

Helsinki Metropolia University of Applied Sciences  
Degree Programme in Information Technology

**Syed Adnan Ali**

**Continuous Integration and Test Automation for  
Symbian Device Software**

Bachelor's Thesis. 1<sup>st</sup> April 2010  
Instructor: Markus Strand, Specialist  
Supervisor: Olli Hämäläinen, Senior Lecturer  
Language Advisor: Taru Sotavalta, Senior Lecturer

Author	Syed Adnan Ali
Title	Continuous integration and test automation for Symbian device software
Number of Pages	88
Date	01 April 2010
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Instructor Supervisor	Markus Strand, Specialist Olli Hämäläinen, Senior Lecturer
<p>Recently, Continuous Integration (CI) has been introduced to one of the biggest companies to enhance the quality of work, processes and even tools. Among a few big changes in this regard are CI builds and test automation processes with an aim to reduce development and production time, to establish an environment of better communication, to decrease processes and manual work, and to enhance quality of low cost products. Thus, the purpose of this project was to see how to implement test automation within the company and in the end to analyze the results of the change continuous integration had brought into the company.</p> <p>My contribution was to automate the build process in general and test automation in particular, which was achieved by the development of Helium Build Framework and setting up new required standards to assist the automation processes.</p> <p>The outcome of the project was test automation in a CI build environment, which was not only achieved but it advances the processes and practices to reduce development and production time. Quality of the products has increased as the testing is done continuously without human interaction. Hence, test automation has minimized the risks involved in testing and reduced production cost.</p>	
Keywords	continuous integration, test automation, Symbian, Agile, software, development, Scrum, Extreme programming, tools, testing, software build, Helium, ATS

## **Acknowledgements**

I am deeply indebted to Markus Strand, who has been my instructor for this work. I also extend my gratitude to my supervisor, Olli Hämäläinen and language advisor, Taru Sotavalta. I am grateful for their cooperation, encouragement and for always being so supportive and cooperative throughout the thesis. I would like to give a special credit to my great teacher, Dr. Muhammad Tahirul Qadri, for his invaluable guidance throughout my life.

I am deeply and forever obliged to my family for their love, support and encouragement throughout my entire life, especially to my sister Rahila Sohail.

I am thankful to all my colleagues at Nokia for extending their technical support and advice during the work, especially Tomi Kallio for his inspiring and motivational personality and Janne Ronkainen.

In the end, my sincere thanks go to my friends and seniors at Metropolia for their help and motivation, especially Abdul Basit.

Syed, Adnan Ali

Espoo, April 2010

## Contents

Abstract

Acknowledgements

1. Introduction.....	6
2. Software Development Processes and Models .....	8
2.1 Software Development Process .....	8
2.1.1 Development Environments.....	9
2.1.2 Requirement Engineering .....	9
2.1.3 Design .....	10
2.1.4 Implementation .....	11
2.1.5 Versioning.....	12
2.1.6 Testing.....	12
2.1.7 Continuous Integration.....	12
2.2 Software Process Models .....	13
2.2.1 Waterfall Model .....	13
2.2.2 V-Model.....	15
2.2.3 Agile Software Development.....	16
2.2.4 Scrum .....	18
2.2.5 Extreme Programming .....	19
2.2.6 Test Driven Development .....	20
2.3 Software Builds.....	21
2.3.1 Build tools and processes .....	21
2.3.2 Software build framework and Helium.....	22
3. Continuous Integration.....	23
3.1 Overview.....	23
3.2 Definition of continuous integration .....	23
3.3 Development in the CI Environment .....	24
3.4 Practices of Continuous Integration.....	25
3.5 Benefits of Continuous Integration .....	27
3.6 Disadvantages of Continuous Integration .....	27
3.7 Continuous Integration Software Tools .....	28
3.7.1 CI Build Tools.....	28
3.7.2 CI Build Schedulers .....	31
3.7.3 Other CI Support Tools.....	33
4. Software Testing and Test Automation .....	35
4.1 Software Testing .....	35
4.1.1 Testing Methods.....	37
4.1.2 Testing types .....	38

4.2 Test Automation.....	41
4.2.1 Overview.....	41
4.2.2 Test Automation Framework .....	43
4.2.3 Advantages of Test Automation.....	43
4.2.4 Common Problems of Test Automation .....	45
5. Symbian Foundation and Build Machinery .....	47
5.1 Overview.....	47
5.1.1 Symbian Build Machinery .....	47
5.2 Symbian Test Frameworks and Testing Tools.....	49
5.2.1 Test Frameworks.....	49
5.2.2 Testing Tools .....	51
6. Software Development and CI for Symbian .....	53
6.1 Agile Software Development Environment.....	53
6.2 Symbian Builds and Continuous Integration .....	54
6.2.1 Symbian Builds.....	54
6.2.2 Continuous Integration Builds .....	56
6.3 Benefits of CI builds Over Manual builds .....	57
7. Test Automation in Symbian Platform .....	59
7.1 Overview.....	59
7.2 Automatic Test System (ATS).....	59
7.3 Role of Helium in the Symbian Test Automation Process .....	61
7.3.1 Directories and Files needed for a testdrop creation .....	61
7.3.2 Generation of <i>testdrop</i> for API and Unit tests .....	63
7.3.3 Generation of <i>testdrop</i> for UI tests .....	67
7.3.4 ATS Server Executes the Test and Generates Reports .....	68
7.3.5 Acceptance Tests for the Helium Scripts.....	69
7.4 Survey on Continuous Integration and Test Automation:.....	69
8. Conclusion .....	74
References.....	75
Appendix A – <i>MT_CPbk2AddressSelect</i> Test Component’s Files .....	79
Appendix B – Configuration Files .....	82
Appendix C – The ‘testdrop’ and the <i>Test.xml</i> File .....	83
Appendix D – The Survey on CI and TA .....	86

## 1. Introduction

*“In business, the competition will bite you if you keep running; if you stand still, they will swallow you.” – William Knudsen [52]*

Competition in the field of information technology is getting intense by every minute. The demands are increasing as well as choices. Brands are still trusted but not as before when a few brands used to have a big share in the market and an impact on consumers. There are more choices available, and as the internet is making the world a global village, people are now more aware of right or wrong and good or bad. By writing and reading reviews they help changing trends quickly. If William Knudsen would be alive today, he could have said, if you *slow down*, they will swallow you.

To meet the demands, satisfy consumers, and not to get lost in a crowd of IT companies one must produce user friendly, reliable, good quality and low cost products, whether it is hardware, software or a mix of both. For that, new technology, methodology, tools and processes must be adopted which better fulfil the needs and the requirements to stay in the race. As technology is growing, new methods and processes are evolving and make it difficult to choose or switch among them. Agile methodology is one of them. It provides a new, easy and adaptable way of working and is considered successful in many medium and large scale companies.

In this project, I will focus more on continuous integration and test automation, which are the most widely used practices of agile methodology and its tools and processes, for example, scrum, extreme programming, test driven development and so on. This project is based on a study of implementation of continuous integration and automation in a large-scale company.

Since technologies are emerging and have dependencies on other technologies, it is not possible to talk about one without others; thus, I will discuss relevant technologies, methods, processes and tools in light of continuous integration and test automation. The goal of my final year project is to create a test automation system as part of continuous integration processes, to execute tests on Automatic Test System (ATS).

The task is to develop and integrate processes, which, during the build, gather test sources and required data, and bundle them to be able to execute on Symbian devices. This will also require setting up new standards and modify current development and build practices to keep the testing process generic. Another aim of the project is to provide an understanding of new technologies, tools and processes to enhance development and production; qualitatively and quantitatively.

The final goal of this project is to find answers, to the questions, from continuous integration perspective that how this can be achieved by using a combination of processes and tools. Is this adaptable, can this improve development and production, how easy it is to maintain a mix of processes and tools and what flexibility will this change bring for future compliance.

## **2. Software Development Processes and Models**

### **2.1 Software Development Process**

Software development is the set of activities that results in software products. Software development may include research, new development, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.

Especially the first phase in the software development process may involve many departments in an organization, including marketing, engineering, research and development and general management. To undertake the task of software development, proper tools and resources are required in the form of development environments [17]. This can be done either by building software from scratch or by using existing software and changing or extending it to meet new requirements. In real world, the latter situation is more prevalent.

The software development process is very complex and it depends on human judgement and creativity, just like any other project. Therefore, it is impossible or very difficult to automate the whole process. One of the factors adding to this fact is the sheer possible scenarios that one faces in software development processes. There is no panacea. Each software development project needs a second thought in order to maximize efficiency. However, there are some parts that can be automated and used for a specific type of project.

Even in the same company there can be different software development processes in use. By looking at different processes in use in the real world, it is possible to categorize the fundamental steps that constitute all software development processes. They are:

- **Software Specification:** Functionalities, limits and outputs of the software are specified explicitly in a clear way.
- **Design and Implementation:** It includes planning, design and actual implementation of the software code.
- **Validation:** Proper testing and making sure that software complies with the specification in the step first above.



- Evolution: Keeping an eye on changing requirements and evolving the software accordingly. [17, 43]

### **2.1.1 Development Environments**

The software development environment (SDE) is the entire environment (applications, servers, network) that provides comprehensive facilities to computer programmers for software development. It should not be confused with an integrated development environment (IDE). An SDE consists of the following:

- Requirements management tools
- Design modelling tools
- Documentation generation tools
- Integrated development environment (IDE)
- Code Analysis tools
- Code Referencing tools
- Code Inspection tools
- Software Building tools (Compile, Link)
- Source Repository (Configuration Management)
- Problem reporting / tracking tools

### **2.1.2 Requirement Engineering**

Problems related to software system development are complicated and difficult to comprehend in the first place. This is especially true if the system under consideration is a new one and there is no existing infrastructure. Subsequently, it is difficult to exactly determine the requirements of the system, and what the customer wants to have as a final product. The description of the services and constraints of the software system are collectively known as requirements, and the principal activities undertaken to elicit and analyze the whole set is known as requirement engineering. The requirement engineering process involves the customer to a great extent. It is important to get as much information as possible from the customers and compel them to express all that they need. This is helpful in drawing a clear picture of the software, and determining its

final shape, as well as improving the efficiency of all the other different processes. The efficient and detailed set of requirements, duly agreed by all the stakeholders, reduce changing the software in the later stages and thus control the overall cost of the project.

The best strategy is to adopt a multi-level description for each software project. One way is following a three tier method, with the requirements being developed as User, System and Software design specifications. These levels can be defined as follows:

- User requirements: A high level description of the system. It may include simple statements and diagrams, explaining different parts of the system.
- System requirements: More detailed narration with functional explanation. This is usually used as a contract between the customer and the software provider.
- Software design specification: All parts of the system specified, explaining their interfaces, internal and external functionalities as well as constraints on implementation. Developers, managers, and system engineers are the target audience for software design specifications.
- Coding and testing: The design specifications are used to write the code. Developers also write various unit and acceptance tests for the code. [17, 99]

With these three levels of documentation, one can make sure that software development process takes off smoothly and unwanted delays are minimized.

### **2.1.3 Design**

Design documents give an overview of what lies ahead. They specify all parts of the system in functional detail, and their purpose. To convert this set of specifications into a working and executable product, the software needs to be designed and implemented. Software design is the description of the software to be implemented. It also involves breaking down the system into smaller, more manageable components. These sub-systems provide related services and functionalities, and their interfaces are properly defined to enable them to communicate with each other, so that they can understand a common language.

Requirement specifications are used to develop a top level architectural design, which involves identifying simple sub-systems. This gives rise to a set of abstract

specifications regarding these sub-components. This is followed by interface design between components, and then design of the components themselves. This completes the system level design of the software product. Data structure design and algorithm specifications to carry out individual tasks and functionalities are defined later in 2.1.4. The software design is an iterative process, with all the above steps being repeated until a satisfactory design emerges. [17, 57]

Many different methods are being used to carry out design process. They include data-flow model, entity-relation model, interaction-based model, object oriented methods, etc. It should be noted that software design is an informal process and different designers will arrive at different designs, although they use same set of system specifications. Designer's creativity plays an important part in the whole process.

#### **2.1.4 Implementation**

Software implementation is the process of actual coding, where the software is written in a computer programming language, compiled, and debugged. The output of the software implementation stage is the actual executable program code, which can be validated by testing. The implementation process can be streamlined by the usage of some automated tools. These tools can generate the overall skeleton of the program code. For example, class definitions or interface details. The programmer only needs to add details of individual operations to pre-defined blocks. On the other hand, programmers can also be asked to start from scratch and perform all the definition steps themselves. This can be accomplished by superiors, who then assign program blocks to each team. [17, 56]

Coding is interleaved with debugging. Programmers usually test their code themselves using professional debuggers. This is helpful in getting correct functionalities of the software at this stage of development. Debugging should not be confused with overall testing. Testing is concerned with the discovery of faults, while debugging is the process of removing those bugs.

### **2.1.5 Versioning**

Software implementation is a huge task and it usually involves many programmers working on the same piece of code or program. Each programmer contributes to the source code as software moves towards completion. In these circumstances, there is a need to maintain one central copy of the code which is accessible and visible to all the participating programmers, who can view, analyze and edit it. To fulfil these objectives, software version control or source control is used. In the version control, the team of programmers developing the software is bound to commit all their additions and changes to a central repository. Each commitment is time stamped and the identity of the committing team member is attached to it. Usually, a small message is also added, which explains the reason and motivation behind the commit.

