθούν πιο ολοκληρωμένα κομμάτια κώδικα από ότι στους ελέγχους μονάδας και ολοκλήρωσης. Αυτή η ολοκληρωση μπορεί να τεθεί υπό το πρίσμα 2 διαφορετικών διαστάσεων, αυτή δηλαδή που έχει να κάνει με το πόσα διαφορετικά κομμάτια κώδικα χρησιμοποιούνται προκειμένου να περατωθεί το έργο του κώδικα, όπου μπορεί αυτά να εκτελούν ανεξάρτητα μεταξύ τους υποεργασίες και ως προς το χρόνο που εκτελούνται δηλαδή κατά την εξασφάλιση της Ποιότητας και όχι της δημιουργίας κώδικα.

2.4.2.1 HttpUnit

Η μεγαλύτερη δυσκολία στους λειτουργικούς ελέγχους είναι ότι σε αυτούς εμπλέκονται και άλλα πλην των ελεγχόμενων κομμάτια κώδικα. Πολλές φορές μάλιστα αυτά τα κομμάτια μπορεί να είναι κώδικας ιδιαίτερα πολύπλοκος ή ακόμα και να μην έχουμε πρόσβαση σε αυτού τις πηγαίες κλάσεις. Κάτι τέτοιο όπως γίνεται αντιληπτό συμβαίνει όταν θέλουμε να ελέγξουμε διεπιφάνειες χρήστη. Σε αυτού του είδους τη λειτουργία πάντα λαμβάνουν μέρος τα στοιχεία που παρέχει ο Web εξυπηρετητής. Μάλιστα αυτά συχνά είναι προσβάσιμα από πολλούς χρήστες πράγμα που κάνει τον κώδικα του σαφώς πολύπλοκο προκειμένου να υποστηρίζει τέτοιου είδους λειτουργικότητα. Πάνω σε αυτήν λοιπόν τη δύσκολη περίπτωση έρχεται να προστεθεί και το δεδομένο ότι ο έλεγχος θέλουμε να είναι ανεξάρτητος από τον περιηγητή (browser) για να μπορούν να αυτοματοποιηθούν οι έλεγχοι και να ελεγχθεί καθαρά ο κώδικας που έχει να κάνει με τη δημιουργία της όψης του χρήστη και όχι όπως αυτή παρουσιάζεται από τον οποιονδήποτε πελάτη-περιηγητή του κώδικα που πρόκειται να ελέγξουμε.

Για να αποφύγουμε τον περιηγητή και να έχουμε πρόσβαση σε ένα web πρόγραμμα το HttpUnit αποτελεί μια πολύ γνωστή και πρακτική λύση. Είναι γραμμένο σε JAVA και μπορεί να προσομοιώσει τα σχετικά με τη συμπρ=εριφορά του περιηγητή κομμάτια περιλαμβάνοντας σε αυτά την υποβολή φορμών, την εκτέλεση JavaScript, την βασική λειτουργία αυθεντικοποίησης http, τη δημιουργία και ανάγνωση cookies και την αυτόματη επανεκατεύθυνσης σελιδών. Η αυτοματοποίση μπορεί να γίνει μέσα από την αξιολόγηση των αποτελεσμάτων σε JAVA κώδικα όπου μπορεί να δεχτεί τα αποτελέσματα ως απλό κείμενο, ως XML DOM ή ακόμα και κλάσεις που περιλαμβάνουν φόρμες, πίνακες και διασυνδέσεις.

Η συνεργασία του με κάποιο πλαίσο Ελέγχου όπως το JUnit είναι που το κάνει ικανό να χρησιμοποιηθεί και μέσα σε πακέτο αυτοματοποιημένων ελέγχων.

```
 WebConversation wc = new WebConversation();
   WebResponse   resp = wc.getResponse(
"http://www.httpunit.org/doc/cookbook.html" ); // read this page
   WebLink       link = resp.getLinkWith( "response" );       // find the link
   link.click();                                               // follow it
   WebResponse   jdoc = wc.getCurrentPage();      // retrieve the referenced page

  WebTable table = resp.getTables()[0];
   assertEquals( "rows", 4, table.getRowCount() );
   assertEquals( "columns", 3, table.getColumnCount() );
   assertEquals( "links", 1, table.getTableCell( 0, 2 ).getLinks().length );

WebForm form = resp.getForms()[0];      // select the first form in the page
form.setParameter( "Food", "Italian" );      // select one of the permitted values for
food
form.removeParameter( "CreditCard" );        // clear the check box
form.submit();                            // submit the form
   assertEquals( "La Cerentolla", form.getParameterValue( "Name" ) );
   assertEquals( "Chinese",       form.getParameterValue( "Food" ) );
   assertEquals( "Manayunk",      form.getParameterValue( "Location" ) );
   assertEquals( "on",           form.getParameterValue( "CreditCard" ) );
```

2.4.2.2 Abbot

Το Abbot παρέχει και αυτό ένα πλαίσιο για ελέγχους των διεπιφανειών χρήστη ανεξάρτητα από το σημείο που βρίσκεται ο κώδικας. Στην περίπτωση που χρησιμοποιείται η οδηγούμενη από έλεγχο ανάπτυξη τότε μπορεί ν αχρησιμοποιηθεί ώστε να κατασκευαστούν οι έλεγχοι μονάδας. Σε περίπτωση όμως που υπάρχει ήδη κώδικας αλλά δεν υπάρχουν καθόλου έλεγχοι μονάδας μπορεί να χρησιμοποιηθεί το επίπεδο δέσμης λειτουργιών (scripting level) του Abbot για να ξεκινήσουν φτιάχνεται ο σκελετός των λειτουργικών ελέγχων γύρω από την εφαρμογή μέχρι το σημείο που θα είναι αρκετά σταθερή ώστε να υποστηριχτεί η τροποποίηση και κατόπιν η δημιουργία ελέγχων μονάδας.

Γενικά ο έλεγχος με το Abbot αποτελείται από την λήψη αναφορών κάποιων στοιχείων της διεπιφάνειας και εκτελώντας κάποιες ενέργειες χρήστη πάνω σε αυτές ή κάνοντας κάποιες παραδοχές εικασίες για τις τιμές τους. Προς διευκόλυνση της διαδικασίας το πλαίσιο παρέχει ακόμα το ComponentReferences για να μπορεί να χειριστεί καταλλήλως ένα στοιχείο διεπιφάνειας ακόμα και όταν δεν υπάρχει. Επίσης παρέχονται αντικείμενα που λειτουργούν αυτόματα (robot-like objects) τα οποία γνωρίζουν πώς να εκτελέσουν ενέργειες σε επίπεδο χρήστη σε διάφορα συστατικά της διεπιφάνειας.Αυτές οι λειτουργίες μπορούν να γίνουν

είτε σε υψηλό επίπεδο, χρήσιμο για λειτουργικούς ελέγχους όπου είναι θεμιτό πολλές φορές να μην εμπλέκεται στον έλεγχο η ομάδα ανάπτυξης, είτε και σε πιο χαμηλό οπότε και μπορεί το πλαίσιο με τη βοήθεια της JAVA να εκτελεστεί μέσα από ένα άλλο πλάσιο ελέγχου.

```
// Suppose MyComponent has a text field and a button...
    MyComponent comp = new MyComponent();
// Display a frame containing the given component
    showFrame(comp);

JTextField textField = (JTextField)getFinder().
        find(new ClassMatcher(JTextField.class));
JButton button = (JButton)getFinder().find(new Matcher() {
    public boolean matches(Component c) {
       // Add as much information as needed to distinguish the component
       return c instanceof JButton &&((JButton)c).getText().equals("OK");
    }
 });
    JTextComponentTester tester = new JTextComponentTester();
    tester.actionEnterText(textField, "This text is typed in the text
field");
    tester.actionClick(button);
    // Perform some tests on the state of your UI or model
    assertEquals("Wrong button tooltip", "Click here to accept",
button.getToolTipText());
```

2.4.2.3 Mock Objects

Τα Mock Objects είναι μία τεχνική η οποία χρησιμοποιείται όταν είναι δύσκολο να απομονωθεί κώδικας ώστε να ελεγχθεί μεμονωμένα χωρίς να επηρεάζεται από άλλα κομμάτια. Έτσι σε αυτήν την τεχνική αντικαθίσταται ο κώδικας που θέλουμε να αποφύγουμε με υλοποιήσεις σταθερού (dummy) κώδικα που προσομοιώνει την πραγματική υλοποίηση. Αυτές οι υλοποιήσεις περνώνται στον στοχευμένο κώδικα οι οποίες τον ελέγχουν από μέσα και από αυτό το γεγονός βγήκε ο όρος Endo-Testing. Αυτός ο τρόπος είναι παρόμοιος με το να γράφουμε αποκόμματα (stubs) με τη διαφορ'α όμως ότι μπορούμε και ελέγχουμμε με έναν πιο ευέλικτο τρόπο (fine granularity) από ότι συνήθως και επίσης χρησιμοποιούμε τους ελέγχους και τα δημιουργηθέντα αποκόμματα ώστε να οδηγηθούμε και στον κώδικα που θα βγει στην παραγωγή (test driven development).

Συνεπώς μπορούμε να πούμε για τα Mock Objects ότι είναι μια αντικατάσταση της πραγματικής υλοποίησης του κώδικα που προσδιορίζει τις συσχετίσεις ανάμεσα στις κλάσεις (domain). Θα πρέπει φυσιολογικά να είναι απλότερος από τον πραγματικό κώδικα και όχι απλά να είναι μία απλή αντιγραφή αλλά να αφήνει τον ενδιαφερόμενο να του αλλάξει και τιμές του ώστε να τον

βοηθήσει στους ελέγχους. Δίνεται ιδιαίτερη έμφαση στην απλότητά του και όχι στην ολοκλήρωση που αυτός θα παρουσιάσει μιας και δεν είναι κώδικας παραγωγής. Για παράδειγμα μία κλάση που ανήκει σε μία συλλογή τέτοιων αντικειμένων μπορεί πάντα να γυρνά το ίδιο αποτέλεσμα από μία μέθοδο ανεξάρτητα από τις παραμέτρους που περνώνται σε αυτήν.