In addition to software development, the version control system is also used with word processors (e.g. Google docs) and some content management systems (e.g. Wikipedia). There are many version control systems available, for example, SVN, CVS, GNU arch, Synergy and Mercurial. [17, 642]

### **2.1.6 Testing**

Modern software systems are huge entities. They consist of several sub-systems, each of them having its own set of functions. Further breakdown show the sub-systems being composed of modules, which in turn are composed of procedures and functions.

Different sub-systems are assigned to different teams for development. Each team is responsible for proper functioning of its own system, block or module. Programmers debug their code to find and fix bugs. Therefore, testing and validation has to be carried out in different steps, with unit level, module level, sub-system level and system level testing. After validation of the software package and fixing of the bugs found, the software is ready for delivery. This section will be discussed in detail in section 4.1

### **2.1.7 Continuous Integration**

As explained in section 2.1, software development consists of different steps, starting from requirement specifications and finishing with testing and followed by deployment. This whole process seems to be straightforward and simple. For single person projects

or for projects with a few persons involved, work is manageable and external dependencies are controllable. However, as the complexity of the software and size of the development team increases, a need arises to integrate different components and test them quite often. It is too risky to keep integration of different components to the last stage, and then face the inevitable delays and budget overruns. Continuous integration (CI) is the solution to all these problems. CI reduces these risks by integrating the software components at small intervals during the project, thus pointing out the defects and resulting in their early fixing. [21]

CI is a software development method in which integration of code from each member of the team is carried out regularly. Usually, every team member commits code to a central repository at least once a day, and thus there are multiple integrations per day. Each integration is then verified by an automatic build mechanism which also includes testing. Therefore, problems in the software system are detected on a daily basis and can be corrected for testing in the next integration build. Each build has to pass all the tests in order to be qualified as successful. Moreover, automated tools can be used to review code quality and carry out dependency analysis, further improving software quality. CI will be explained in detail in chapter 3.

## **2.2 Software Process Models**

The software process model is an approach which is designed to accomplish the task of software development. There are many different models being used in industry. Each model looks at the software development process from one angle, thus trying to optimize the resources and overcome deficiencies. Thus, each model follows a particular life cycle to complete software development process. In this section, widely used development models are discussed.

### **2.2.1 Waterfall Model**

Waterfall model is one of the first software development models which received wide acceptance. It was introduced by Winston Royce in 1970. Royce did not call it waterfall; instead he was using this model to criticize commonly used software practices. [31] In the waterfall model, the whole process of software development is

divided into different phases. One phase follows another, in step-wise fashion [2] as shown in figure 2.1. Therefore, because of this cascading of different phases, the model gets its name. Next phase is only started after previous phase is successfully finished.

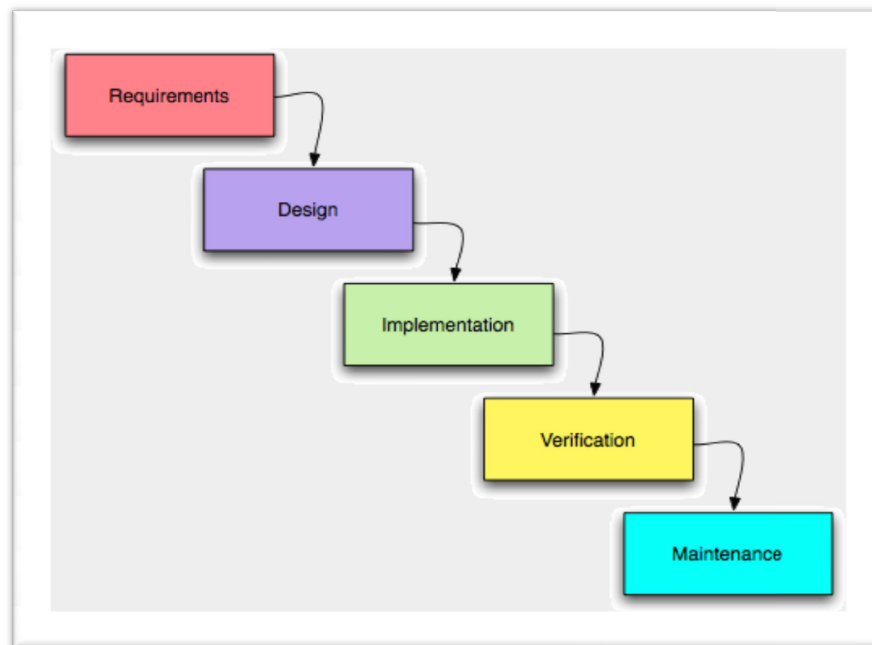


Figure 2.1: Phases of the Waterfall model [3]

Waterfall model can be divided into five phases. Each phase is explained below.

- Requirements analysis and definitions: Requirement engineering as explained in section 2.1.2 is carried out in this phase. System users are consulted and a clear picture of the system goals, limits and external interfaces is drawn.
- System and software design: This phase precedes the actual coding. At this stage, system specifications from previous stage are studied and system design is prepared in detail. It includes breaking down of the system in to smaller components and then carrying out detailed design activities up to module, data structure and algorithm level.
- Implementation and unit testing: Software coding starts in this phase. System is divided into smaller components, called units. Each unit is then coded and unit-

tests to verify the functionality of each unit are defined. All unit-tests must pass before the process advances to the next stage.

- Integration and system testing: Individual program units are integrated in this phase. The previous phase ensured that each unit is working as defined in design specifications. This phase guarantees that all units when put together work in a desired way. System testing is carried out and the bugs are fixed. Also, the completed piece of software is deployed at the customer site for testing [2].
- Operation and maintenance: This is the longest or better to say never ending phase of the waterfall model. It starts with the actual deployment of the system. Generally, problems with the system, which were not found during development, start to emerge when the system starts functioning in a real environment. Since this is a continuous process and system faults are discovered continuously during the operation of the system, therefore, this is also known as the maintenance phase.

### **2.2.2 V-Model**

An extension to the waterfall model is the V-model software development process defined by Paul Rook, which does not terminate downwards, instead it moves upwards after the coding stage. This model can be divided into three phases: verification (left), coding (bottom) and validation (right) of the V as shown in figure 2.2. However, various testing methods are used to validate software, such as unit testing, integration testing, regression testing, system testing and acceptance testing. [2]

As the verification and validation phases run in parallel, more time can be saved and better quality software is produced as compared to the waterfall model. However, the customer is involved at the later stages as in the waterfall, which is mostly a cause of serious changes at the end of the development.

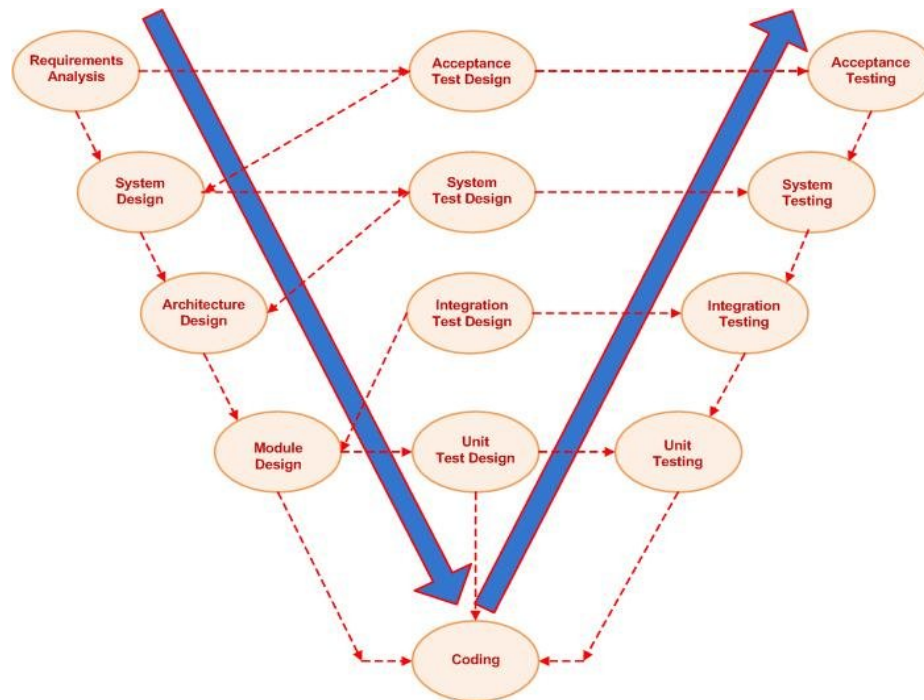


Figure 2.2: V-model software development stages [4]

V-model is not considered a good choice for small or medium scale projects as it is expensive.

### 2.2.3 Agile Software Development

Agile software development is a set of practices for iterative development by combining the number of software development methodologies. This nimble way of development boosts a controlled project management process and encourages frequent reviews and adaptation. Figure 2.3, provides a glance of agile software development methodology.

Agile software development term was first used in 2001 after the formulation of “Agile Manifesto” [32] which consists of twelve principles:

- Satisfy the customer through early and continuous delivery.
- Welcome changing requirements, even late in development.
- Deliver working software frequently.
- Business people and developers work together daily.
- Build projects around motivated individuals.



- Convey information via face-to-face conversation.
- Working software is the primary measure of progress.
- Maintain a constant pace indefinitely.
- Give continuous attention to technical excellence.
- Simplify by maximizing the amount of work not done.
- Teams self-organize.
- Teams retrospect and tune their behaviours. [33]

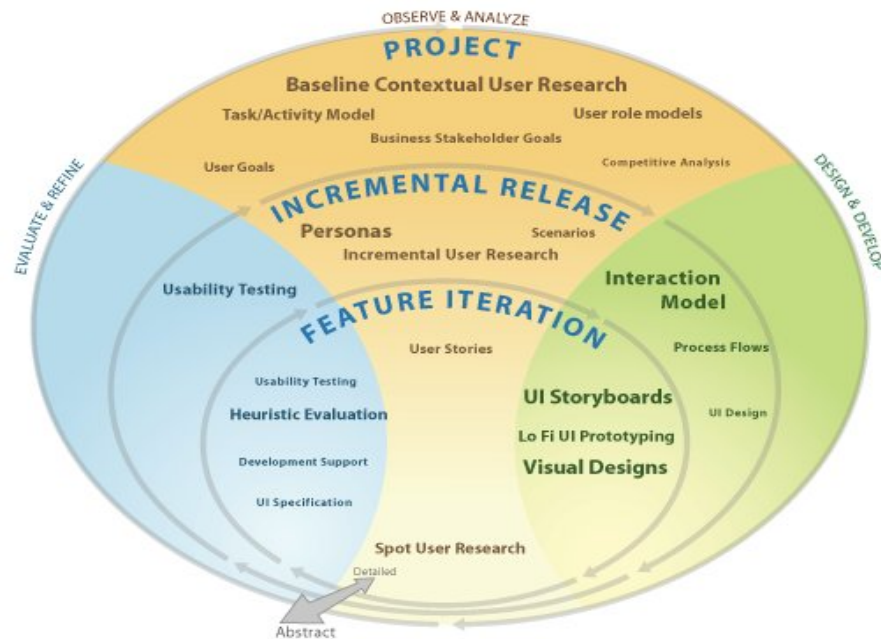


Figure 2.3: Agile Software Development Methodology [37]

There are several agile development methods and most of them promote development, teamwork, collaboration and process adaptability throughout the life cycle. The tasks are broken down into short iterations and the whole team works through a full software development process/cycle (see section 2.1), for each iteration. The goal is to have minimal bugs at the end of the release rather than many but erroneous features. During the planning meetings team members take responsibilities for the tasks needed to deliver functionality for the particular iteration.

Specific tools and techniques such as continuous integration, automated testing, pair programming, test driven development, design patterns, code refactoring and other techniques are often used to improve quality and enhance project agility.

## 2.2.4 Scrum

Scrum is an iterative incremental framework and commonly used with agile software development methodologies, in which teams maintain a product backlog of the features which they have to develop (also known as stories in agile development terminology) and frequently release small features. Figure 2.4 summarises scrum.

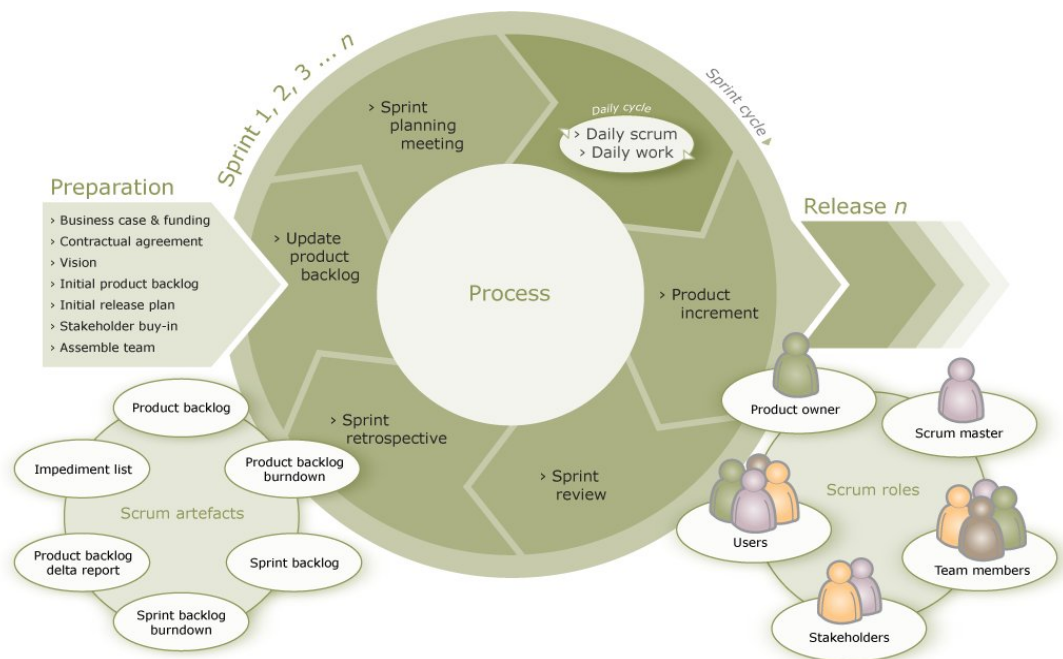


Figure 2.4: Scrum Overview [38]

The product backlog is prioritized on a regular basis by a product owner together with the customers of the product, who raise their requirements and set priorities for the required features. The stories of high priority are placed on top of the backlog.