Γενικά η χρησιμοποίηση αυτής της τεχνικής ελέγχου μπορούμε να πούμε ότι είναι πολύ σταθερή ως προς τον τρόπο που υλοποιείται και εφαρμόζεται και αποτελείται λίγο ή πολύ από τα παρακάτω βήματα. Αρχικά δημιουργούνται τα Mock Objects. Κατόπιν καθορίζονται οι τιμές που τα προσδιορίζουν (state). Μετά ορίζεται το τι περιμένεις από αυτά τα αντικείμενα. Ύστερα καλείται ο κώδικας που θέλουμε να ελεγχθεί με αυτά τα αντικείμενα σαν παραμέτρους. Τέλος επιβεβαιώνεται η συνέπεια των Mock Objects ως προς τις προσδοκίες μας.

```
public void printPersonReport(Person person, PrintWriter writer) {
writer.println(person.getName());
writer.println(person.getAge());
writer.println(person.getTelephone());
}
public void printPersonReport(Person person, PrintWriter writer) {
person.printDetails(writer);
}


public class Person {
public void printDetails(PrintWriter writer) {
writer.println(myName);
writer.println(myAge);
writer.println(myTelephone);
}
...
}
public void handleDetails(PersonHandler handler) {
handler.name(myName);
handler.age(myAge);
handler.telephone(myTelephone);
}
void testPersonHandling() {
myMockHandler.setExpectedName(NAME);
myMockHandler.setExpectedAge(AGE);
myMockHandler.setExpectedTelephone(TELEPHONE);
myPerson.handleDetails(myMockHandler);
myMockHandler.verify();
}
void testPersonHandler() {
myMockPrintWriter.setExpectedOutputPattern(
".*" + NAME + ".*" + AGE + ".*" + TELEPHONE + ".*");
```

```
myHandler.name(NAME);
myHandler.age(AGE);
myHandler.telephone(TELEPHONE);
myHandler.writeTo(myMockPrintWriter);
myMockPrintWriter.verify();
}
```

2.4.2.3.1 EasyMock

Το EasyMock είναι μία βιβλιοθήκη που προάγει έναν εύκολο τρόπο χρησιμοποίησης των Mock Objects για συγκεκριμένες κλάσεις και διεπιφάνειες. Το γράψιμο και η συντήρηση των Mock Objects μπορεί να καταλήξει σε μία πολύ σχολαστική και δύσκολη εργασία η οποία μπορεί να οδηγήσει σε λάθη. Το EasyMock δημιουργεί δυναμικά Mock Objects χωρίς να χρειάζεται να γραφτούν καθόλου. Αυτό επιτυγχάνεται μέσα από τη μαγεία των δυναμικών διαμεσολαβητών, η οποία είναι μία πολύ συνήθης τακτική σχεδιασμού στον αντικειμενοστρεφη προγραμματισμό και υλοποιείται διαρκώς στην JAVA ακόμα και από τις κεντρικές λειτουργίες της. Το EasyMock δίνει τη δυνατότητα να δημιουργήσεις τη βασική υλοποίηση μίας διεπιφάνειας μόνο με μια γραμμή κώδικα. Προσθέτοντας επίσης το EasyMock μπορούν επίσης να υλοποιηθούν και προσομοιώσεις κλάσεων. Αυτές οι προσομοιώσεις μπορούν να δαιμορφωθούν για οποιονδήποτε σκοπό, από το να χρησιμοποιηθούν ως παράμετροι στην κλήση μιας μεθόδου ώστε απλά τηρήσει την υπογραφή της αυτή η κλήση μέχρι να χρησιμοποιηθούν για την αξιολόγηση μιας σειράς μεθόδων.

```
import java.io.IOException;

public interface ExchangeRate {

  double getRate(String inputCurrency, String outputCurrency) throws
IOException;

}


import java.io.IOException;

public class Currency {

   private String units;
   private long amount;
   private int cents;
```

```
    public Currency(double amount, String code) {
        this.units = code;
        setAmount(amount);
    }

    private void setAmount(double amount) {
        this.amount = new Double(amount).longValue();
        this.cents = (int) ((amount * 100.0) % 100);
    }

    public Currency toEuros(ExchangeRate converter) {
        if ("EUR".equals(units)) return this;
        else {
            double input = amount + cents/100.0;
            double rate;
            try {
                rate = converter.getRate(units, "EUR");
                double output = input * rate;
                return new Currency(output, "EUR");
            } catch (IOException ex) {
                return null;
            }
        }
    }

    public boolean equals(Object o) {
        …
    }

}


import junit.framework.TestCase;
import org.easymock.EasyMock;
import java.io.IOException;

public class CurrencyTest extends TestCase {

    public void testToEuros() throws IOException {
        Currency testObject = new Currency(2.50, "USD");
        Currency expected = new Currency(3.75, "EUR");
        ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
        EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
        EasyMock.replay(mock);
        Currency actual = testObject.toEuros(mock);
        assertEquals(expected, actual);
    }

}
```

2.4.2.3.2 JMock

Το JMock είναι ακόμα μία βιβλοθήκη που παρέχει βοήθεια στη δημιουργία και εκτέλεση ελέγχων με τη χρήση των Mock Objects. Το JMock βοηθάει στη γρήγορη υλοποίηση τέτοιων προσομοιώσεων ώστε να μην κόβεται ο ρυθμός των προγραμματιστών το οποίο καθιστά έναν πολύ σημαντικό παράγοντα στον οδηγούμενο από ελέγχους προγραμματισμό. Επίσης αφήνει τον ενδιαφερόμενο να συγκεκριμενοποιήσει με ακρίβεια τις αλληλεπιδράσεις μεταξύ των αντικειμένων. Επίσης ακόμα ένα σημαντικό προνόμιο που παρέχει είναι η ολοκλήρωση που έχει με τα γνωστότερα ολοκληρωμένα περιβάλλοντα προγραμματισμού. Είναι πολύ ευέλικτο και μπορεί είτε να επεκταθεί και να οριστεί πάνωω στην υπάρχουσα λειτουργία του νέες ειδικές ρυθμίσεις ή τρόποι λειτουργίας. Τέλος η ευελιξία του έγκειται και στο ότι μπορεί να «κολλήσει» σε πολλά γνωστά πλαίσια ελέγχου (συνήθως χρησιμοποιείται με JUnit).

```
interface Subscriber {
    void receive(String message);
}

import org.jmock.Mockery;
import org.jmock.Expectations;

public class PublisherTest extends TestCase {
    Mockery context = new Mockery();

    public void testOneSubscriberReceivesAMessage() {
        // set up
        final Subscriber subscriber = context.mock(Subscriber.class);

        Publisher publisher = new Publisher();
        publisher.add(subscriber);

        final String message = "message";

        // expectations
        context.checking(new Expectations() {{
            oneOf (subscriber).receive(message);
        }});

        // execute
        publisher.publish(message);

        // verify
        context.assertIsSatisfied();
    }
}
```

2.4.3 Διερευνητικοί και Χρηστικοί Έλεγχοι

Το καλό λογισμικό απαιτεί συνεργσία και επικοινωνία. Οι έλεγχοι αυτού του τεταρτημορίου βασίζονται αλλά παράλληλα βελτιώνουν κιόλας αυτήν την αξία. Στους ελέγχους αυτούς εκτός από τη εύρυθμη λειτουργία σε επίπεδο από χρήστη ως το τέλος (end to end) ελέγχεται και το κατά πόσο συμβαδίζουν οι απαιτήσεις του πελάτη με αυτό που αντιλήφθηκε και υλοποιεί η ομάδα ανάπτυξης. Μπορούμε να το θέσουμε λοιπόν πως στα προηγούμενα 2 τεταρτημόρια ελέγχεται αν ο κώδικας που χτίζεται είναι σωστός, στους ελέγχους αυτού του τεταρτημορίου ελέγχεται αν είναι ο σωστός κώδικας αυτός που χτίζεται.

2.4.3.1 FIT

Το FIT είναι ένα εργαλείο για να ενδυνανμώνει και να προάγει την συνεργασία και την επικοινωνία, 2 πολύ σημαντικούς παράγοντες στις Ευέλικτες Μεθοδολογίες. Ο τρόπος λειτουργίας του είναι ο εξής: με έναν κειμενογράφο ένας πελάτης ή ένα μέλος της ομάδας ελέγχου μπορεί να περιγράψει σε ένα παράδειγμα τη συμπεριφορά που αναμένεται από την εκτέλεση του. Έτσι δε χρειάζεται κάποιος να είναι προγραμματιστής για να γράψει υποθέσεις ελέγχου. Το FIT συγκρίνει αυτόματα αυτά τα παραδείγματα που έχουν δημιουργηθεί έναντι των αποτελεσμάτων που επιστρέφονται από την εκτέλεση του προγράμματος και με αυτόν τον τρόπο λειτουργεί ως συνδετικός κρίκος ανάμεσα στις ομάδες των επιχειρησιακών ευθυνών και της ανάπτυξης Λογισμικού. Τα κύρια πλεονεκτήματα του FIT είναι ότι μέσα από τα παραδείγματα ο πελάτης δίνει πιο καθαρές οδηγίες στην ομάδα υλοποίησης του τι περιμένει, ξεκαθαρίζοντας τις απαιτήσεις του και ότι ο πελάτης έχει μια πιο ξεκάθαρη άποψη από το τι δημιουργείται από την ομάδα ανάπτυξης δίνοντας το ευκαιρία να εντοπιστούν αμέσως και να κοπούν οι οποιεσδήποτε παρεκκλίσεις από τις πραγματικές το υεπιθυμίες.