Teams decide how frequently they want to make releases. There can be two to four weeks development and release cycle which is called a sprint. At the beginning of a sprint there is always a sprint planning meeting in which the development team members pick the stories and break down every story into smaller and simple doable

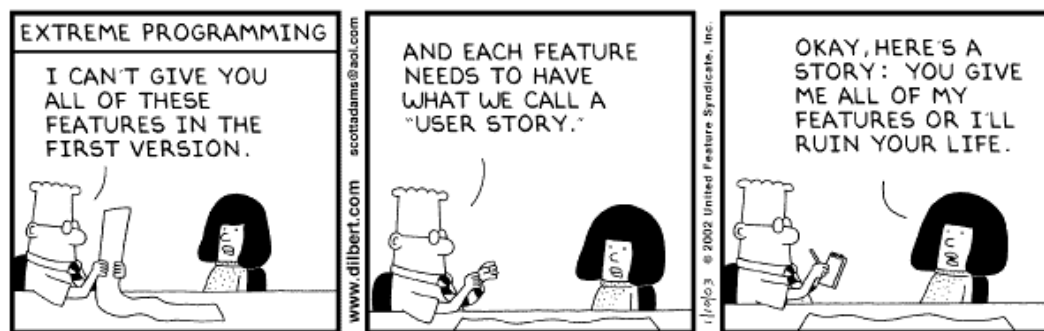
tasks. During a sprint the sprint backlog is frozen and no one is allowed to change the sprint backlog

Scrum is alleviated by a Scrum Master whose job is to remove hindrances to the ability of the team to deliver sprint goal. Scrum Master is not a team leader but acts as a buffer between the team and distracting influences. Scrum Master's major role is to protect the team and keep them focused on the tasks in hand.

### 2.2.5 Extreme Programming

Extreme programming (XP) is one of many popular agile processes. It has already been proven to be very successful at many companies of different sizes and industries worldwide. Extreme programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software from day one. They deliver the system to the customers as early as possible and implement changes as suggested [34].

The most surprising aspect of extreme programming is its simple rules that user stories are always written. Release planning should give a clear release schedule for frequent and small releases [34]. Therefore, the following comic strip, figure 2.5, is not a representation of XP in which a customer is asking all the features in a first release.



Copyright © 2003 United Feature Syndicate, Inc.

Figure 2.5: Dilbert on extreme programming [53]

Projects should be divided into small iterations and all of them should be planned. The team should be given an open workspace and a sustainable pace should be set. A stand-

up meeting (daily scrum) is arranged each day and project velocity is maintained [34].

Figure 2.6 shows a diagrammatic overview of XP.

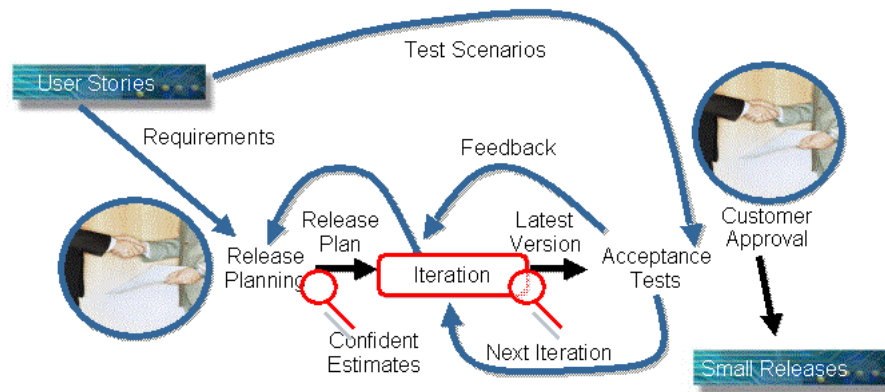


Figure 2.6: Extreme Programming Process Model [39]

Developers are encouraged to review the changes with the customers as often as possible. The standards must be agreed on beforehand and maintained throughout the development process. XP encourages writing unit tests first (Test Driven Development). All code must have unit tests and acceptance tests which should be executed frequently. The tests must be passing before any release is set out. These rules define an environment that promotes team collaboration and empowerment. [34]

### 2.2.6 Test Driven Development

Test driven development is a style of software development that runs well with CI. TTD is based on the fact that nothing goes to the integration build unless it has an associated test, which can be run to verify the functionality. It is 'think before you act' approach. For the whole functionality, that a programmer codes, he or she has to write a test for it, only then it can be committed to the central repository. After creating a test, code is then written to get this test passed, and this process continues until the whole functionality is coded. Thus, an exhaustive set of programmer tests is maintained and run after every integration. [30]

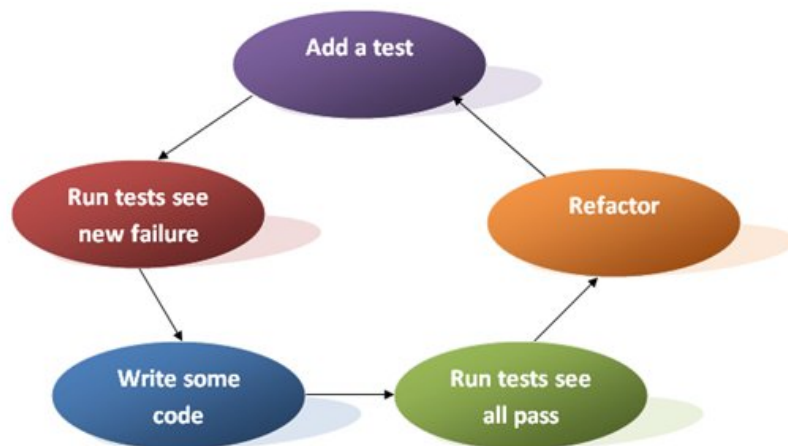


Figure 2.7: Test Driven Development Flow [40]

Test-Driven Development, as illustrated by figure 2.7, is a core part of the agile process (which is explained in section 2.2.3). It is an extremely high discipline, very effective, and resilient to change. [30]

## 2.3 Software Builds

The term software build refers either to the process of converting source code files into standalone software artifact(s) that can be run on a computer, or the result of doing so. One of the most important steps of a software build is the compilation process where source code files are converted into executable code.

While for simple programs the process consists of a single file being compiled, for complex software the source code may consist of many files and may be combined in different ways to produce many different versions.

### 2.3.1 Build tools and processes

The process of building a computer program is usually managed by a build tool, a program that coordinates and controls other programs. Examples of such a program are *make*, *ant*, *maven* and *SCons*. The build utility needs to compile and link the various files, in the correct order. If the source code in a particular file has not changed, then it may not need to be recompiled (may not rather than need not because it may itself

depend on other files that have changed). Sophisticated build utilities and linkers attempt to refrain from recompiling code that does not need it, to shorten the time required to complete the build.

A few examples are ElectricAccelerator which speeds up *make*, Microsoft Visual Studio, and Apache Ant based builds by parallelizing them and ElectricCommander which is a Web-based application for defining and executing distributed processes such as those used for building and testing software applications [35]. Modern build utilities may be partially integrated into revision control programs like Subversion.

### 2.3.2 Software build framework and Helium

A more complex process may involve other programs producing code or data for the build process. The combination of build tools which are combined to execute a full build which includes a number of build stages is known as a build framework.

#### Helium build framework

A well known example of a build framework is Ant based Helium, which is an open source tool developed in Nokia and distributed by the Symbian Foundation, primarily for Symbian builds.

It compounds many other CI tools, such as Hudson, CruiseControl (see table 3.4) and support for version control systems like Mercurial, Synergy (see table 3.5). For build compilation Helium supports Make based EBS, ElectricCommander and ElectricAccelerator. SBSv2 (discussed later in chapter 3, table 3.2) provides all the functionality required to build the Symbian platform. Helium is also used to automate the other steps that are performed as part of running a full build: for instance, fetching the source code, preparing the environment, creating ROM images, packaging tests for test automation and analysing and publishing the results [44].

### **3. Continuous Integration**

#### **3.1 Overview**

Software development is an extremely complex process that consists of sub-processes. In many cases the sub-processes are also divided into many sub sub-processes. However, software may consist of smaller components or features, each of which has its own development cycle.

The features can be developed by several teams which cannot be collocated geographically. Once the features are developed they are enthusiastically integrated to see fully developed and functional software. It is very normal that the excitement turns into disappointments with discovery of errors. This can cause uncertainty and disappointment among the developers and integrators, and can take several iterations before an acceptable limit is achieved and can be costly.

Continuous integration (CI) was first discussed by Martin Fowler and Kent Beck at the end of 1990s [20], Continuous integration addresses late discovery of errors and prevents delay of software releases by integrating and testing the software components on an average daily basis.

#### **3.2 Definition of continuous integration**

Continuous integration can be defined as: Application of software engineering tools and processes to speeding up development with decrease in integration time, in more automated way [20]. It is not only about using tools in certain defined processes but it is about a culture in which development, integration, builds and testing is done. The culture is to set best practices for development and to choose set of efficient tools and then defining and applying processes.

Therefore, CI best practices is about establishing a culture where the dependencies over tools and processes are reduced to a minimum possible extent while they all are integrated to complete the task, that is, release of the software by reducing integration time, in a robotic way.

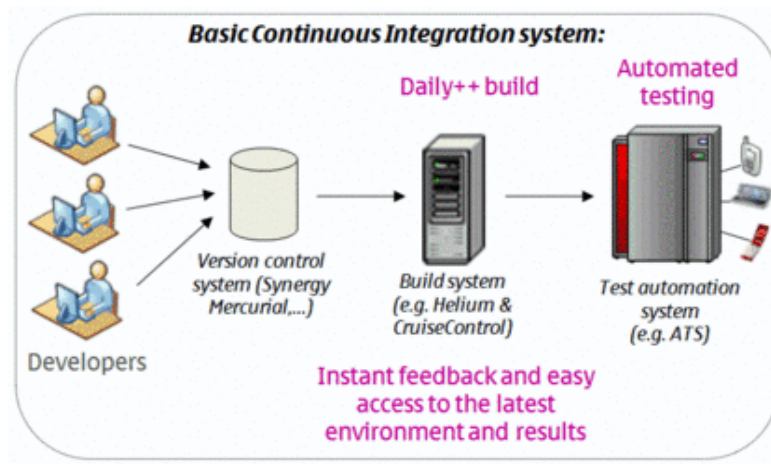


Figure 3.1: A basic continuous integration system [21, 5]

Figure 3.1 shows a basic CI system in which developers are checking-in code in code repository. Every day a software build is executed and it compiles and integrates all the features and components. The compiled image is then sent to the testing server where the compiled image is installed on the devices and several tests are performed. In the end feedback reports are sent back to the developers.

### 3.3 Development in the CI Environment

CI is a boost to an iterative and agile software development practice, in which developer commits, compiles and verifies new software change(s) more often and in more automated way.

In software development, more frequent and faster feedback have high importance, and these can be attained by improving processes and tools with added automation. The feedback shows the visibility and evolution of the software and also showing early errors and integration problems.

By resolving the identified problems at an early stage, better quality can be achieved and the shipment time to the market can be reduced.



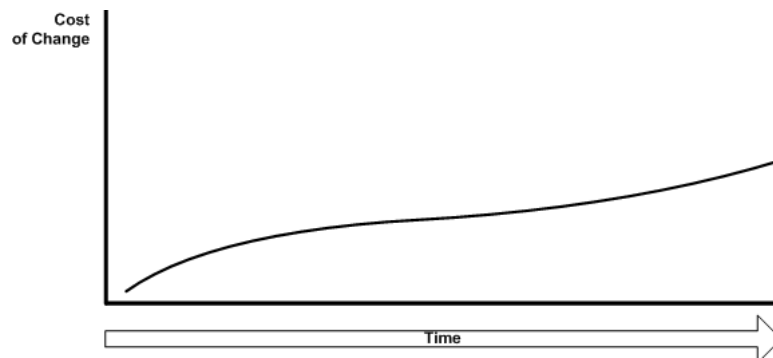


Figure 3.2: Error cost curve for agile software development [51]

Figure 3.2 shows that the errors rate is higher in the early development phase than later but the cost to solve the errors is much lower in the premature phase than afterwards. However, the defects found in different stages are gradually increasing which is because of feature integration.

There are, however, a few action points for a developer to maintain a better development culture, and they include

- Commit changes at least once per day
- Code little but commit to the repository
- Check feedback reports (build and tests error logs etc.)
- Immediate error fixing
- Maintain team cooperation and communication [20]

### 3.4 Practices of Continuous Integration

Martin Fowler describes key practices for continuous integration, to work efficiently in daily lives, which are briefly discussed below [20]:

*i. Maintain a single source repository:*

A revision controls system (or code repository) should be used and all the artifacts needed for the build are placed in the repository. Mainline version (or trunk) should be the place for the working software.

- ii. *Automate the build:*  
A build should have started by triggering a single command by using one of several available build tools. For example Make, Ant, MSBuild or IBM Rational Build Forge are some of the widely used tools in continuous integration environments.
- iii. *Make your build self-testing:*  
One of the best practices is Test Driven Development (TDD) in which developer first writes a test of required functionality first and then the actual code is written to make the test pass. All the tests are rerun once the whole code is written.
- iv. *Commit to mainline everyday:*  
Committing changes to the main development line prevents code conflicts, feature instability with other features, finding and fixing problems quickly. This is a good agile practice too as developers can break down hours into smaller chunks which helps track progress and provides sense of advancement.
- v. *Commit should build the mainline:*  
If the build starts on every commit and it fails then it should be fixed immediately, which prone to fewer errors and a stable environment is available for others to check in their changes.
- vi. *Keep the build fast:*  
The point of CI is to provide rapid feedback, so that the problems can be quickly identified.
- vii. *Test in clone of the production environment:*  
Test environment setup should be close to the production environment because production environments can be a lot different. There can be unidentified errors left in the software if the test environment in not closer to production environment.
- viii. *Executables are accessible:*  
The tests are performed on several levels other than CI build tests. For those testers and stakeholders the ready image (or executable) should be published to a shared location which will save testers' time to build and rebuild for any broken feature.
- ix. *Progress should be visible:*  
Feedback, reports and logs are the backbone of CI which helps in identifying causes of errors and build breaks. The build process should be visible to every one who is related to the project.

x. *Automate deployment:*

There can be several test environment setups on either a single or a cluster of live test servers, for a build to execute. In a complete automated build environment, an application should automatically be deployed to the live test server where everyone has access.

### 3.5 Benefits of Continuous Integration

There are many advantages of continuous integration in the software development and production environments, some of them are mentioned below:

- Defects are detected and fixed sooner because CI system runs a build several times or at least once a day [21, 29]. No last minutes fixings before releasing.
- Health of software is measureable such as complexity can be tracked over time [21, 30]
- Immediate unit testing helps discovering bug(s) or test failures, in such cases the code can be reverted back to a bug free state.
- Early warnings of broken, incompatible code and conflicting changes.
- Constant availability of build for testing, demo or release purposes.
- Instant feedback to developers on the quality, functionality or system-wide impact of code.
- Modular and less complex code is an outcome by developers because of frequent code check-ins.
- Metrics generated from automated testing and CI (such as code coverage, code complexity, features complete) focus developers developing functional, quality code and help develop momentum in a team.

### 3.6 Disadvantages of Continuous Integration

CI, where it has several benefits to enhance the development and production of the software, it has a few disadvantages too, which include

- Increased overhead in maintaining the CI system which involves several automated and manual processes.

- Too much change as there are too many processes that need to change to achieve CI for their legacy. [21, 32]
- Too many failed builds, as the CI cycle keeps on running on every code check-in by the developers, there can be unfixed errors which might cause several builds to fail in a sequence.

### 3.7 Continuous Integration Software Tools

Finding the best automated software tools for an environment and development process requires having clear understanding and defined needs. Of course, the best tool would be the one which will serve longer and reduce pain from the development team.

There is a number of software tools available for CI from repository to build to test to feedback reporters, but the most important criteria in choosing a tool is whether it does what is required. In the following an overview of a few CI tools is given under the categories:

- CI Build tools
- CI Build schedulers

#### 3.7.1 CI Build Tools

Build tools are used to compile the code and make a build. Essentially a build tool should provide following functionality:

Table 3.1: Functionality of build tools [21, 248]

Functionality	Description
<b>Code compilation</b>	A main ingredient in building software
<b>Component packaging</b>	After compilation software typically needs to be bundled into deployable component such as Java JAR files or Windows EXE files.
<b>Program Execution</b>	Good support to invoke a program in a target platform or on command-line interface.

<b>File manipulation</b>	Creating, copying, and deleting files.
<b>Development test execution</b>	Execution of automated developer of tests for the software
<b>Version control tool integration</b>	Automated responses to version control tools (or code repositories)
<b>Documentation generation</b>	Automated generation of API documents when build is run.
<b>Code quality analysis</b>	Generation of code coverage reports by analyzing the code.

Table 3.1 describes the basic functionality of a build tool of choice. It is always difficult to find the right tool, that is, all the features in onetool but some tools provide good features and almost all the required functionalities, which can, of course, be customized.

In table3.2 there are a few examples of widely used build tools.

Table 3.2: Examples of continuous integration build tools

Name of the tool	Description
<b>Make</b>	Make is a widely used utility to automate building executable programs and libraries from source code. Configuration files are called ‘makefiles’ which specify how to drive the target program. Makefiles, however, are not typically good for cross platform dependencies. The structure of make is not a programming language but it has a fair amount in common with declarative programming languages. [22]
<b>ANT</b>	Apache ANT is Java based build tool, similar to ‘Make’ and the execution requires Java platform and is best suited to build java code. But it is also a tool of choice for other development platforms. Ant uses XML files for configuring builds. Use of xml

	<p>files resolved the issue of creating platform independent configuration files, and on the other hand configuration files can be messy and can easily increase complexity.</p> <p>The XML file has <i>properties</i>, which are set to configure a build, for example, build number, paths, project name et cetera; and <i>targets</i>, which are the tasks or processes, these targets are called to invoke the operations. [26]</p>
<b>SBSv2</b>	<p>The Raptor Build system, sometimes referred to as Symbian Build System version 2 (SBSv2), is a new build system for building Symbian platform C++ applications, packages, or indeed the entire platform. Raptor can run on both Windows and Linux, and works with Symbian^1/Symbian OS v9.4 and later (only). Raptor builds are often significantly faster; when used with third-party parallel make programs, such as PVM Gmake and EMake, it can use multiple build machines to create a build farm [43].</p>
<b>Maven</b>	<p>Apache Maven is another open source tool that is able to build software project, run developer tests, produce a number of useful source quality reports, and generate a Web site to contain the output of all of these steps. [23]</p>
<b>Rake</b>	<p>Rake is a build tool like Make, but the configuration files, known as ‘rakefiles’ syntax is defined completely in Ruby and also has number of other differences. [24]</p>
<b>Groovy</b>	<p>It is an agile and dynamic tool mainly builds upon Java but has additional power features inspired by languages like Python, Ruby and Smalltalk. It makes writing shell and build-scripts easy, simplifies testing by supporting unit testing and mocking out-of-the-box and compiles straight to Java byte-code. [25]</p>

### 3.7.2 CI Build Schedulers

Build schedulers are used to schedule a CI build either once, twice or several times a day. Some most typical functionalities of a widely used build scheduler are summarized in table 3.2.

Table 3.3: Functionalities of build schedulers [21, 250]

Functionality	Description
<b>Build execution</b>	Execution of automated builds on periodic basis.
<b>Version control and build tool integration</b>	Fetching of complete set of files from version control for each changed file and build dependencies, this is crucial that how well the tool identifies the changes.
<b>Feedback</b>	Sending email, text messages or any other way of providing feedback to developers of build managers, on build completion or on failures.
<b>Build labeling</b>	Marking of artifacts which are contributed to a given build.
<b>Interproject dependencies</b>	Execution of any dependent build to make a main build successful. This is how a build scheduler is configured to know if there are dependent builds.
<b>User interface</b>	For altering configuration, checking and downloading artifacts and watching build status.
<b>Artifact publication</b>	Publishing the build reports, documentation, test results and other artifacts, and make them available to everyone who may in need, by providing, for example, a directory
<b>Security</b>	Configuring access rights over the build execution processes, viewership of artifacts and so on.

Table 3.4 lists some examples of build schedulers most of them fulfil the functionalities listed above in table 3.3.

Table 3.4: Examples of continuous integration build schedulers

Name of the tool	Description
<b>BuildForge</b>	IBM's BuildForge is a heavy-duty commercial build management tool that provides high-performance, distributed build, test, and deployment functionality. It provides web console for centralized user and administration access and for real-time software build process anytime, anywhere. Job process optimization ensures parallel and distributed jobs handling whereas, error log filtering and automated email notifications allow for rapid error detection and resolution. [27]
<b>CruiseControl (CC)</b>	Open source product CC is by far the most widely used CI server. CC is packaged as several complementary components, such as the main CC service, an optional reporting Web application and many others. CC is typically set up to run as a background process, with the Java Web application providing the front-end and reporting interface. [21, 266]
<b>Hudson</b>	Hudson monitors building a software project or jobs. Currently Hudson focuses on Building/testing software projects continuously, and Monitoring executions of externally-run jobs. It is considered a tool which provides easy installation and configuration options. RSS/Email integration, change set support, JUnit/TestNG test reporting, distributed builds and custom plugins are some the features of Hudson. [28]
<b>BuildBot (BB)</b>	BB automates compile/test cycle to validate code changes. It automatically rebuilds and tests the tree each time something has changed and quickly pinpointed build problems, before other



	<p>developers are inconvenienced by the failure. It supports several platforms e.g. Windows, Unix, Linux, Solaris etc. BB delivers status, through web page, email, IRC, and other protocols, and builds tracking, for instance, estimated completion time. [29]</p>
<b>Luntbuild</b>	<p>A recent addition to build schedulers, Luntbuild is another popular open source Web-based CI server. The Web-based user interface can be somewhat confusing and counterintuitive but it does offer more flexibility than other Webbased CI servers once the usability hurdle is overcome. [21, 271]</p>
<b>Continuum</b>	<p>Continuum supports many of the leading version control tools, such as Subversion and CVS. It includes an easy-to-use Web-based setup and user interface with availability of remote management capabilities. It provides various feedback mechanisms such as e-mail and instant messaging (IRC, Jabber, and MSN). [21, 266]</p>

### 3.7.3 Other CI Support Tools

There are many other CI tools/resources available. Almost all the CI build tools either provide all the functionalities of CI automated processes or they support some of them. But the missing functionalities can be integrated. If, for some reasons, the provided functionality by the CI tools is not suitable for the development team, then other CI tools can be integrated with the CI tool to enhance the capability of the tool and to maximize the output of CI automated builds.

In table 3.5, there are a few examples of the tools for each CI category.

Table 3.5: Continuous integration resources [21, 233]

CI Category	Tools
<b>Version control resources</b>	ClearCase, Concurrent Versions System (CVS), MKS, Subversion, CM Synergy, Mercurial, SnapshotCM, StarTeam, Visual SourceSafe, Perforce
<b>Database resources</b>	Hypersonic DB, Mckoi, MySQL, Oracle, PostgreSQL
<b>Testing resources</b>	Agitator, DbUnit, Fit, FitNess, Floyd, HtmlUnit, JUnit, JWebUnit, AntUnit, PyUnit, Pylint, NDbUnit, NUnit, Selenium, SQLUnit, TestNG, xUnit Test Patterns
<b>Automated inspection resources</b>	CheckStyle, Clover, Cobertura, EMMA, FindBugsFxCop, JavaNCSS, JDepend, NCover, NDepend, PMD, Simian, SourceMonitor
<b>Deployment resources</b>	Capistrano
<b>Feedback resources</b>	Ambient Devices, GoogleTalk, Jabber, X10, Java James (Apache Java Enterprise mail server), Lava lamps, Gaim
<b>Documentation resources</b>	Doxygen, Javadoc, Ndoc

Continuous integration is an application of software engineering tools and processes to enhance development and detection of errors at early stage by automating builds, tests, documentation and deployment by establishing an automated feedback loop between the processes, developers and the entire team members and customers who are involved in the development chain.

## 4. Software Testing and Test Automation

### 4.1 Software Testing

Meaning of software testing has a wider scope, it is about testing of functions, module, components consists of several modules, sub-system which consists of components and then fully functional system which can be comprised of several sub -systems.

These tests can be performed either manually or in an automated environment. Setting up an automated environment is an extremely difficult process, but once it is ready, the chaos of executing tests manually, creating and delivering tests results and reports to the specific development teams is ended. It reduces a lot of development and testing costs.

Software testing is a set of activities conducted with a goal of discovering software errors. [18]. Also, it is a software development process intended to assess a software item (such as system, sub-system, unit), features (functionality, performance) against the given set of system requirements. [19]

Testing is never an easy job, neither are all the defects identified within software. Testing is a process to produce optimized, bug-free code to a level of required satisfaction, meaning that the functional software should perform what it is expected to do with integrity with compliance mutually with other software modules, components and applications. However, in practice, some bugs or defects remain stick in test cases.

Hence, the testing can be broken down into component testing and integration testing. Finally, acceptance testing is performed once the system is fully developed and ready.

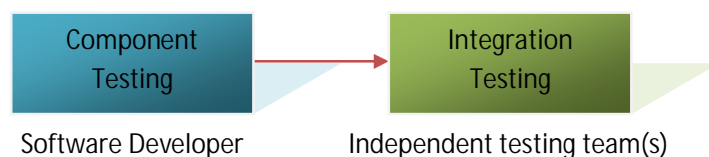


Figure 4.1: Testing phases [17, 441]

Figure 4.1 shows that there are two main test phases.