Το FIT δουλεύει απλά διαβάζοντας πίνακες σε HTML αρχεία οι οποίοι μπορούν να έχουν φταιχτεί ακόμα και με έναν απλό κειμενογράφο. Ο κάθε πίνακας μεταγλωτίζεται από μία εγκατάσταση (fixture) την οποία έχουν φτιάξει οι προγραμματιστές. Η εγκατάσταση αυτή ελέγχει τα παραδείγματα μέσα στους πίνακες τρέχοντάς τα στο πραγματικό πρόγραμμα. Το FIT δίνει τη δυνατότητα στους πελάτες να φτιάχνουν συγκεκριμένα παραδείγματα και να δίνουν στους προγραμματιστές τη διαίσθηση για το προϊόν που χτίζεται. Από την άλλη πλευρά

οι προγραμματιστές δουλεύοντας πάνω στις εγκαταστάσεις και στο λογισμικό αφήνουν τους πελάτες να πειραματιστούν και να γνωρίσουν και αυτοί το τι γίνεται. Μέσα από αυτή τη συνεργασία ολόκληρη η ομάδα μαθαίνει περισσότερα για το προϊόν και παράγει καλύτερα αποτελέσματα.

```
public class WeeklyCompensation : ColumnFixture
{
    public int StandardHours;
    public int HolidayHours;
    public Currency Wage;

    public Currency Pay()
    {
        WeeklyTimesheet timesheet = new WeeklyTimesheet(StandardHours, HolidayHours);
        return timesheet.CalculatePay(Wage);
    }
}
```

2.4.3.2 FitNesse

Το FitNesse είναι και αυτό ένα εργαλείο συνεραγσίας που μπορεί και συνεργάζεται με εργαλεία και πλάισια των άλλων τεταρτημορίων και δη του πρώτου. Πρόκειται στην ουσία για έναν web εξυπηρετητή τύπου wiki που παρέχει στο χρήστη τη δυνατότητα να γράψει-δημιουργήσει απαιτήσεις οι οποίες θα χρησιμοποιηθιούν ως είσοδος στους ελέγχους. Οι έλεγχοι αυτοί μπορούν στη συνέχεια να λειτουργήσουν ως έλεγχοι αποδοχής (acceptance tests) και φυσικά να εκτελούνται αυτόματα. Για την ακρίβεια το FitNesse είναι ένα εργαλείο το οποίο περικλέιει την αρχιτεκτονική του FIT και την κάνει προσβάσιμη στους χρήστες με έναν άλλο τρόπο.

Στο FitNesse οι έλεγχοι όπως άλλωστε και οι αυτοί του FIT οι είσοδοι των ελέγχων έρχονται ως δεδομένα τοποθετημένα μέσα σε πίνακες και αντίτοιχα τα αναμενόμενα αποτελέσματα είναι και αυτά αποθηκευμένα σε πίνακες. Οι έλεγχοι επιζούν σε μία σελίδα ελέγχου δημιουργημένη και διαμορφωμένη σύμφωνα με τη γνωστή στους περισσότερους wiki γλώσσα τυποποιημένων συμβόλων (markup). Επίσης παρέχεται από το πλαίσιο και ένα εργαλείο γραφής για όσους δεν είναι εξοικειωμένοι με το wiki πρότυπο. Στις σελίδες/α ελέγχου αυτές είναι τοποθετημένοι οι πίνακες που περιγράφηκαν πιο πάνω και η εκτέλεση αυτών των ελέγχων μπορεί να γίνει ανα πάσα στιγμή απλά με το πάτημα ενός κουμπιού στη

σελίδα. Από την πλευρά του προγραμματιστή οι απιτήσεις προκειμένου να δημιουργηθούν οι έλεγχοι είναι οι ίδιοι όπως στο FIT δηλαδή να δημιουργηθούν οι εγκαταστάσεις (fixtures) ώστε να συνδεθεί με αυτόν τον τρόπο οι σελίδες με τους ελέγχους και το υπο δοκιμή λογισμικό.

2.4.3.3 Concordion

Ακόμα ένα πλαίσιο εγγραφής αυτοματοποιημένων ελέγχων αποδοχής. Όπως και στα προηγούμενα το ζητούμενο είναι η συνεργασία των δυο πλευρών , των πελατών δηλαδή και των ατόμων που εμπλέκονται στην ανάπτυξη για τη δημιουργία ελέγχων που θα λειτουργούν ως έλεγχοι αποδοχής, που θα πιστοποιούν πιο απλά ότι αυτό που επιτυγχάνουν είναι και αυτό που ζητείται από τον πελάτη. Ο τρόπος λειτουργίας του πλαισίου μοιάζει και σε αυτό το κομμάτι με τα 2 προηγούμενα, χρηιμοποιεί εγκατάστασεις που έχουν χτιστεί από τους προγραμματιστές για να συνδεθούν οι περιπτώσεις ελέγχων με την υλοποίηση του προϊόντος. Η διαφοροποίηση εδώ σε σχέση με τα προαναφερθέντα παραδείγματα είναι αφ’ενός στην υλοποίηση του συγκεκριμένου προϊόντος και αφ’ετέρου στον τρόπο χρησιμοποίησης και των παροχών που προσφέρει στον χρήστη.

Στο Concordion οι έλεγχοι γράφονται σε απλό αρχείο κειμένου και έχουν μια πιο φυσική όψη πιο κοντά στην ανθρώπινη γλώσσα. Μέσα στο κείμενο που χρησιμοποιείται υπάρχουν πράγραφοι, πίνακες και σημεία στίξης χωρίς να είναι απαιτούμενη η δομή των εναλλλακτικών διαδρομών όπως σε άλλα πλάισια. Το κείμενο γράφεται μέσα σε HTML αρχείο κάνοντας τα ικανά να χρησιμοποιηθούν ακόμα και ως επίσημα έγγραφα του προϊόντος και μάλιστα δίνοντας το περιθώριο να χρηιμοποιηθούν μέσα στο σώμα του κειμένου ακόμα και σύνδεσμοι και εικόνες.Στο Concordion μπορούν να τεθούν σε σειρά διάφοροι έλεγχοι και καθ’αυτόν τον τρόπο μία πολύπλοκη περίπτωση να «σπάσει» σε περισσότερες αλλά και απλότερες περίπτωσεις-ελέγχους. Οι έλεγχοι στο συγκεκριμένο πλαίσιο τρέχουν και αυτοί σαν έλεγχοι μονάδας, δηλαδή με τη βοήθεια ενός πλαισίου και όταν πρόκειται για ελέγχους σε JAVA (υποστηρίζει και άλλες γλώσσες προγραμματισμού) σε JUnit. Παρακάτω ακολουθεί το πιο απλό παράδειγμα ελέγχου (Hello World).

```
<html
xmlns:concordion="http://www.concordion.org/2007/concordion">
  <body>
```

```
        <p concordion:assertEquals="getGreeting()">Hello World!</p>
    </body>
</html>

package example;

import org.concordion.integration.junit4.ConcordionRunner;
import org.junit.runner.RunWith;

@RunWith(ConcordionRunner.class)
public class HelloWorldFixture {

    public String getGreeting() {
        return "Hello World!";
    }
}
```

2.4.3.4 Arbiter

Οι προδιαγραφές του συστήματος μπορούν με τη βοήθεια αυτού του πλαισίου αφού γραφτούν σε κανονικό κείμενο και προστεθούν κάποια συντακτικά στοιχεία που αντιλαμβάνεται ο εξυπηρετητής του πλαισίου όπως και κάποια γραφικά αντικείμενα όπως επισημασμένα σημεία του κειμένου και στοιχισμένες κατηγοριοποιήσεις να διαβαστούν από τον Arbiter εξυπηρετητή και να εκτελεστούν οι έλεγχοι. Τα αποτελέσματα των ελέγχων θα φανούν στην ίδια τη σελίδα και έτσι ο πελάτης μπορεί να παρακολουθεί την πορεία της ανάπτυξης. Το Arbiter είναι και αυτό ένα εργαλείο συλλογής και πιστοποίησης απαιτήσεων, δικτυακών προϊόντων . Στόχος του είναι να οξύνει και αυτό την επικοιινωνία μεταξύ πελάτη και προγραμματιστή.

Από πλευράς λειτουργίας πρόκειται για μία εφαρμογή αποθήκευσης απλών κειμένων σε web εξυπηρετητή. Όταν προστίθονται ή μεταβάλλονται κείμενα απαιτήσεων τότε τρέχουν οι έλεγχοι. Οι έλεγχοι είναι πραδείγματα πως θα περίμενε κάποιος χρησιμοποιώντας τον περιηγητή του να λάβει πληροφορία από τον υποτιθέμενο χώρο που ελέγχεται. Γενικά ότνα βρεθεί κείμενο το οποίο είναι έντονο αυτό θεωρείται μέρος του συντακτικού που πρέπει να μεταγλωτιστεί από το πλαιίσιο.

Show a share price

Any member of the public can use our free search utility to look up a single share price.