*Component testing* is the functionality of identifiable components like functions, or groups of methods collected together into a module of object are tested. *Integration testing*, the second stage, the components are incorporated to form a sub-system or a complete functional system and tests are performed to check integrity and interactions between the components and performance of the system as a whole.

There is one more test phase and that is acceptance testing phase. Generally, *acceptance testing* is known for testing customers or end user requirements specifications if they match with the system, subsystems, components or units.

Figure 4.1 shows how the testing phases are co-related, but in reality it is a little different than the one shown by Ian Sommerville in his book [17, 441].

I agree with both the testing stages but these are not performed one after another. It is more likely that new component errors are identified during the integration testing and hence the component testing phase overlaps the integration phase. Similarly, acceptance tests are performed throughout the development and integration phases to check whether the code, sub-system and the completed system are all according to the specification and fulfilling the requirements, in that case, both, component and integration testing phases are encircled within acceptance tests phase. Then figure 4.1 can be redrawn as figure 4.2.

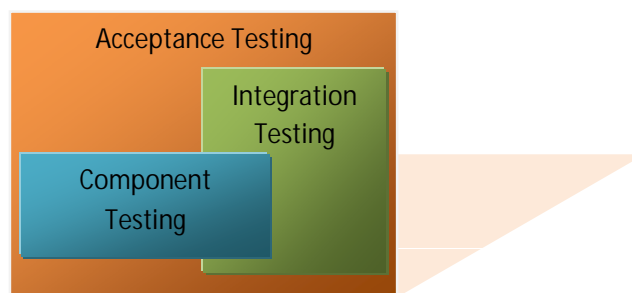


Figure 4.2: Testing phases, redrawing

Figure 4.2 shows that acceptance tests are part of component testing, integration testing stages as well as, acceptance itself is another stage, for instance, interface testing.

Therefore, acceptance testing is an integral part of both component and integration testing.

#### **4.1.1 Testing Methods**

In the previous sections it was discussed that why software testing is done and its importance, and development processes were discussed in chapter 2. Now, by relating them, the definition of software testing can be rephrased that it is a way to identify errors or to check if the code is erroneous during software development process.

Suitable software development process is important because it gives a level of ease or difficulty for finding out errors and also efficient ways of fixing the errors in less time and cost effectively. But at this stage a question arises how to find errors. This means that methods for identifying errors should be known.

I will briefly discuss the most prevalent strategies generally known as black-box testing, white-box testing and grey-box testing [6].

##### Black-box testing

In the black-box testing the goal is to be completely unconcerned about the internal behaviour and structure of the program. The system is considered as a black-box; data structure, implementation and internal behaviour are completely ignored and the main focus is the input and the output of the system. Tests are written to provide input to the system based on the identified input specification and the output is analyzed or tested to check if the system works as it should according to the required output specification.

Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing [6].

Tester inputs the data and sees the output produced by the test object. The requirement of providing thorough test cases, as an input, is a must to verify that at given input the output value either “is” or “is not” the same as the expected value according to the specification [7].

Tester mostly has no knowledge of internal code; the aim is to find out errors with a perception that *a code must have bugs*. Whereas black-box testing is also considered

like “walking in a dark labyrinth without a flashlight” because tester does not know how the software being tested was created. As a result there can be a situation where tester writes test cases to check something which requires only one test case and some parts of the code are left untested at all [8].

Types of testing fall in the black-box testing category include acceptance testing, integrating testing, functional and system testing and beta testing [9].

#### White-box testing

In white-box testing the internalities of the program are concerned, which may include any internal logic, function, module and/or group of modules, feature/component. The idea is to go through each and every statement in the program and execute at least once. White-box testing is also known as logic-driven testing [6].

White-box test cases should test different paths, decision points (both true and false decisions), execute loops, and check internal data structures of the application. The goal is to cover testing as many of the statements, decision points and branches in the code base as possible.

White box testing is used in three basic types of testing which are unit testing, integration testing and regression testing [9].

#### Grey box testing

Grey-box testing is performed at user level or at black-box level, but to create some test cases, knowledge of internal structure and program logics should be known. This is important during integration testing between two modules of code, written by two different developers are to be tested and interface is the only thing exposed for test. Integration testing is the only testing type which is common in both black-box and white-box testing.

### **4.1.2 Testing types**

Testing can be categorized into two main types, functional and non-functional testing. These two are further categorized.

### Functional Testing

Functional testing means to test under-developed and/or developed applications against the requirement specification for as less as a piece of code to a fully developed application. During these tests, code, data structures and logics for a class, module, component and a complete application are tested for acceptance. Generally six basic types of functional testing are known.

- *Unit testing*

White-box testing methodology applies to unit testing in which functionality of code is tested generally at function and/or class level. Developers write the code to test and verify the functionality of a piece of software.

- *Regression testing*

A white-box testing methodology applies to regression testing, in which a program's new functionality is tested. Regression testing is done by running all of the previous unit tests written for a program, if they all pass, then the new functionality is added to the code base. [9]

- *Integration Testing*

During this testing, modules are collectively tested, which are typically code modules, individual applications, client and server application on a network etc. [10] Main purpose is to reveal defects in the interfaces and interaction between integrated components.

- *Acceptance testing*

Acceptance testing is a black-box testing technique, performed on a software system to test customer requirements. Smoke tests are executed prior to integration testing to check whether the completed sub system meets the desired level of requirements in the specified hardware and software conditions. [11]

- *System testing*

System testing falls within black-box testing and is done to ensure that the entire software system is in compliance with the requirements specification. It does not require any knowledge of inner design (logic and/or code) of the system [9] [10].

- *Beta testing*

The software is released to limited audience, outside of the development team. It is a kind of black-box testing and is performed to ensure that the software has few defects or bugs. It is also known as Ad/hoc or third party testing. [9]

### Non-functional Testing

Non-functional testing means to evaluate the readiness of a system according to several criteria. It allows the measurement and evaluation of the testing of non-functional attributes of software systems.

- *Performance testing*

Performance testing is done to verify and validate systems response, quality and reliability. The system is tested in various scenarios to check its speed and to determine that how much stress or load the system can stand [10]. Power consumption testing is one of the examples, which is one of the important thing in the mobile business.

- *Stability Testing*

Stability testing is done to test the software for crashes. A software ability to work well in unacceptable conditions and/or whether the software can work after the acceptable period. This testing is also known as endurance testing.

- *Usability Testing*

Main reason for usability testing is to test if software user interface is easy to use and understood. User input screens are tested and the outputs, usually printed and on screen reports are tested usually by people who do not know about the software and feedback is collected.



- *Security testing*

Security testing is meant to ensure that confidential and secret data are out of reach by irrelevant system users or by hackers.

- *Internationalization and localization testing*

Software is tested for other regions wherever it will be used. During this testing language grammar, spellings and interfaces are verified and software adaptations for a new culture, currencies and time zones are tested.

## **4.2 Test Automation**

### **4.2.1 Overview**

Software testing, applicable methods and types were discussed in section 4.1. These types of testing can be performed either manually or in automated test environments. However, there are advantages and disadvantages in both. Manual testing is very time consuming and complex and hence expensive as many software developers and testers are needed but a good thing is that the manually written tests are more accurate. However, it does not mean that the performing of the tests are also error free, there can be human errors while inputting data, mistakenly not executing some tests or changing the sequence of tests.

On the other hand, automated tests are also complex in nature but the important part of automation is the system which is used to perform automated testing. That system replaces testers and must have the capability and intelligence to adapt any unusual condition. The cons of the automated systems are time and money saving.

Once enough time and money is spent to construct a test automated systems and they do their job, less people are required to maintain the systems and it almost replaces testers. The test automated systems generate error/feedback reports and testing logs which are frequently sent to developers. The time requires for writing the report and sending it to the developer, manually, is drastically reduced.

Test automation (TA) is the use of software to manage the execution of tests, the comparison of real outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. [12]

The application of software tools and processes to execute tests, for example, tests management, tests design, tests execution and results evaluation, in an automated way, is known as test automation [13]

Therefore, test automation is a process in which software tool is used to perform various tests, mentioned in section 4.1 of this chapter, to save time, increase accuracy and efficiency and reduce cost of development.

Test automation has changed the cost and time versus test coverage and quality trends. After implementing an appropriate and effective test automation system, figure 4.3 below shows how this trend can really be altered.

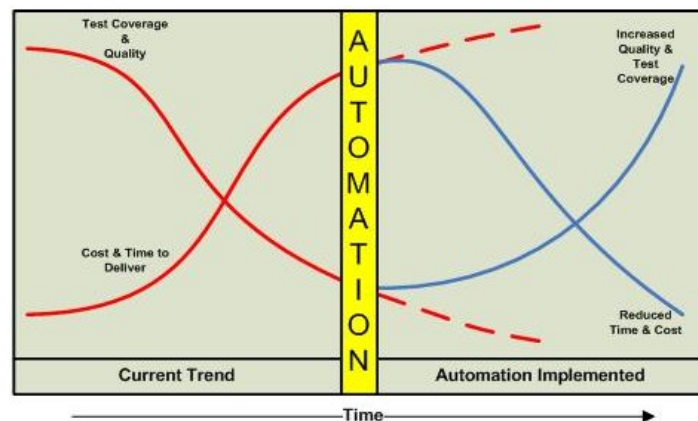


Figure 4.3 Trend showing manual test execution versus test automation [14]

As can be seen in figure 4.3 above that with passage of time test automation can bring a big difference in terms of increase in code coverage and quality with decrease in cost and time.

#### 4.2.2 Test Automation Framework

Later in the paper, in chapters 5 and 7, most of the discussion will be about test automation framework (TAF) so it is important to know the difference between TA and TAF.

TAF is a set of theories, ideas and tools that offer support for robotic software testing, providing that the maintenance cost of the framework is low.

It means that if a correct logic is implemented in a right way to establish an automated test environment then it pays back in terms of cost and efficiency. But it is not direct rule because many factors need to be concerned while planning and designing the system.

Development of TAF is itself a complete software development lifecycle process which includes business cases, requirements management, planning and then implementation using right selection of scripts, tools and procedures. Inside TAF there are several other automated processes which are in a way part of test automation but can be considered as separate processes, for instance, report or document creation process, feedback system and so on.

Therefore, a good planning before establishing a TAF is important and it requires a clear understanding of what has to be automated and why it is being automated as well as what are the expected required outputs and goals of test automation?

#### 4.2.3 Advantages of Test Automation

Automated software testing answers to many questions and provides solution to several problems which software developers are facing today. If test automation is implemented correctly, it has many advantages; some of them are below.

- i. *Execute already existing regression tests*: All regression tests are executed on new version of programs, this is important as many version of same application are frequently modified. A minimum effort of manual configuration, it helps performing series of regression tests automatically on different versions. [15, 9]

- ii. *Execute more tests more frequently.* A clear benefit of automation is the ability to execute more tests in less time and hence to make it possible to execute more frequently. This will lead to greater confidence in the system. [15, 9]
- iii. *Perform tests which would be difficult or impossible to do manually.* For instance, a test requires input from 200 users which seems difficult and costly, which can, otherwise, be simulated using automated tests.

Another example could be a graphical user interface (GUI) object which may trigger some event that does not produce any immediate output. A test execution tool may be able to check that the event has been triggered, which would not be possible to check without using a tool. [15, 9]

- iv. *Better use of resources.* Automating tedious and boring tasks, such as repeatedly inputting same test data, gives greater accuracy as well as improved staff morale, and frees skilled testers to put more effort into designing better test cases. Machines which usually are idle during the nights can be used to execute automated tests. [15, 9]
- v. *Reliability and repeatability of tests.* Tests that are repeated automatically will be repeated exactly every time (at least the inputs will be; the outputs may differ due to timing, for example). This gives a level of consistency to the tests which is very difficult to achieve manually. [15, 9]
- vi. *Reusability.* The effort put into deciding what to test, designing the tests, and building the tests can be distributed over many executions of those tests. Tests which will be reused are worth spending time on to make sure they are reliable because an automated test would be reused several more times than manual testing. [15, 9]
- vii. *Earlier time to market.* [15, 9] Automated testing allows to quickly find and fix bugs and the time elapsed, as compare to manual testing, is much shortened. This helps in releasing qualitative software timely to the market.

- viii. *Increased confidence.* Knowing that extensive sets of automated tests have run successfully, there can be greater assurance that there will not be any horrid shocks when the system is released, providing that the tests being run are good tests. [15, 9]
- ix. *Reducing the time and cost of software testing.* It demands considerable amount of time, money and skills to produce a worth having, reliable and efficient TAF. In return the system becomes more cost effective than the cost was spent to build the system. Efforts and time decreases in terms of test data generation, test execution, test results analysis, monitoring error status and its correction, report creation and other migrating factors. [16]
- x. *Enhanced Software Quality.* Tests execute regularly and in combination of old and newly created tests bring integrity issues which can be solved on earlier stages of development and before integrating new functionality and features. Also, most of the TAFs are capable of executing code coverage algorithms which help identifying useless and unused functions and code which can either be removed or used to bring efficiency to the software.
- xi. *Other improvement areas.* This includes improved – build verification testing, regression testing, multiplatform compatibility and configuration testing, security or memory leak testing (difficult to test manually), light-out and performance testing, stress and endurance testing, workload and concurrency testing, execution of mundane tests, requirements definition, quality measures and test optimization, system development lifecycle and documentation and traceability.

Therefore, more thorough testing can be achieved with less effort, giving increases in both quality and productivity.

#### **4.2.4 Common Problems of Test Automation**

There are also problems that may be encountered in trying to automate testing, which come as a complete surprise, are usually more difficult to deal with but most of them can be overcome. Some of the more common problems with TA are discussed below

- i. *Unrealistic expectations.* Vendors naturally highlight the benefits and successes, and may play down the amount of effort needed to achieve lasting benefits. The effect of

optimism and salesmanship together is to encourage unrealistic expectations. If management expectations are unrealistic, then no matter how well the tool is implemented from a technical point of view, it will not meet expectations [15, 10].

- ii. *Poor testing practice.* Automated testing is not recommended if the testing practices are of poor quality with badly organized tests, little or inconsistent documentation and tests that are not efficient in finding defects [15, 11].
- iii. *False sense of security.* A test suit can run without finding errors, it does not mean that the software has no defects; the tests may be defected and incomplete. [15, 11]
- iv. *Maintenance of automated tests.* Maintaining tests, when software is changed, is often necessary. A test maintenance effort has been death of many test automation systems.
- v. *Technical problems.* It happens often that commercial test automation tools are themselves not very well tested and cause serious problems and require efforts to execute tests. It may take several months before a patch for a fix is available from vendor. [15, 11]

## 5. Symbian Foundation and Build Machinery

### 5.1 Overview

Symbian OS is an operating system (OS) intended for mobile devices and smart-phones, with related libraries, user interface, frameworks and reference implementations of common tools, developed by Symbian Ltd. In 2008, the Symbian Limited was acquired by Nokia and a new independent non-profit organisation called the Symbian Foundation was established [41].

The Symbian Foundation began operating during the first half of 2009 and continues to grow, gaining widespread industry support. Foundation membership is open to any organization and the code is available to all for free [42].

The Symbian Foundation provides, manages and unifies the platform for download and development. On 4th February 2010, the foundation achieved an initial goal of open sourcing the Symbian platform source code. With the code now published under the Eclipse Public License, the entire platform is available to all for free, bringing additional innovation and more frequent and widely-sourced code and feature contributions; and engaging an even broader community in future development [42].

#### 5.1.1 Symbian Build Machinery

Symbian Build Machinery refers to the operations, tools and processes, used by the Symbian Foundation, which can be divided into three categories; physical view, functional view and software build framework.

*Physical view* of the operation is about the types of machines used within the organization and how they are physically configured in terms of hardware, software, operating systems, networks and networks topologies [45].

*Functional view* consists of number of functions and operations required to execute, automate a build and test automation. Figure 5.1 shows functional system overview of the Symbian builds [45].

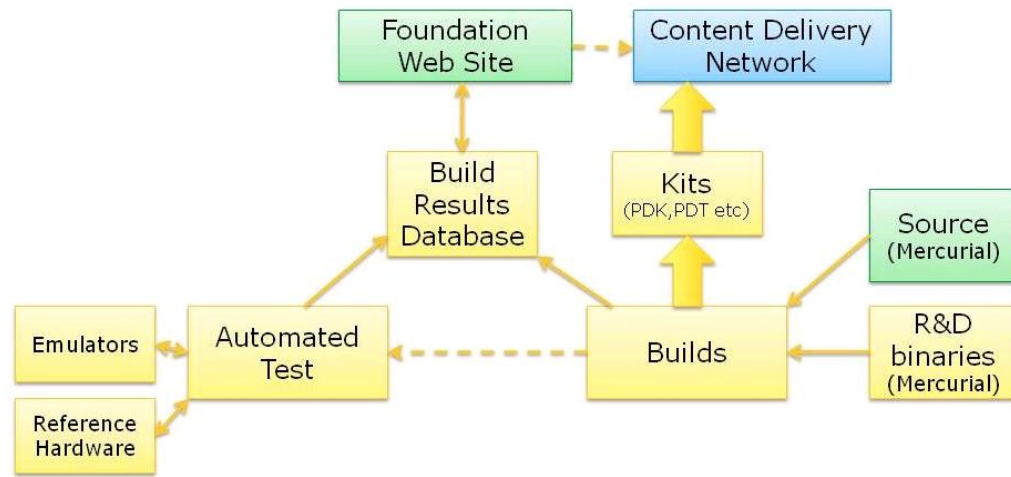


Figure 5.1 Trend showing manual test execution versus test automation [45]

In the above diagram, *Builds* is the point from where the actual build process starts. A full build involves several sequential and parallel processes and Helium Build Framework (see section 2.3.2) is used for accomplishing these processes.

*The Software Build Framework* includes the software, which are running on the build servers and how they are used to support automation of the build and test cycle. In figure 5.2 the first layer is the OS layer on top of that are compilers, for the device, for which the software is being compiled. The make files are bridge between the compilers and the build tools and are used as driver to create executables from the source and PVM is the parallel virtual machine that distributes the Gmake build threads to a cluster. Raptor and ABLD are the build tools which parse the source code and pass the targets to makefiles. On top of that there is Helium build framework which takes care of the complete building process from fetching source code to publish the executables and ROM images and even testing.



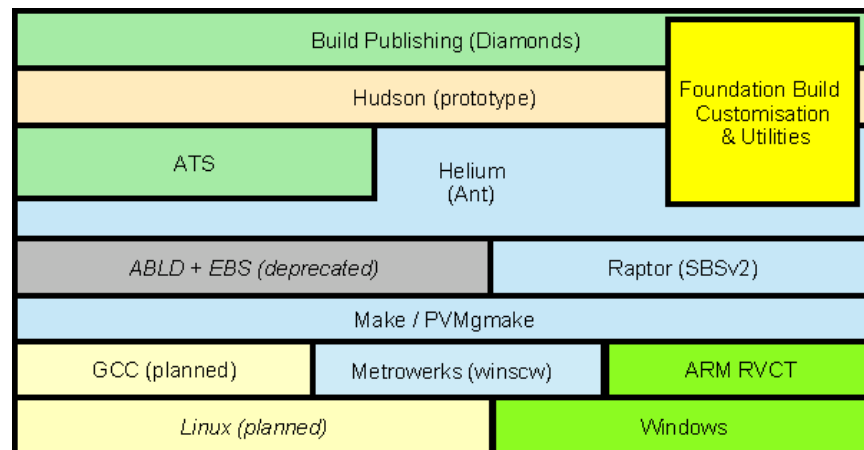


Figure 5.2 Symbian software build framework overview [45]

ATS is a test automation system where devices are connected with clients and tests are executed. All the building process is automated using the build scheduler, Hudson, which acts like a robot and displays ongoing build status and trigger new builds.

## 5.2 Symbian Test Frameworks and Testing Tools

### 5.2.1 Test Frameworks

A test automation framework is a set of assumptions, concepts and tools that provide support for automated software testing. The main advantage of such a framework is the low cost for maintenance. If there is change to any test case then only the test case file needs to be updated and the Driver Script and Startup script will remain the same. There's no need to update the scripts in case of changes to the application [46].

Testing tools, used for program testing to detect faults and errors, are mostly debuggers. However, there are other testing tools which are used to measure software credibility and performance.

Test framework for Symbian build environment can be categorised into two groups, API/unit test framework for development or coding and User Interface (UI) test framework for performance and credibility testing. There are engines for executing

these test frameworks which should be set before executing tests and the engines is typically known as *harness*.

#### API and unit test development frameworks

- STIF – The System Test Integration Framework (STIF) is a standard framework for Symbian platform testing [47]. That is,
  - A test harness for testing Symbian non-UI components
  - A toolkit for test case implementation
  - Supports implementation of test cases in dedicated test modules (DLLs)
  - A test execution environment
  - Not an automatic test case creator but powerfully supports test case implementation
  - Delivered by Symbian
- EUnit – EUnit Pro is a unit, module and integration testing framework for Symbian C++ developers. It helps writing, reusing and execute tests on the code. EUnit Pro follows the general architecture and patterns of xUnit family of unit test frameworks, so the unit testing practise does not differ from JUnit for instance. EUnit Pro is integrated into Carbide.c++ IDE, which enables seamless test creation and execution workflow from the IDE [47].  
 The EUnit Carbide.c++ extension contains wizards for creating EUnit test projects, test suites, stubs for M-class interfaces and path tests. EUnit test suites can be directly executed from Carbide.c++ on emulator and hardware. A commercial license is required for using the framework [47].
- TEF – The Test Execution Framework (TEF) is a test framework similar to STIF. The framework is also delivered by Symbian Foundation [47].

#### UI test framework

- ASTE – UI test automation tool which is primarily used within Nokia and is being gradually replaced by MATTI test framework.  
 In ASTE framework, screenshots of different applications are taken in advanced and are placed in one package. For example, a screen shot of an *Options menu* is taken in advanced and is stored in form of a bitmap (binary image) in one

common place. Later, when a test for the *Options menu* is executed, the test code triggers a command to open the *Options menu* on the device, as part of the test, another screen shot is taken in bitmap format, which is then compared with the previous one. If the comparison matches the test is passed.

- **MATTI** – Another UI test automation tool fully supporting Qt. Not a Symbian Foundation contribution [48]. Qt is a cross-platform application development framework, widely used for the development of GUI programs.

In Matti, UI is tested by matching the attributes of the screen with that of mentioned in the test file. For example, one test is written to check if an *Options menu* has 11 items and the first one is selected, then Matti will compare the test by collecting all the attributes of the screen when the *Options menu* is displayed.

### 5.2.2 Testing Tools

There are mainly two types of testing tools available which are used in collaboration with test frameworks to perform variety of tests on the code and user interfaces.

#### Static Analysis tools

- **CodeScanner** – CodeScanner Pro is a static source code analysis tool for Symbian C++ code. CodeScanner Pro is a command line tool that can integrate into a build system and is delivered by the Symbian Foundation [47].
- **Cyclomatic Complexity** – CMT++ is a tool for analyzing the static complexity, size and maintainability properties of software written in C or C++. Also assembly files can be measured. Tool is developed by Testwell and is available on commercial license [47].
- **Compatibility Analyzer** – Compatibility Analyser is a command line tool that compares a set of files in two different Symbian platform releases against predefined Binary Compatibility (BC) criteria and reports any changes that affect BC. The analysis is done on header files or import libraries of the platform source code and is delivered by Symbian [47].

### Dynamic Analysis tools

- Code Coverage Analysis – CTC++ is a coverage measurement and dynamic analysis tool developed by Testwell. It works for both emulator and hardware targets. The tool is not Symbian specific, so it can be used for coverage analysis of any C/C++ code under Windows/Unix/Linux. It also supports analysis of QT applications; a commercial license is required for the tool [47].
- BCS – The Symbian Binary Compatibility Suite (BCS) is a set of tools for testing the backward binary compatibility of the Symbian platform. The BCS comprises utilities and test cases that are used to verify Binary Compatibility (BC) of the devices based on Symbian C++ Application Programming Interfaces (APIs). BC helps ensure efficient and sustainable development for the Symbian platform and third party application developers. BCS is delivered by Symbian [47].

## **6. Software Development and CI for Symbian**

### **6.1 Agile Software Development Environment**

A few years ago in The Company, there was a mix of almost all the software development methodologies that exist so far. Hundreds of software development teams were producing hundreds and thousands of software components but they were acting as individual entities and were following either their own development practices or they adopted a software development model, some of the development teams used the models as they were and some of them enhanced it to fit to their needs and requirements.

One of the many consequences of working in shells was the slow and unpredictable delivery. When the developed components were integrated, lots of errors were identified and the teams were notified to fix them. It took several weeks before the components were reintegrated and retested.

The Company then adopted agile software development methodologies (discussed in detail in section 2.2.2) in which all the bigger teams were divided into smaller teams consist of 4 to 6 persons each and Scrum (see section 2.2.3) was introduced.

Now a Scrum master, who is mainly responsible for the development activities leads daily scrum meetings where the developers give their daily reports of what they have been doing, what they are planning to do and also discuss any problems they may have with the rest of the team members. This small meeting is very useful to see the growth of the sprint and everyone feels responsible in the team. The development growth also increases since everyday developers have to share their status with everyone else in the team.

The outcome of this change is a frequent, small, complete and consistent product releases, usually fortnightly. All the components are then integrated regularly and errors are notified to the development teams earlier during the development of the whole system. Since every release is based on smaller features, it is easy to identify errors and

fix them rather than a big fully developed component which typically comprises several hundreds of features.

As a result all the teams are aware of their way of working and are more harmonized and focused on the development of the whole system. The teams feel more responsible than before if integration fails and the system comes to a halt.

## **6.2 Symbian Builds and Continuous Integration**

### **6.2.1 Symbian Builds**

Symbian devices are installed with a flash-able ROM image (compiled software file which bundles all the components and core applications). Every software component which is when developed is compiled and tested. During compilation any syntax and logical errors are removed and in the end a working piece of software (also called software component) is produced. Compilation of software components and after that creating a ROM image file is typically known as a build process.

Symbian builds consist of thousands of components which, after receiving from the development teams are integrated and once again the whole build is executed for a particular device. The ready build is then used to produce an image (ROM image) to flash on the device.

The Symbian build process involves multiple stages which are briefly described below, in context to the discussion in chapter 2 to 4. There are many steps involved in a build process, I will describe the seven most typical steps involve in The Company.