1. Go to home page
2. Click on find shares
3. Set search to IBM
4. Click search
5. Should see "IBM share price"

This adds value to new vistors to the site and will be used by marketing to promote paid for features of the service.

```
class PublicSharePriceUseCase extends ArbiterTestCase {
   function testShowASharePrice() {
      $this->goToHome();
      $this->click('find shares');
      $this->setField('search', 'IBM');
      $this->click('Search');
      $this->assertText('IBM share price');
   }
}
```

2.4.4 Έλεγχοι Επιδόσεων και Ασφάλειας

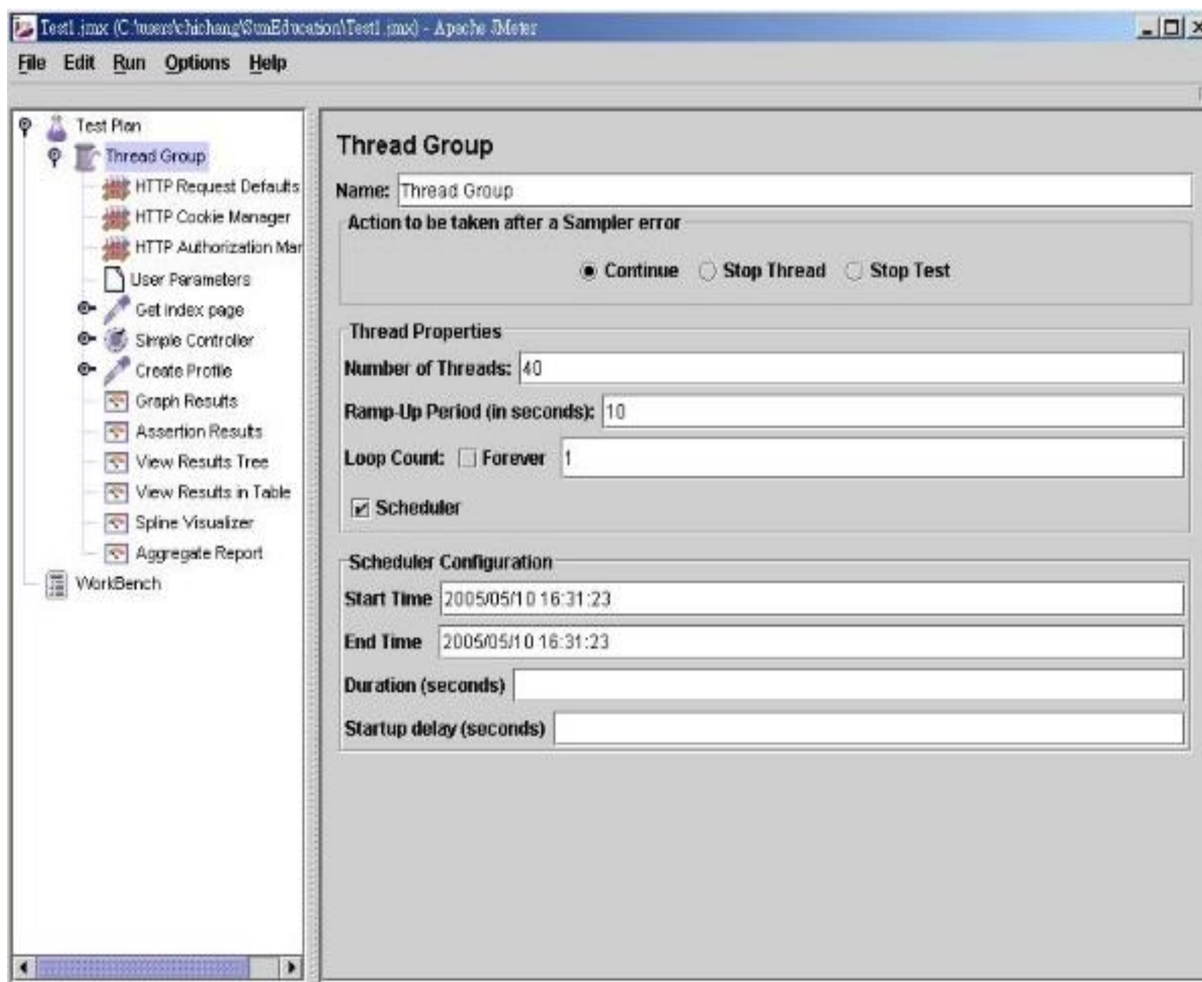
Οι έλεγχοι αυτοί συμπεριλαμβάνουν την αξιολόγηση πεδίων τα οποία ανήκουν στις λεγόμενες μη λειτουργικές απαιτήσεις. Αυτές συνήθως υποτιμόνται από άποψης κόστους και μόχθου προκειμένου να επιτευχθούν και εέτσι πολλές φορές ολόκληρα προϊόντα έχουν ακυρωθεί λό΄γω μη τηρήσεως τέτοιου είδους απαιτήσεων. Ο σημαντικότερος παράγοντας αποτυχίας επίτευξης αυτών των στόχων είναι ότι αφήνονται στο τέλος να ελεγχθούν και να τροποποιηθεί ο κώδικας ώστε να φτάσει στο σημείο ικανοποίσης τους το προϊόν, κάτι το οποίο βέβαια πρέπει να τονίσουμε ότι στην περίπτωση των Ευέλικτων Μεθοδολογιών αποφεύγεται λόγω του κύκλου ανάπτυξης που αυτές καθορίζουν. Παρακάτω παρατίθονται κάποια πολύ γνωστά εργαλεία τα οποία τονίζεται ότι πρέπει να χρησιμοποιούνται κατά τη διάρκεια της ανάπτυξης.

2.4.4.1 Apache JMeter

Το Apache JMeter είναι μια εφαρμογή γραφείου που χρησιμεέι για να φορτώνει λειτουργική συμπεριφορά και να μετράει τις επιδόσεις μιας υπο έλεγχο εφαρμογής. Μάλιστα αρχικά το JMeter είχε σχεδιαστεί για τον έλεγχο δικτυακών εφαρμογών, τώρα πια όμως δεν περιορίζεται μόνο σε αυτού του είδους τις λειτουργίες αλλά η γκάμα έχει εξαπλωθεί και σε άλλες. Το JMeter μπορεί να μετρήσει τις επιδόσεις τόσο στατικών όσο και δυναμιλών πόρων όπως αρχείων, αντικειμένων JAVA, βάσεις δεδομένων, εξυπηρετητές μεταφοράς αρχείων και φυσικά Servlets. Ο τρόπος χρησιμοποιείται κοινώς είναι να προσομοιώνει βαριές λειτουργίες, πολλαπλά αιτήματα σε έναν εξυπηρετητή, ένα δίκτυο ή ακόμα και ένα JAVA αντικείμενο και να αναλύει τις επιδόσεις του κάτω από διαφορετικού τύπου φορτώσεις κάθε φορά. Μια από αυτές τις προσομοιώσεις υψηλής πίεσης που μπορεί να δημιουργήσει κανείς με αυτήν την εφαρμογή είναι και η ταυτόχρονη διάθεση κάποιων πόρων του συστήματος (concurrency) και ο έλεγχος της σωστής λειτουργίας του. Τέλος, μπορεί να δημιουργήσει ακόμα και γραφικές παραστάσεις αυτών των επιδόσεων.

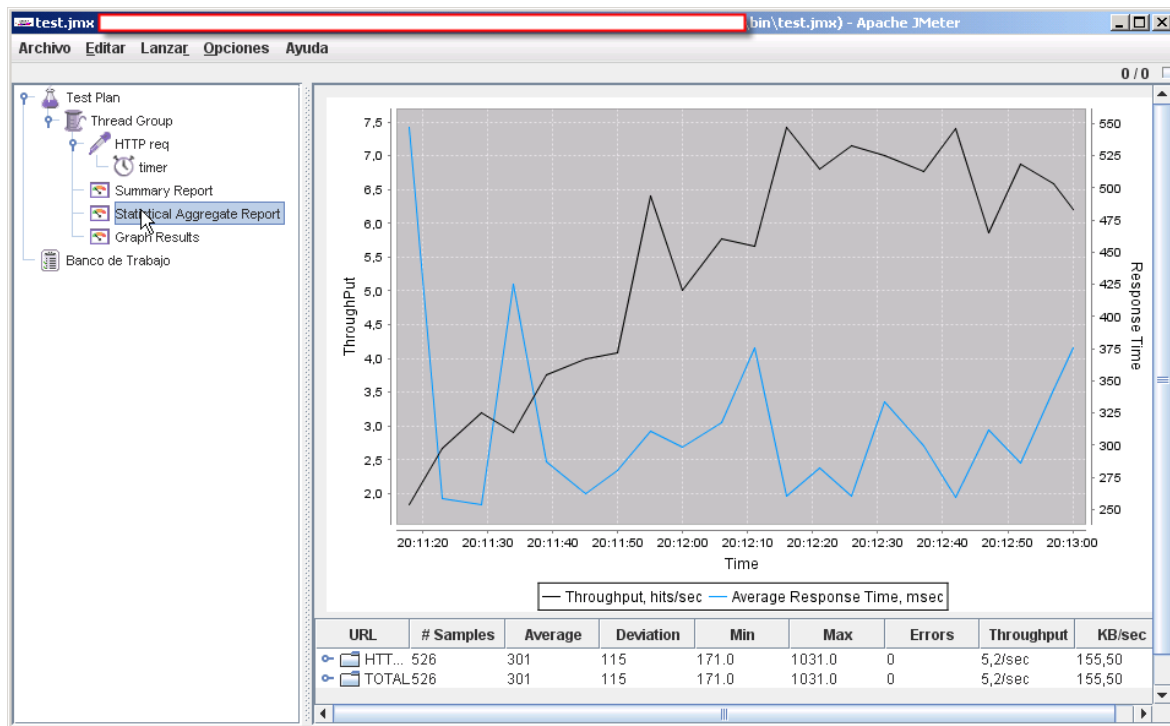
Οι τύποι των προτοκόλων που υποστηρίζει και μέσα πάνω στα οποία μπορεί να δημιουγήσει πολλαπλές αιτήσεις περιλαμβάνουν τα HTTP, HTTPS, POP3, IMAP, SMTP και SMTPS. Μπορεί ακόμα να ελέγξει δικτυακές υπηρεσίες πάνω από το προτόκολο SOAP αλλά και άλλα ενδιάμεσα συστήματα (middleware) όπως JMS. Επίσης μέσω της σύνδεσης με βάσεις δεδομένων που προσφέρει το JDBC είναι ικανό για τη μέτρηση και αυτού του είδους τις λειτουργίες αλλά εκτός από σχεσιακές βάσεις μπορεί να ελέγξει και συστήματα τύπου LDAP.

Ο τρόπος που επιτυγχάνει όλους αυτούς τους ελέγχους, δηλαδή την ταυτόχρονη δημιουργία και εκτέλεση πολλών διαδικασιών είναι με τη χρησιμοποίηση νημάτων. Μέσα από την οργάνωση αυτών των νημάτων και την ομαδοποίησή τους μπορεί κάποιος με τη βοήθεια και ταυτόχρονη χρήση χρονομετρικών υπηρεσιών που προσφέρονται, ένα πλήθος επιλογών εγκατάστασης, και λογικών ελεγκτών, δηλώσεων αξιολόγησης (assertions) μπορεί να δημιουργηθεί ένα ολόκληρο πλάνο ελέγχου. Όλα τα παραπάνω στοιχεία μπορούν να φτιαχτούν μέσα από τη διεπιφάνεια που προσφέρει το JMeter και που φαίνεται στην εικόνα που ακολουθεί.



Εικόνα 10 Διεπιφάνεια JMeter

Μία ομάδα νημάτων χρησιμοποιείται για να οριστεί το πόσα νήματα και κατ'επέκταση πόσοι έλεγχοι θα εκτελεστούν, σε πόση ώρα θα γίνουν αυτοί οι έλεγχο και πόσες φορές θα εκτελεστούν. Το JMeter περιλαμβάνει 2 διαφορετικούς ελεγκτές, τα δείγματα και τους λογικούς ελεγκτές. Με τα δείγματα ορίζεται ο τύπος αίτησης που αποστέλλεται στον εξυοηρετητή και ο τύπος απάντησης φυσικά που αναμένεται. Με τη βοήθεια των λογικών ελεγκτών δημιουργούνται κάποια σενάρια δηλαδή μια ροή που περιλαμβάνει σε ποια περίπτωση και υπο ποιες συνθήκες καθώς και με ποιον τρόπο θα τρέξει ένας έλεγχος. Οι ακροατές (listeners) παρακολουθουν τα στοιχεία που συγκεντρώνει η εφαρμογή κατά την εκτέλεση των ελέγχων και μπορούν να χρησιμοποιηθούν για τη δημιουργία γραφικών παραστάσεων όπως δείχνει και η εικόνα 11. Οι χρονομετρητές χρησιμοποιούνται



Εικόνα 11 JMeter Γράφημα

για να μπορεί να ελεγχθεί η συχνότητα των ελέγχων και κυρίως για να μπορεί να προστεθεί κάποια καθυστέρηση μετξύ των αποστολών δειγμάτων για την καλύτερη λειτουργία αλλά και ως παραμετροποίηση των ελέγχων. Οι δηλώσεις αξιολόγησης είναι αυτές σύμφωνα με τις οποίες κρίνεται η επιτυχία ενός αποτελέσματος.

2.4.4.2 DBMonster

Το DBMonster είναι ένα εργαλείο που διευκολύνει τους προγραμματιστές εφαρμογών βάσεων δεδομένων να βελτιστοποιήσουν τη δομή της βάσης, τη χρήση των δεικτών και φυσικά να ελέγξουν την εξέλιξη της κάτω από αυτές τις δοκιμές εξέλιξης υπο μεγάλο φόρτο. Το DBMonster δημιουργεί τυχαίο περιεχόμενο και μπορεί να εμφυτευτεί αυτό σε μια βάση. Μπορεί να δημιουργήσει αυτόματους ελέγχους και να πιέσει τη βάση με πολλαπλά ερωτήματα που τρέχουν μέσω της έναρξης πολλών νημάτων. Είναι απλό στην παραμετροποίηση του μέσα από ένα αρχείο το οποίο λέγεται dbmonster.properties και δέχεται βασικές επιλογές όπως η διεύθυνση της βάσης και τα στοιχεία αυθεντικοποιήσης.

2.5 Επίλογος

Η ποιότητα λογσμικού μπορεί να μην είναι κάτι το τελείως ξεκάθαρο ως προς τον ορισμό της αλλά σίγουρα είναι κάτι το οποίο επιδιώκεται από κάθε ομάδα ανάπτυξης λογισμικού για δύο λόγους. Πρώτον για να ικανοποιήσουν τις ανάγκες του πελάτη και δεύτερον για να μπορούν να ανταποκριθεί η ομάδα σε μελλοντικές απαιτήσεις βοηθούμενη από τον κώδικα. Στην ποιότητα λογισμικού που παράγεται σημαντικό ρόλο παίζουν οι έλεγχοι. Μάλιστα οι έλεγχοι στις Ευέλικτες Μεθοδολογίες έχουν το δικό τους μερίδιο στην ανάπτυξη και δεν θεωρούνται κάτι ξεχωριστό ή και «πολυτέλεια». Οι έλεγχοι βοηθούν στην ανάπτυξη στην εύρεση λαθών αλλά και στην αντιμετώπισή τους. Για αυτό τον λόγο οι έλεγχοι απαρτίζουν ολόκληρες ομάδες στις Ευέλικτες Μεθοδολογίες με ξεκάθαρο ρόλο. Οι ομάδες Ευέλικτου Ελέγχου πρέπει να γνωρίζουν σε βάθος εκτός από τον χειρισμό και την παραμετροποίηση των εργαλείων που χρησιμοποιούν κατά κόρον και από προγραμματισμό ώστε να μπορούν να αξιοποιήσουν στο έπακρο τη βοήθεια που τους παρέχεται από τα εργαλεία ελέγχου τα οποία και είναι πια απαραίτητο κομμάτι στη βιμηχανία της ανάπτυξης λογισμικού.

Αναφορές

1.http://www.agilemodeling.com
2. http://www.versionone.com/
3. http://en.wikipedia.org
4. http://testng.org/doc/index.html
5. https://github.com/junit-team/junit
6. http://docs.seleniumhq.org/
7. http://httpunit.sourceforge.net/
8. http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/testing.html
9. http://www.opensourcetesting.org/performance.php
10. http://fitnesse.org/
11. http://www.concordion.org/
12. http://fit.c2.com/wiki.cgi?FitDocumentation
13. http://www.easymock.org/EasyMock3_1_Documentation.html
14. http://jmock.org/getting-started.html
15. http://www.ibm.com/developerworks/java/library/j-easymock/index.html
16. http://scaledagileframework.com/nonfunctional-requirements/

Βιβλιογραφία

1. P. Abrahamsson, J. Warsta, M.T. Siponen, J. Ronkainen, *New directions on agile methods: a comparative analysis*, in: Proceedings of the 25th International Conference on Software Engineering (ICSE'03), IEEE Press, 2003.
2. Beck, K., et al., *The Agile Manifesto.* 2001: p. http://www.agileAlliance.org.
3. Mikael Lindvall, Vic Basili, Barry Boehm3, Patricia Costa, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero, Laurie Williams, and Marvin Zelkowitz. *Empirical Findings in Agile Methods*.
4 Tim Mackinnon, Steve Freeman, Philip Craig, *Endo-Testing: Unit Testing with Mock Objects.*
5. P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, *Agile software development methods: review and analysis*, VTT Technical report, 2002.
6. Kirsi Corhonen . *Evaluating the Impact of the Agile Transformation – a longitudinal Case Study in distributed context*.
7. K. El-Emam, "Finding Success in Small Software Projects," *Agile Project Management*, vol. 4, 2003.
8. Lucas Layman, Laurie Williams, Lynn Cunningham. *Exploring Extreme Programming in Context: An Industrial Case Study*, Agile Development Conference, 2004.
9. C. Jones, *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley, 2000.
10. Olle T.W, Sol H.G, Tully C.J. *Information system design methodologies: A feature analysis*, 1983.
11. Jim Highsmith, *What Is Agile Software Development? An agile case story*.
12. David Cohen, Mikael Lindvall, Patricia Costa, *An Introduction to Agile Methods.*
13. Jussi Auvinen, Rasmus Back, Jeanette Heidenberg, Piia Hirkman, Luka Milovanov, *Improving the Engineering Process Area at Ericsson with Agile Practices.A case Study*.
14. Tore Dyba, Torgeir Dingsoyr*, Empirical studies of agile software development: A systematic review*, 2008.
15. S. Ilieva, P. Ivanov, E. Stefanova, *Analyses of an agile methodology implementation,* in: Proceedings 30th Euromicro Conference, IEEE Computer Society Press, 2004

16. Aldo Dagnino, Karen Smiley, Hema Srikanth, Annie I. Antón, Laurie Williams, *Experiences in applying agile software development practices in new product development*.

17. Andy Hunt, SteveThomas, *Software Construction: Mock Objects , 2002.*

18. Cockburn A., *Agile Software Development*, 2002

19. F. Macias, M. Holcombe, M. Gheorghe, *A formal experiment comparing extreme programming with traditional software construction*, in: Proceedings of the Fourth Mexican International Conference on Computer Science (ENC 2003), 2003.

20. Scott Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process,* 2002.

21. Paul R. Allen, Joseph J. Bambara, *Sun Certification for the Enterprise Architect,* 2007.

Παράρτημα A

Importance of Agile testing. Comparing 2 Agile Methodologies' code quality: An empirical study.

Abstract. Over a decade now, agile methodologies have been used in software development. Many arguments have been set about their effectiveness and their way they are or can be used. This comes to competent the question "How can Agile replace old fashioned methodologies ?" .This paper reports result of a comparison between 2 organizations, the first one using an agile method, Crystal Orange from Crystal Family methodologies and the second an agile as well and specifically extreme programming. Its purpose is to depict mostly the necessity of testing during the development and pre-release phases, how important is testing for these methodologies and how defects' rate and code quality can be affected. The comparative analysis is performed using defect data reports and defect data metrics. Finally, some suggestions are proposed based on the results and the open discussion with the participants of these projects.

Keywords: defect management, agile software, agile testing, software quality.

1. Introduction

Over the last years when the agile methods have gained popularity, hundreds of cases [11, 13, 14] have been published which extol their efficiency and their ability to concentrate on humans' cooperation in order to achieve success [2]. Agile methodology of software development has been emerged due to the need of having a flexible development lifecycle which can adapt to requirements' changes. This is a situation is faced more and more frequently, in almost every type of software projects regardless its criticality or its size, as the IT technologies and the needs of public change/evolve rapidly.