- i. Code is checked-in into a version control system by the developers (see section 3.3-d). In our case, The Company is using Mercurial as a version control system (see table 3.5 for the list of version control software).
- ii. Build integration teams then take all the latest checked-in sources from Mercurial for a specific product or device. Taking component one by one may involve several days, therefore some scripting is done to get all the needed components. When the script is executed a work-area is created for a specific project and the components are checked-out. The script only takes those

components which are updated since the last build, which reduces the check-out time.

- iii. As mentioned earlier in chapter 5 that a typical Symbian build consists of several thousands of components and all the components and core applications are compiled every time a full build is executed. And I mentioned above in (ii) that only those components are checked out which were modified after the last build. Therefore, a completed build is always saved on a file server including all the components and only those components are replaced (or updated) which were checked-out.
- iv. Once the build environment is ready, number of build engineers start to compile each and every component. There are many scripts written in different scripting languages e.g. Perl, Python etc. which are responsible to execute the commands and compile the components. Together these scripts are called build tools (see table 3.1 for the characteristics of build tools and table 3.2 for the list of widely used build tools).

The components are mentioned in a list which can be a text file or an xml file. The Company is using xml files in which components are categorized and their names and locations are specified. A build tool uses the list of the components for compilation.

- v. After the compilation, software image is created which is flashed on a specific device for which the image was created, for example, if the build was done for a product X, then the image will only be flash-able on the device X.
- vi. Then testers install (or flash) the device and boot the device. This is the first test of every build whether the device boots up or not. After that a number of tests are performed to verify the components integrity and device stability. All the features and functionalities of the device are tried to get tested but it is not possible to execute tests for each and every component manually, it requires time and manpower to do this tedious, complicated and indeed a costly process.
- vii. Test results are then sent the development teams which take any action if needed to fix the bugs and check-in the source code to Mercurial.

This, a typical, manual build, underwent number of processes before a flash-able image was produced. This involved number of people and an enormous amount of time.

Typically it used to take around two weeks after the code was checked in by developers (i) and to handover the test reports back to them (vii).

The Company then decided to adopt continuous integration practices for the production of builds, together with agile software development process, the scrum.

### **6.2.2 Continuous Integration Builds**

Literature on continuous integration is available in reference sets chapter 3. Application of CI in The Company is discussed here. The first phase of continuous integration was discussed in section 6.1 in which scrum development was introduced to the software development teams and they have started to release components on at least biweekly basis.

Later, build and testing systems were modified by introducing Helium (see section 2.3.2), a continuous integration build framework. Helium requires xml based build configuration file which is called `build.xml` file. In case of complex configuration the xml file can be split into smaller files which can be linked together using standard xml files manipulation techniques. Configuration includes product, Mercurial projects and details of components, as well as file, network, and test servers with credentials.

Using build schedulers (see section 3.6.2) the builds are set to start automatically several times a week and in some cases every day. The projects are continuously checked through the Mercurial and if found changed, the components are taken out and a new build starts.

The environment is taken from a file server for a specified device and the components from Mercurial are replaced in the environment. A components list is re-created and existing binary (previously compiled) files are removed. Using the component list every component is compiled using build tools and compilers (see table 3.2). In case of Symbian, RVCT, ARMV, WINSCW and SBS compilation tools are used.

After the compilation is over ROM image is created for the device. At the end the image is packaged with test components and the package is sent to the test automation server, where the devices are flashed and tests are executed. Test automation will be discussed later in detail, in chapter 7.



Everything that happens throughout the build since its initialization is automatically logged by the CI build framework. These log files are of critical importance to find out reasons in case of build failure and these are also used to analyze performance and other factors to improve quality by eliminating any weaknesses and problems. These log files and test results are sent to the relevant teams and people, who use the information

- to correct errors
- for debugging purposes
- to check stability of the product
- to take decisions for example product and
- to create statistics.

Necessary files, for example product image, logs, test results and other files are updated on network and file servers for other interested people who can re-run the tests on the device using the same image to verify the broken tests and for many other reasons.

### **6.3 Benefits of CI builds Over Manual builds**

The benefits of the CI in general was discussed in section 3.4 but let us discuss if they are proven to be correct after the implementation in The Company.

- After the implementation of CI build system, build times are drastically decreased from a couple of weeks to a few hours
- Productivity increases as errors are detected quickly and regularly and are fixed immediately.
- Product quality enhances as several builds can be setup for one product to check and verify different areas of the build for example every development team can set up a build of their own to get instant feedback.
- Manpower decreases to execute a build. For example, typically manual build was managed and maintained by 10 persons each responsible for each step of the build and then managers on top of group of people. Now, one person can handle one build and
- An amazing drop in the production cost.

Therefore, the CI build contributed in the development and production of the company with cost effectiveness. I also contributed in setting up the CI in The Company. Initially I helped out in the deployment of Helium.

During the deployment I learnt different ways that the build teams were using to execute their own builds. Their way of working was not harmonized with each other which resulted in complications, for example, there were hundreds of scripts in use maintained by individuals responsible for a particular task. When the person leaves the job or in case of absence it was difficult for someone else to fix the issue and let the process continue.

On the basis of these different ways of working and requirements, standardized configuration was created for Helium, which has truly harmonized the builds. It was the first step towards continuous integration.

## **7. Test Automation in Symbian Platform**

### **7.1 Overview**

Thus, software testing is a process to identify errors and discover inappropriate and unacceptable behaviour of software being developed by applying several different testing methods. Software is mainly tested for its integrity, acceptance, performance and security. These tests can be executed manually and can be automated.

Automated testing makes software testing easy and efficient. The tests can be executed frequently and accurately as same test cases are executed every time, which reduces chances of human mistakes and errors. However, writing automated tests and the test automation systems are complex objects and require proper understanding of the test and careful designs.

The focus of this chapter will be on the implementation of software and build test automation in light of The Company.

### **7.2 Automatic Test System (ATS)**

The Automatic Test System (ATS) is an extensible framework and management system for running tests on Symbian devices, including emulators. It automates distributed large-scale testing operations, but can be also used to ease test development for individual test engineers in semi-automated modes. ATS has been developed to enable end-to-end test automation: ATS can flash test images to devices, boot devices, and copy required files to devices, execute defined tests automatically and extract test reports [47].

It can automate tests written for any of the Symbian test execution harnesses (see section 5.3.1), that is, STIF, TEF and EUNIT, as well as other harnesses commonly used by Symbian developers [49].

### 7.2.1 Working of ATS

An ATS network is a distributed system comprising a number of systems. One machine is the ATS server and the rest are ATS clients. Server installation and client installation are supported by the same ATS package: either can be chosen at installation time [49]. The step-by-step working of the system is briefly listed below:

- i. The ATS server receives and despatches requests for tests to be executed on devices. An ATS client controls one or more testable devices that are attached to or resident on the client machine. A device is a set of hardware and/or software resources, such as an emulator, a reference board or a phone, capable of executing tests belonging to one or more of the test harnesses that ATS supports. A device must be registered with the ATS server [49].
- ii. Whenever that client is available, the server will assume that any devices associated with it are also available. Each time a client is started, it notifies the server of its availability and details of the devices it controls [49].
- iii. The ATS client responds to requests from the server by running tests on the devices that it controls and communicating results back to the server. The server can select a client to execute a test request based upon the test harness specified by the test request and the choice of available clients that have devices supporting the specified harness. A test request can also specify the particular device that the server must use. Test requests are queued by the server until they can be despatched [49].
- iv. The ATS server provides a web interface through which administration and user operations can be performed, and also, the status and progress of operations, and the record of past operations, can be viewed [49].
- v. Test requests can be entered manually via the server's web interface, or they can be sent remotely to the server by posting an action URL to its http address. Automated test operations are supported in this way [49].
- vi. A test request directs the server to read and execute a testdrop whose location is given in the test request. This location must be a network location that is readable by the server. A *testdrop* is a zip archive that contains
  - an XML recipe for preparing and performing the required test.

- a (possibly empty) set of files for deployment to the device in preparation for the test. The deployment of these files, if any, is specified by the XML.
- vii. When the server has despatched a test request and it has been executed by a client, a test execution log and a report of the results are retrieved by the server. The log and report can be viewed via the server's web interface, linked to a graphical entry that records the test run. The detailed output of the test harness is also stored by the server on its local file-system in a directory created for the test run [49].

### 7.3 Role of Helium in the Symbian Test Automation Process

A testdrop is a zip file which typically includes an XML file, ROM images (to be flashed on a device), Dynamic Linked Libraries (DLLs) and/or executables (EXEs) to be tested. The testdrop is sent to the ATS server to be unzipped and the files it contains are installed on a specific device, and tests are executed.

Helium basically creates the testdrop package. It is not an easy and straightforward process. It involves number of logical operations for choosing the right set of files, for example, ROM images to be flashed on the phone, DLLs and/or EXEs and their dependencies, test harness to select right engine for test execution and test data files: their source (in the build area) and destination (on the devices) are identified first. Then based on this information, an XML file is generated which follows a predefined structure. All this happens after the compilation and image creation stages, as discussed in section 6.2.2. Most of the logic, design and implementation of this process was done by me and is discussed in the following sections.

#### 7.3.1 Directories and Files needed for a testdrop creation

Tests are written by test developers as part of the software development process to test the code and functionality of the software (see section 4.1). A single unit of the test set is known as *test component*. A test component comprises of sub directories and files. The files include necessary data which is parsed by Helium scripts to create a testdrop.

### Directory and file structure of a test component

The tests are included in the Symbian build environment with the rest of the code.

Figure 7.1 below, represents a typical structure of a test component.

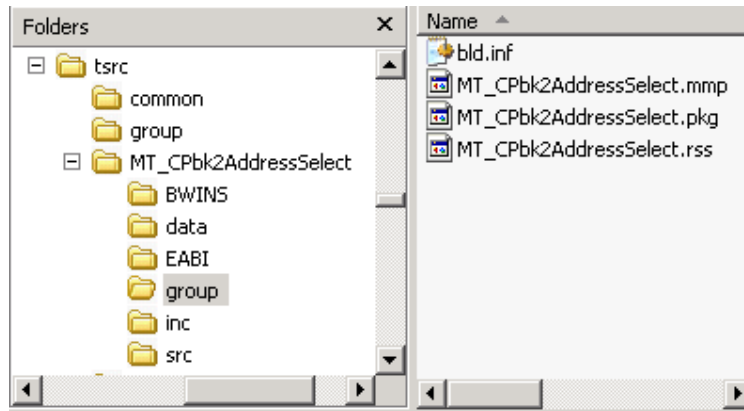


Figure 7.1 A Test component directory structure

The folders section in figure 7.1 shows that *tsrc* is a directory where the tests components are resided. For every component like the one *MT\_CPb2AddresSelect* which is being considered as an example, there must be at least one directory called *group* under it. All other directories are optional.

The *group* directory must contain at least two files, that is, *bld.inf* and a *MT\_CPb2AddresSelect.mmp*<sup>(\*)</sup>, if only a DLL or EXE is required to be installed and tested, without any supportive data files to push into the device. In case there are other files needed for the test execution those files can be placed anywhere inside the test component and there must be a *MT\_CPb2AddresSelect.pkg*<sup>(\*)</sup> file. See Appendix A for contents and description of the files (*bld.inf*, *MT\_CPb2AddresSelect.mmp* and *MT\_CPb2AddresSelect.pkg*).

<sup>(\*)</sup> It is not mandatory to have filename after a component's name

### Directory and file structure of Helium Scripts

The whole Helium framework is required in order to execute even only the testing part. The testing part comprises of several scripts mostly written in Python, ANT and Java.

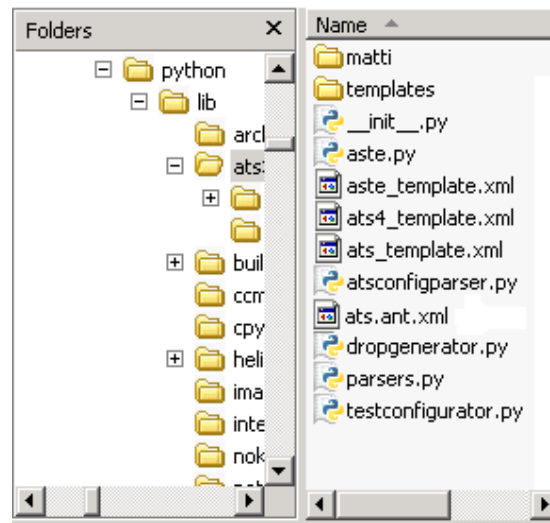


Figure 7.2 Helium scripts responsible for creating a test drop

In figure 7.2, *ats.ant.xml* is the file which has the ANT target (see table 3.2) *ats-test* which invokes the process of testdrop creation.

### 7.3.2 Generation of *testdrop* for API and Unit tests

In a step by step manner, testdrop creation is discussed below for API and Unit tests which were discussed in section 5.2.1.

#### Step 1: Component Selection

When *ats-test* target is initiated, it first checks for component(s) to be packaged in a *package\_definition.xml* file [50]. The file contains a list of components to be built and tested. According to the *package\_definition.xml* file version 1.3 and version 1.4 (to see the file for the component which is being discussed here, see the Appendix B – a) the template of the component definition looks like:

```
<layer name="name_test_layer">
  <module name="module_name_one">
    <unit unitID="unit_id1" name="unit_name1"
      bldFile="path_of_tsrc_folder_to_be_built" mrp="" />
  </module>
</layer>
```

The paths of the selected test components are then sent to a file *\_\_init\_\_.py* shown in figure 7.2.