A plenty of methods are considered agile and the most known of them are extreme-programming (XP), SCRUM, Crystal Family, Feature Driven Development (FDD), Agile Modeling (AM), Adaptive Software Development (ASD), Dynamic Systems Development Methods (DSDM), Pragmatic Programming (PP), Internet-speed Development (ISD), Lean Development, the Rational Unified Model (RUP) [5, 11]. The most commonly used are extreme-programming and SCRUM, however, those are not the only ones as many of agile methodologies are continuously generated, simply by using a mixture of practices of them, as hybrids [15, 16]. This is also one of the main reasons which have turned Agile so famous beneath the software development industry, the freedom that gives in order to deliver functional software [2, 7].So, after that, it is easily concluded that no rule indicates when and which agile methodology must be used, but after the experience of their utilization some guide lines have been arranged which in general take into account the size of the project, the experience and abilities of the team's members, the criticality, security and reliability of the system that must be accomplished [3].

On the other side there are the traditional and stricter methods which they are simply referred also as document-driven (in literature they are also found as process-oriented or plan-driven) and mainly consist of four steps: Analysis, Design, Implementation and Testing. Some well known of these document driven methods are Waterfall, Spiral Model, Capability Maturity Model(CMM) .These are based upon the first phases of development lifecycle and more specifically the requirements analysis and systems' architecture and design. As these are the critical phases of those methodologies the more effort and time are consumed during them which mean that implementation and testing are treated as procedural. The main disadvantage of these methods is that if something goes

wrong, or a feature is not predicted or even worse the customer wants to add/modify requirements then the whole project's implementation is under danger.

Despite the differences they have, agile methodologies have some common principles and values. In order to achieve their goals agile methodologies utilize some practices/techniques, some of them have been also used with document driven methodologies. This is how we can categorize an organization regarding the development methodology they use. If the organization uses during every phase of the development cycle agile practices and this is called fully agile. In the case that was aforementioned, where an organization uses document driven methodologies but utilizes also some of the agile practices then this organization is called intermediate and of course there are the fully document driven organizations which are also called basic [6,7].

Some of these practices are really common in Software Development Industry and it's almost impossible someone in this have not ever used at least one of them. Indicatively some of the practices which are used with XP are: planning games, small/short releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour week, on-site customer, coding standards, open workspace, just rules. In SCRUM some of the practices are: product backlog, effort estimation, sprint, sprint planning meeting, sprint backlog, daily scrum meeting and sprint review meeting [5, 12].

After that, someone could decipher that almost none organization is basic in true and this is because some of the agile practices emerge without conscious adoption and are utilized by the developers just because they are simply found useful [17]. On the other hand, referring to an organization as fully agile could also be imprecise as there is no official or even clear agreement of how to distinguish the two approaches (agile-traditional) [5], but the concession that has been mentioned above.

This paper proceeds as follows: Section 2 presents the background of the organizations, describes the agile practices used by them, as part of the methodologies they followed and the conditions under the results have been fetched. In section 3 a brief description of the metrics that have been used and how these results have been concentrated is portrayed. Section 3 cites the results of the 2 projects' code comparison. We discuss open issues in Section 4 and present our conclusions in Section 5.

2. Background

This empirical study compares code quality of two organizations and this chapter focuses on the description of these which constitute the domain of the study. In order to have a safer conclusion from the reports, giving a detailed delineation of these organizations is important. The first participant organization is a multinational company and its project in general was an integrated system which allowed automated transmission of information between countries of EU and their insurance organizations. The second is medium sized organization which is expertise in software services for Telecommunications Providers. Below some context factors which are defined by C. Jones [9] are used in order to achieve a detailed description of the organizations context and for the sake of brevity we will refer to them as Insurance Project for the former and Telco Product for the latter.

According to C. Jones' classification the first organization's software can be sorted as outsourced since the requirements were dictated by a contract with EU. The second participant software of this study can be marked as commercial, since there was no initial requirement from any customer and it turned to a licensed product for Telco Industry.

More specifically the first project was a pretty complicated system which had been fragmented in 10 different parts-modules and the plan was to be revealed in 6 phases each one of them having duration from 2 to 3 months. Every module was completed in various phases, spreading from 2 to 4 of them. The development team was consisted of 12 developers. In project were also fully employed one architect, two project managers and two development team leaders, a testing team made of one testing team leader and 4 testers and at the end one technical writer. The concept was that these 12 developers would be divided into three or four teams, depending on the phase and the modules' size and number, which everyone should have a single module to release. The second product is acting as mediation between Service Providers backend systems and first level resources by implementing some telecommunication protocols and also provides clustering, logging, monitoring and scheduling features. The product is still under maintenance and of course development since customers' new requirements always arises. Results of study and context description too is given for the last calendar year and is a representative sample for at least the last 5 years of organization's operational as no big differentiations have been occurred. There are 10 developers, one architect, a technical writer and 2 project managers. During the development both projects had departures of developers and from both its

architect left too. These incidents can prove that personnel were working under pressure and strict deadlines. Table 1 shows the sociological factors for both of the compared parts.

Table 1 Socialogical factors

Context factor	**Insurance Project**	**Telco Product**
Team size	22	14
Highest Degree Declined	Bachelors:10 Masters:12	Bachelors:10 Masters:5
Experience Level of Team	6-10 years:8 <5 years:14	6-10 years:4 <5 years:10
Domain Expertise	Medium	Low
Language Expertise	Medium	Medium
Experience of Project Managers	Medium	Medium
Specialist Available	System architect, DB expert	System architect, DB expert
Personnel Turnover	36%	43%
Morale factors	Architect change, 3 developers change	Architect change,2 developers change

The Insurance Project was mainly implemented with JAVA and actually it was deployed partially on JBoss and Weblogic application servers, so JEE platform and its provisioned libraries have been used. Some of the technologies used were EJB, Hibernate ORM, Web Services (CXF), ESB (Camel), JMS, JMX, JSF. Also a user interface has been implemented with PHP language. As database Oracle DB had been selected for back-end storage. Telco Product is totally implemented to JBoss server and it also takes advantage of JEE technology like Web Services, JMS, JMX, EJB and its libraries and it also uses MySQL as RDBMS. Technological factors are depicted in Table 2. As a developmental tool Eclipse IDE was used by all developers of both teams.

Table 2 Technological factors

Context factor	**Insurance Project**	**Telco Product**
Software Development Methodology	Crystal Orange	Waterfall with some XP practices
Project Management		
Defect Prevention and Removal Practices	Code Reviews, unit testing, integration testing, pair programming	Ad hoc testing by developers and customer
Languages	JAVA, PHP	JAVA
Reusable Materials	Third party libraries, unit test suites	Third party libraries

The first organization had been using for several years agile methodology and specifically SCRUM. Almost every member of the team had an experience of agile practices consequently. Due to the criticality and the size of the project it was decided to be used a Crystal method from Crystal Family methodologies, which was Crystal Orange. The Crystal methodologies do not limit any development, tools or working products while also allowing the adoption of practices from other methodologies, for example, XP and SCRUM practices [18]. Second organization followed the Extreme Programming methodology and utilized also XP Practices. The reason which such a methodology was preferred was the nature of requirements, which were not clear and the potential of change request was in every phase of the project pretty high and customer's demands for often small releases with minimal documentation as well. Finally both organizations appreciated the fact that the methodologies they used gave project management feasibility in contrast with XP and other agile hybrids which do not [1].

Below follows a list with a brief explanation each of of the practices that have been used from the teams and how utilization has been accomplished. Table 3 depicts in summary this list.

Incremental delivery: This is an agile practice mostly featured by XP methodology. It's mainly a strategy which indicates splitting the system in smaller pieces and developing its one of them in separate rate or times. Finally, integration of the completed pieces into a larger functional system takes part. This practice had been followed from the start by the Insurance Project's team by separating the final product in 10 different modules/subsystems and the Telco Product's team bu modularizing its product according the concern it was satyisfying its piece of code.

Continuous integration: It refers to performing all tests, previous and new ones, against the code that has been added. This is done usually in large scale and for the Insurance Project the execution of these tests was testing team's responsibility. Moreover some tools are usually used in order to achieve statistics extraction like performance measurements, stress testing to simulate production conditions and security reassurance. Tools like Ant, SOAP UI, Apache JMeter, Oracle Application Testing Suite, and Nessus Network Vulnerability Scanner.

Measure progress by working code: There are many evaluation techniques which try to calculate the progress of a project but which finally counts is how the code you deliver is getting more functional and satisfies client's requirements. This is what in general both companies adopt as a first priority.

Interactive planning: Another agile practice is interactive planning which can be translated into continuous association between project managers and customers so as to end up with specifications agreement. This requires technical knowledge by project manager or even presence of developers/team leaders in meetings. Only the first team used this practice and in some cases developers had to travel to be present in pre-development meetings.

Developers estimate task efforts: This practice's name is pretty descriptive and is applied by both organizations against their projects, mainly by team leaders which had also discussed with developers for effort estimation about smaller/partitioned pieces of work.

Use cases: Use cases are a very common artifact not only within agile methodologies but also in document driven. Use cases are an abstraction of how people will work with your system [20]. Those are used by developers and designers to be more specific about customers' requirements and testers to generate the test cases as well. Use cases were utilized by the first corporation for designing, developmental and testing purposes.

Design patterns: Design patterns, or patterns, are solutions to recurring problems in a given context amid competing concerns such as performance, security, reusability, scalability and many more [21]. Except their functional pros they have also some educational and communicational advantages as they can be used in a sort of standards during a meeting. Teams in both cases used on purpose many of them such as the Factory Method, Singleton, Façade, Adapter, Observer and others. Of course JEE's platform technology artifacts apply persistently many of design patterns (EJB, JMS) and that results to indirect usage as well.

Continuously developed architecture: This refers to the ability of project's architecture and organization's personnel being adapted to new requirements or constraints by improving in large scale system's design. First organization had to change/improve its architecture of some subsystems as many times security constraints, network connectivity issues and customer's insistence for automation enforced such architecture alterations.

Pair programming: It is one of the most known XP practices in which 2 developers work together at one workstation. One of them writes code and the other observes. The two participants switch roles periodically. The first organization in contrast with the second used in a great degree this tactic.

Collective code ownership: This practice means that no one owns the code exclusively and anyone can modify it if this is necessary. There are also many

tools which help to achieve collective code ownership. Both organizations used this practice, mainly by using an SVN (subversion) server.

Coding standards: This is not exactly an agile practice but more a recommendation which is relative to the programming language and it had to do with some conventions that are used in code. Such conventions could be naming conventions, white spaces and indentation, comments, declarations, methods' size and others. The first organization unlike the second one was forced to adhere to a set of coding conventions and particularly it had also to deliver a report of those conventions that were not followed. Developers discover the conventions impingement and exported the release report with Eclipse check style plug-in tool eclipse-cs.

Refactoring: It is the code restructuring of a piece of code in order to be more readable, scalable, maintainable and less complicated. Developers of the Insurance Project struggled for the aforementioned virtues and every time tried to redesign their code so they are prepared for any additional requirement.

Write tests first: This practice specifies that before a piece of code being included in the project it has to be implemented in unit testing form and so be validated for its functionality. For the first organization, developers followed this practice partially in a relatevily good grade and many pieces of pre-integrated code had been implemented firstly in JUnit classes. These classes, as it is explained below, had been used for automated unit testing as well.

Automated unit testing: Many times implications of new code addition can be undetermined and automated unit testing can save much developer's time from debugging. Usually, automated testing is organized per classes and packages that are relative to which piece of code or module they are testing. Whenever a developer integrates a piece of code to the already existed functional part he/she also adds if available new unit tests to the existed packages. After that the new extended package of unit tests can run against the new extended code. This strategy was followed by the first organization from any single developer. Some tools which helped were: TestNG and JUnit for the creation of these unit tests and input of their mock data. Ant for the package test execution by testing team (black box testing) and report extraction.

Customer writes acceptance tests: This practice also used by the first organization, states that some tests is written by the customer in simple text format like use cases. Their success/pass is defined as prerequisite before a new release delivered to the customer.

Frequent team meetings: This practice obviously comes from SCRUM methodology which determines several types of scheduled meetings (daily, sprint). Within first organization's daily duties were the presence at a short daily (morning) meeting in which new day's goals and previous' issues were discussed among developers, team leaders and sometimes even architect.

Small releases: Another practice used by the first organization and indicates small/often releases like patch improvements, apart from the official delivery dates. The code enriched with new code or debugged every 2-3 working days for the first organization. Of course, within these releases no additional documents were delivered but the code.

Simple design: A practice that encourages the developers/designers always to use the simplest way to implement a piece of code. This has as a result the facilitation of code's maintenance and extension. Only the first organizations used this practice.

Team develops its processes: Processes of first organization team was not changing during the project's development. That might be because of the experience of the team with agile development. In opposite the second organization changed/improved its processes. A reason could be the differentiation of difficulty level between iterations.

No continuous overtime: XP methodology advices for 40 working hours per week per person. This is somewhat impossible in IT Industry but the avoidance of continuous overtimes is something which also keeps productivity of stuff at high rates and does not raises cost of code. That principle followed by none of the two organizations.

Table 3 Practices used

Practice	Insurance Project	Telco Product
Incremental delivery	✓	✓
Continuous integration	✓	
Measure progress by working code	✓	✓
Interactive planning	✓	
Developers estimate task efforts	✓	✓
Use cases	✓	
Design patterns	✓	✓
Continuously developed architecture	✓	
Pair programming	✓	
Collective code ownership	✓	✓
Coding Standard	✓	

Refactoring	✓	
Write tests first	✓	
Automated unit testing	✓	
Customer writes acceptance tests	✓	
Frequent team meetings	✓	
Small releases	✓	
Simple design	✓	
Team develops its processes		✓
No continuous overtime		

Crystal Orange methodology dictates some rules [5, 18] which were met by the team of Insurance Project. In order to achieve better communication the whole team was located in 2 different rooms and actually they were divided according to their mission. Something analogous is followed by the team of Telco Product. Many times customers' different locations and time zones created problems because of the deterioration or lack of communication. Unfortunately both projects had to face such problems due to geographical reasons. From the side of EU, which was the customer of first organization, an action to deal with it was the periodical visits of some key persons to Development Center for an interval of 2 weeks. The geographical factors' data can be found in table 4.

Table 4 Geographical factors

Context factor	**Insurance Project**	**Telco Product**
Team Location	Co-located	Co-located
Number of customers	1	8
Customer location	Remote, multinational, several time zones	Remote, multinational, several time zones

3. Results

In this survey we concentrate on the development team's productivity and the quality of final software delivered to the client. There are plenty of statistics that can or already have been used for the metrics of productivity and quality. For purpose of this paper productivity is measured by the quantity of code per man day. The measurements have been obtained by the development teams' reports. Specifically the first organization needed these metrics as they were given to the client as well. Both development teams used the same tool and this was EMMA plug-in for Eclipse IDE. EMMA plug-in is a tool focused mainly on test metrics and

despite the fact that the second organization didn't use it for such purposes development team utilized its output for project management reports. In tables 5 and 6, size of each product are shown respectively. First organization started its implementation from scratch but second's measurements are fetched for 5 additional deliveries.

Table 5 Insurance Project Size

Iteration Length	Classes	Methods	Blocks	Lines of additional code(sum of code lines)
40 w/d	144	1673	28423	8225
38 w/d	236	2553	48773	8853 (17078)
20 w/d	236	2560	49472	748 (17826)
40 w/d	352	3934	91291	11212 (29038)
35 w/d	538	5530	104579	4004 (33042)
60 w/d	696	7160	142782	9118 (42164)

Table 6 Telco Product Size

Iteration Length	Classes	Methods	Blocks	Lines of additional code(sum of code lines)
30 w/d	42	253	14689	4890
60 w/d	293	676	31173	8259 (13149)
60 w/d	418	1143	52562	7546 (20695)
60 w/d	525	1837	71432	8092 (28787)
60 w/d	565	2086	88001	8086 (36873)

From the above we can export the average productivity per developer per day. This is not a totally representative index of productivity but it is a commonly used and easily measured and it does not include other complicated, uncountable or subjective parameters. The division size by the number of working days and

then by the number of developers gives us the average productivity pace. For first project and sorted by the iterations date the results are 17.13 lines/ day , 19.41 lines/day, 3.11 lines /day, 23.35 lines /day, 9.53 lines /day and finally 12.66 lines /day. The average of all iterations is 15.08. For the second organization the relative numbers are: 16.3 lines /day, 13.7 lines /day, 12.5 lines /day, 13,5lines/day and 13,5lines/day. The overall average is 13.79 lines /day.

Table 7 Productivity per Delivery Period

Product	**1st delivery**	**2nd delivery**	**3rd delivery**	**4th delivery**	**5th delivery**	**6th delivery**	**Overall**
Insurance Project	17.13 lines/ day	19.41 lines/day	3.11 lines/day	23.35 lines/day	9.53 lines/day	12.66 lines /day	15.08 lines/day
Telco Product	16.3 lines/day	13.7 lines/day	12.5 lines/day	13,5 lines/day	13,5 lines/day		13.65 lines /day

It is obvious that the 2 organizations have similar production pace but the first one's is almost 9% faster. Another factor that can be exported from the above data is the curve of the productivity pace which is based on the code's production differencies that appeal among the deliveries for each of the product. The smoother is the productivity's curve the better it is for an organization as this generally indicates the normality of organization's functionality and a normal-steady pace helps to the predictability of product's results in next phases-deliveries. The big gap for the first product appeals during the 3rd delivery and it cannot be a coincidence that architect's change incurred for the 1st organization that period. So, we can conclude that the 1st organization and specifically the methodology it was following were vulnerable to structure's changes and particularly to replacement of some key persons.

Next we represent the main issue of this work which is connected to QA (Quality Assurance) and focus mainly to defect management as this is a metric that can be used for this topic. The first organization used the "Write tests first" practice and a report of their gradeness was over the code was given to the customer. Those data are also represented below, but for the second organization no such practice had been followed and thus there are no data. Defect data for

both teams were collected by their tracking systems and e-mail for the 2nd organization's pre-released defeats as its tracking mechanism is used only for released code and accessed only by customers.

Table 8 Unit test code coverage and results

Releases	Class	Method	Block	Line	Tests	Executed	Failures	Errors
Rel. 1	61.80% (89/144)	41.66% (697/1673)	48.57% (13806/28423)	48.51% (3990.5/8225)	130	130	0	0
Rel. 2	50%(118/236)	37.83% (966/2553)	38.56% (18807/48773)	39.55% (6754.5/17078)	213	213	10	0
Rel. 3	50%(118/236)	37.77% (967/2560)	38.61% (19105/49472)	38.92% (6938.5/17826)	216	216	0	0
Rel. 4	53.69% (189/352)	45.17% (1777/3934)	49.73% (45402/91291)	48.22% (14004.3/29038)	347	346	5	1
Rel. 5	61.52% (331/538)	44.97% (2487/5530)	49.15% (51408/104579)	47.99% (15859.5/33042)	867	867	6	8
Rel. 6	63.64% (443/696)	51.91% (3717/7160)	55.23% (78869/142782)	53.60% (22603.4/42164)	1748	1748	0	0

On the above table it is shown the tested code coverage that has been accomplished through the unit tests. There was a relative stable coverage among the phases. Unfortunatelly, a similar table can not be shown for the second organization since unit testing was not a "must" for developers and TDD it was not a practice that had been followed. However, we can surely assume, after discussions with the 2nd organization's developers, that this rate was much smaller than this of the first organization.

Next, we are going to specify the code quality from the perspective of defect management. There are many ways by which you can evaluate a defect and arrange it in a specific category [6]. Both organizations created and executed end-to-end test cases which have been executed during system integration testing. First organization had completed those tests' agreement prior implementation

phase. Everyone of those tests was acting like a requirement and thus some of them was not finally executed as during the implementation it was inferred that this test case was capturing a real requirement or even the requirement wan not anymore valid. Responsible of those tests' execution was exclusively the testing team in association with the customer's site.

Table 9 Insurance Project Test Cases

Release	Test Cases	Test Cases Executed	Defects detected and fixed	Defects not fixed
Rel. 1	257	171	Cosmetic:3 minor:25 low:2 major:16 critical:2	Cosmetic:7 minor:0 low:2 major:0 critical:0
Rel. 2	426	383	Cosmetic:24 minor:37 low:0 major:20 critical:0	Cosmetic:3 minor:7 low:0 major:0 critical:0
Rel. 3	438	437	Cosmetic:23 minor:32 low:0 major:18 critical:0	Cosmetic:2 minor:1 low:0 major:0 critical:0
Rel. 4	602	591	Cosmetic:23 minor:47 low:21 major:36 critical:0	Cosmetic:2 minor:1 low:0 major:1 critical:0
Rel. 5	759	750	Cosmetic:25 minor:42 low:10 major:26 critical:0	Cosmetic:15 minor:34 low:0 major:4 critical:0
Rel. 6	1051	907	Cosmetic:47 minor:57 low:3 major:24 critical:0	Cosmetic:4 minor:3 low:0 major:1 critical:0

On the other hand these tests for the second company were written by the responsible developer after the closure of code writing. The tests ran from the client as acceptance tests and the responsible developer/s were supporting the customer throughout the testing phase. Results of those tests are depicted in table 10.

Table 10 Telco Product Test Cases

Release	Test Cases	Test Cases Executed	Defects detected and fixed	Defects not fixed
Rel. 1	43	43	Cosmetic:0 minor:7 low:5 major:3 critical:0	Cosmetic:0 minor:0 low:4 major:0 critical:0
Rel. 2	103	103	Cosmetic:0 minor: 7 low:3 major:5 critical:0	Cosmetic:0 minor:4 low:1 major:0 critical:0
Rel. 3	156	156	Cosmetic:0 minor:13 low:0 major:12 critical:0	Cosmetic:0 minor:4 low:0 major:3 critical:0
Rel. 4	201	201	Cosmetic:0 minor:28 low:14 major:3 critical:02	Cosmetic:0 minor:7 low:4 major:1 critical:0
Rel. 5	241	241	Cosmetic:0 minor:13 low:7 major:0 critical:0	Cosmetic:0 minor:2 low:0 major:0 critical:0

At a first sight we can observe that the first organization executed much more tests per code unit. Also it is somehow expected that first organization's tests exposed more defects than those of second's. Their defect correction rate was for both of organizations high and no versions for both projects revealed with critical issues.

Below we represent defects that have been detected after release and these data have been collected through the organozation's ticketing mechanisms.

Table 11Defects detected after release

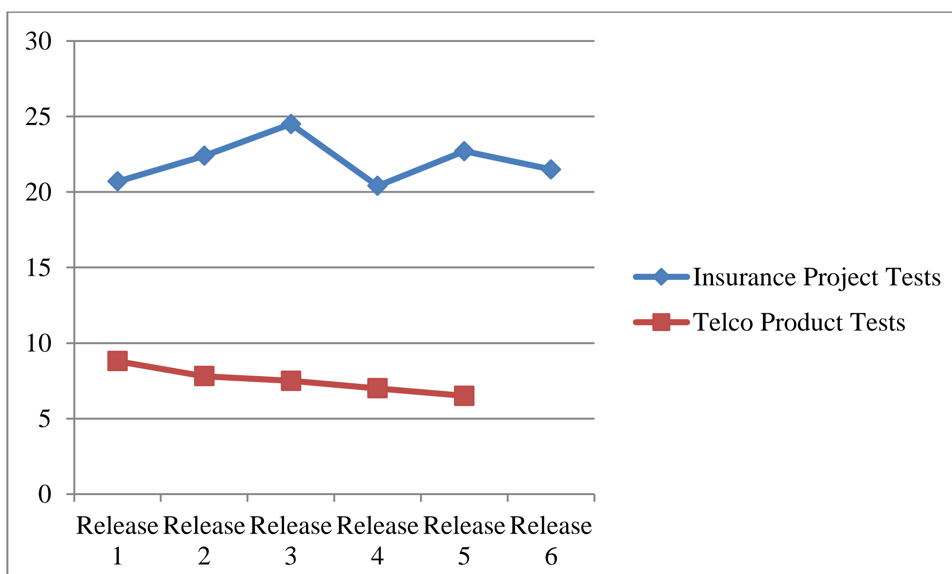
	Insurance Project	Telco Product
Release 1	cosmetic:3 minor:2 low:2 major:1 critical:0	cosmetic:0 minor:4 low:3 major:3 critical:0
Release 2	cosmetic:0 minor:2 low:3 major:0 critical:0	cosmetic:0 minor:3 low:2 major:3 critical:0
Release 3	cosmetic:0 minor:1	cosmetic:0 minor:2

	low:1 major:0 critical:0	low:0 major:2 critical:0
Release 4	cosmetic:0 minor:2 low:4 major:2 critical:0	cosmetic:0 minor:2 low:1 major:2 critical:1
Release 5	cosmetic:0 minor:1 low:2 major:0 critical:0	cosmetic:0 minor:2 low:2 major:1 critical:0
Release 6	cosmetic:0 minor:1 low:1 major:0 critical:0	

From the above table we can conclude that first organization's post-release defect rate was better than this of second even though during the first release the results were shown better for the second. Especially at the categories of major and critical defects the first organization have only 3 major issues, none of them critical, meaning that at no time there was an outage of service. Oppositively at second's organization product heve been occurred 12 defects one of them critical.

Comparing the 2 organizartions' results we can provide a metric about their pre-release and post-release defects per lines of code and how many test cases have been created/executed per lines of code as well. Firstly in image 1 we depict the test cases executed per 1000 lines of code for each delivery for both organizations.

Image 1 Test Cases Executed/ 1000 lines of code



It is obvious that the first organization used to execute many more tests compared with the second one. These results they are somehow expected as the first organization had a team of five people that dealed explicitly with testing. For the second organization there was a responsible engineer that executed and also created the test cases.

On tables 9 and 10 we can also notice the defects that have been found during development time and how many of these could not be fixed and carried through the next delivery. The chart that is depicted in image 2 compares these results. Of course, these defects, we have to remention that were noticed prior release and even though they are collected with informal tools and processes (e-mails, excel documents written by testing teams etc) they are typical of both organizations' debugging data. A graphical representation of pre-released defects per lines of code combined with those that were carried to the next iteration is provided below on image 2. For the sake of readability we have unified the 3 minimal categories of defects (cosmetic, minor, low) to a single one named as low-importance and the others (major, critical) to the one labeled as high-importance.

4. Conclusions

5. Discussion

6. References

1. P. Abrahamsson, J. Warsta, M.T. Siponen, J. Ronkainen, *New directions on agile methods: a comparative analysis*, in: Proceedings of the 25th International Conference on Software Engineering (ICSE'03), IEEE Press, 2003.

2. Beck, K., et al., *The Agile Manifesto.* 2001: p. http://www.agileAlliance.org.

3. Mikael Lindvall, Vic Basili, Barry Boehm3, Patricia Costa, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero, Laurie Williams, and Marvin Zelkowitz. *Empirical Findings in Agile Methods.*

4. Lowell Lindstrom and Ron Jeffries. *Extreme programming and agile software development methodologies*.

5. P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, *Agile software development methods: review and analysis*, VTT Technical report, 2002.

6. Kirsi Corhonen . *Evaluating the Impact of the Agile Transformation – a longitudinal Case Study in distributed context*.

7. K. El-Emam, "Finding Success in Small Software Projects," *Agile Project Management*, vol. 4, 2003.

8. Lucas Layman, Laurie Williams, Lynn Cunningham. *Exploring Extreme Programming in Context: An Industrial Case Study*, Agile Development Conference, 2004.

9. C. Jones, *Software Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley, 2000.

10. Olle T.W, Sol H.G, Tully C.J. *Information system design methodologies: A feature analysis*, 1983.

11. Jim Highsmith, *What Is Agile Software Development? An agile case story*.

12. David Cohen, Mikael Lindvall, Patricia Costa, *An Introduction to Agile Methods.*

13. Jussi Auvinen, Rasmus Back, Jeanette Heidenberg, Piia Hirkman, Luka Milovanov, *Improving the Engineering Process Area at Ericsson with Agile Practices.A case Study*.

14. Tore Dyba, Torgeir Dingsoyr*, Empirical studies of agile software development: A systematic review*, 2008.

15. S. Ilieva, P. Ivanov, E. Stefanova, *Analyses of an agile methodology implementation,* in: Proceedings 30th Euromicro Conference, IEEE Computer Society Press, 2004

16. Aldo Dagnino, Karen Smiley, Hema Srikanth, Annie I. Antón, Laurie Williams, *Experiences in applying agile software development practices in new product development*.

17. Jari Vanhanen, Jouni Jartti and Tuomo Kähkönen, *Practical Experiences of Agility in the Telecom Industry.*

18. Cockburn A., *Agile Software Development*, 2002

19. F. Macias, M. Holcombe, M. Gheorghe, *A formal experiment comparing extreme programming with traditional software construction*, in: Proceedings of the Fourth Mexican International Conference on Computer Science (ENC 2003), 2003.

20. Scott Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process,* 2002.

21. Paul R. Allen, Joseph J. Bambara, *Sun Certification for the Enterprise Architect,* 2007.

22. David Talby, Arie Keren, Orit Hazzan and Yael Dubinsky, *Agile Software Testing in a Large-Scale Project,* 2006.