### Step 2: Setting of test-specific properties

Along with the list of paths of the test components, some properties (configuration) are also plugged into the script, which are used to create `test.xml` file (See appendix C). The parameters are defined in the `ats.ant.xml` file and are listed below in table 7.1.

Table 7.1 ATS specific parameters and their description for 'testdrop' creation

Parameters	Description
<code>ats.product.name</code>	Name of the product to be tested. For example: "PRODUCT".
<code>eunitexerunner.flags</code>	Flags for EUnit exerunner can be set by setting the value of this variable. The default flags are set to "/E S60AppEnv /R Off".
<code>ats.email.list</code>	The property is needed to receive an email from ATS server after the tests are executed. There can be one to many semicolon(s) ";" separated email addresses.
<code>ats.flashfiles.minlimit</code>	Limit of minimum number of flash files to execute ats-test target, otherwise ATSDrop.zip will not be generated. Default value is "2" files.
<code>ats.plan.name</code>	Modify the plan name if you have understanding of test.xml file or leave it as it is. Default value is "plan".
<code>ats.product.hwid</code>	Product HardWare ID (HWID) attached to ATS. By default the value of HWID is not set.
<code>ats.script.type</code>	There are two types of ats script files to send drop to ATS server, "runx" and "import"; only difference is that with "import" ATS doesn't have to have access rights to testdrop.zip file, as it is sent to the system over http and import doesn't need network shares. If that is not needed



	“import” should not be used. Default value is “runx” as “import” involves heavy processing on ATS server.
<code>ats.target.platform</code>	Sets target platform for compiling test components. Default value is “armv5 urel”.
<code>ats.test.timeout</code>	To set test commands execution time limit on ATS server, in seconds. Default value is “60”.
<code>ats.testrun.name</code>	Modify the test-run name if really needed otherwise leave it as it is. Default value is the name of the module defined in the package_definition.xml file
<code>ats.trace.enabled</code>	Should be “True” if tracing is needed during the tests running on ATS. Default value is “False”.
<code>ats.ctc.enabled</code>	Should be “True” if coverage measurement and dynamic analysis (CTC) tool support is to be used by ATS. Default value is “False”.
<code>ats.ctc.host</code>	CTC host, provided by CATS used to create coverage measurement reports. MON.sym files are copied to this location, for example “10.0.0.1”. If not given, code coverage reports are not created
<code>ats.obey.pkgfiles.rule</code>	If the property is set to “True”, then the only test components which will have PKG files, will be included into the test.xml as a test-set. Which means, even if there’s a test component (executable) but there’s no PKG file, it should not be considered as a test component and hence not included into the test.xml as a separate test. By default the property value is False.
<code>reference.ats.flash.images</code>	Fileset for list of flash images (can be .fpx, .C00, .V01 etc) It is recommended to set the fileset, default filset is given below which can be overwritten. set

	<i>dir=</i> ”< <i>something</i> >“ attribute of the filset to “\${build.output.dir}/variant_images” if “variant-image- creation” target is being used.
<code>ats.report.location</code>	Sets ATS reports store location. Default location is “\${publish.dir}/\${publish.subdir}”.

The values for the parameters listed in table 7.1 are defined in a configuration file which is known as `build.xml` file (see Appendix B – b). These parameters are used by the scripts, and a python dictionary which is used to generate the `test.xml` file is created.

### Step 3: Test component files’ parsing

The `__init__.py` script passes the execution control to the `testconfigurator.py` and the `parsers.py` scripts which start parsing the components directories and files in order to fetch test related data and information to be used to select test engine (harness), test type, image files, supporting data files, executables or binaries. See Appendix”A” for the description of files and data fetched.

The process performs several numbers of sequential and hierarchical conditional and logical checks before it finalizes data structure. This part can be considered as backbone of the entire testdrop creation process and is very complex.

The end result of this stage is a well structured nested python dictionary which includes each and every detail required to produce `test.xml` file.

### Step 4: XML file and ‘testdrop’ formation

The gathered data is written to a predefined XML file format (see Appendix C for an example `test.xml` file and a brief description). The `test.xml` file, ROM images, dll file and the data file are collected and a .zip file is created, which is called a ‘testdrop’.

### Step 5: Sending of ‘testdrop’ to ATS server

As the final step, Helium sends the ‘testdrop’ package file to the ATS server. This step requires a few more configuration properties to notify Helium about the server location and server credentials.

Table 7.2 Properties required sending the ‘testdrop’ to the ATS server

Parameters	Description
<code>ats.server</code>	For example: “4fix012345” or “atssrv001.atsserver.net:80”. Default server port is “8080”. The host can be different depending on site and/or product.
<code>ats.drop.location</code>	Server location (UNC path) to save the ATSDrop file, before sending to the ATS Server. For example: <code>\\trwsem00\some_folder\</code> . In case, <code>ats.script.type</code> is set to “import”, ATS doesn’t need to have access to <code>ats.drop.location</code> , its value can be any local folder on build machine, for example <code>c:/temp</code> (no network share needed).
<code>ats.username</code>	A user name required to login to the ATS server. The user ID is created by registering on the ATS server web interface. The registration requires an administrator approval.
<code>ats.password</code>	The password is required during the login process. This password is chosen by a user during the registration process.

In the end a connection with an ATS server is established and the testdrop is sent to the server.

### 7.3.3 Generation of *testdrop* for UI tests

For the User Interface framework, discussed which was discussed in section 5.2.1, the ‘testdrop’ creation is almost similar. Instead of `ats-test` target, `ats-matti` or `ats-aste` targets are called.

The targets get the packages and data files from specified locations from the configuration files, and pass the parameters to the scripts which are responsible for creating a 'testdrop'. In the case of UI testdrop creation, most of the parameters are same as were in API and Unit tests but with a few differences. The testdrop is then sent to the ATS server where the tests are executed.

### **7.3.4 ATS Server Executes the Test and Generates Reports**

Helium sends the testdrop packages to the ATS server, where they are queued for execution if there are no available devices for those tests. Once the devices, attached to clients ATS machines are freed, the test drop is unpacked and test.xml file is read. There are various commands in the test.xml file which are executed one by one (see Appendix C for test.xml file).

First the image files are flashed into the phone and the phone is booted up. If the phone successfully boots up, one part of automatic (or default) test is completed and then the tests in the file are executed. For a set in the file, DLLs, Executables, data files and/or any other files are installed on the device, separate install steps are needed in test.xml file for the installation of the files and the files have to be in the correct location in the package as pointed by the step in test.xml.

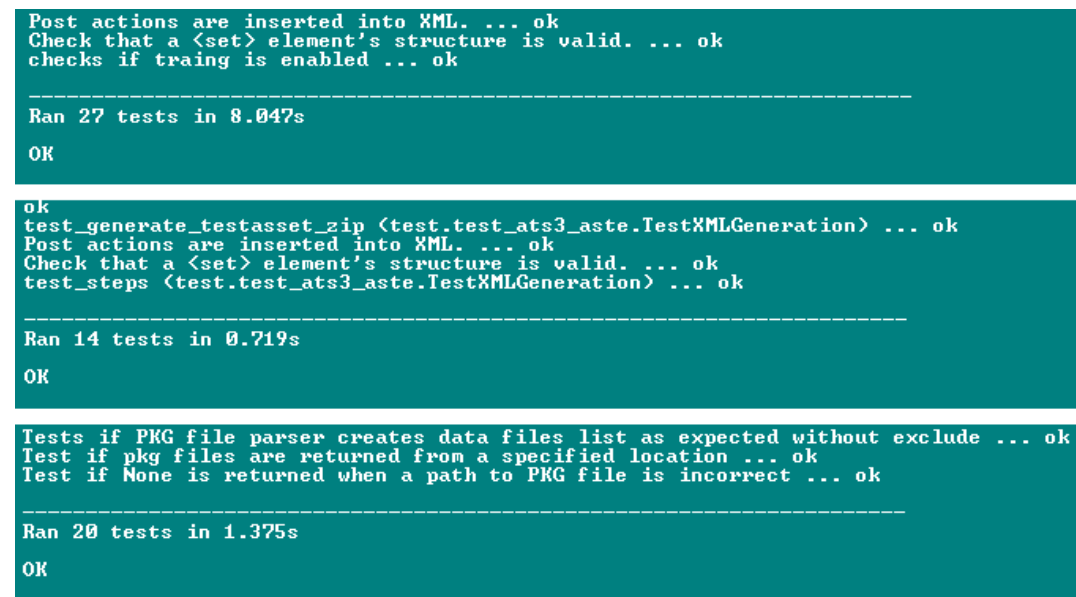
Then other commands are executed for every test, which include, commands to create log files, commands to execute test DLLs and/or EXEs, commands to perform different tasks and at last commands to fetch log files and generate test log files are executed. The reports are sent to the email address(es) provided in the test.xml file and also are published on different file and web servers.

The working of ATS servers and clients has been also discussed in section 7.1.1 in detail. By the time this part is being written, the final decision, for making ATS publicly available, had to be taken. So the reports and emails which ATS generates cannot be shared for confidentiality reasons. However, the reports have details of the tests components which were tested, date and time stamps of the execution and whatever the commands from the test.xml file were executed is shown in the report. The pass and

failure of the commands are clearly visible in different colours with a short explanation of the reason of failure.

### 7.3.5 Acceptance Tests for the Helium Scripts

More than 50 tests were written to check the integrity and acceptance of the Helium scripts itself and for the process of creating `test.xml` file. Those tests are executed every time any script is updated. Also new tests are added continuously.



```

Post actions are inserted into XML. ... ok
Check that a <set> element's structure is valid. ... ok
checks if traing is enabled ... ok
-----
Ran 27 tests in 8.047s
OK

ok
test_generate_testasset_zip <test.test_ats3_aste.TestXMLGeneration> ... ok
Post actions are inserted into XML. ... ok
Check that a <set> element's structure is valid. ... ok
test_steps <test.test_ats3_aste.TestXMLGeneration> ... ok
-----
Ran 14 tests in 0.719s
OK

Tests if PKG file parser creates data files list as expected without exclude ... ok
Test if pkg files are returned from a specified location ... ok
Test if None is returned when a path to PKG file is incorrect ... ok
-----
Ran 20 tests in 1.375s
OK

```

Figure 7.3 Screenshots of the unit and acceptance tests written for the Helium scripts

Figure 7.3 shows the total number of tests executed and an “OK” at the end shows that all the acceptance tests of the Helium code, which is responsible of creation of testdrop, passed successfully. The tests help discovering any logical or syntactical error in creation of the testdrop and/or `test.xml` files, any bug in program logic itself or any unusual happenings.

## 7.4 Survey on Continuous Integration and Test Automation:

One year after the implementation of the CI system, *The Company* together with VTT Technical research centre of Finland, conducted a survey on continuous integration to

check its affects on the development and production betterment? And whether the implementation of CI within the company has proven fruitful? A few selected questions and their responses from the survey are presented in Appendix D. A few questions from the questionnaire (see Appendix D – *question groups C, F and G*) and their responses are being discussed here as conclusion of my Thesis.

– *C. General assumptions about CI's benefits, weaknesses, and challenges*

This part covers the questions which were asked by different groups of people from development, production and quality assurance teams. The results are displayed in the form of graphs. Four different questions were asked with a scale ranging from 0 to 100 which shows level of disagreement to the level of agreement, respectively.

There are also two graphs for every question; the one on the left shows the response from everyone who participated in the survey whereas the graph on the right shows responses from three specific roles, that is, developers, test engineers and the project managers.

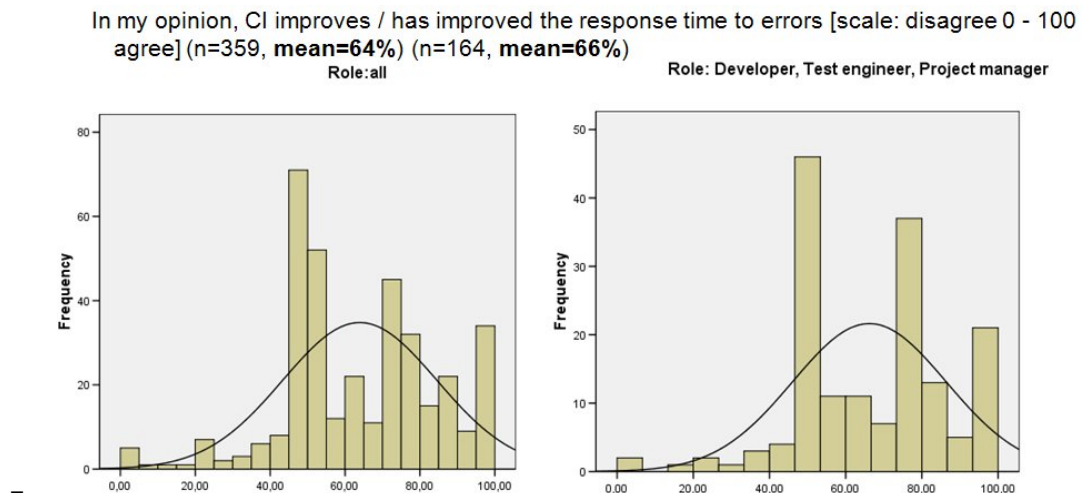


Figure 7.4: Graph showing improvement in response time to errors

Figure 7.4 shows that an average 64% of general people and 66% of people having specific roles think that CI has improved the response time to errors, that is, it is now faster to receive an error report and hence faster to fix.

In my opinion, CI improves / has improved features' quality [scale: disagree 0 - 100 agree]  
 (n=355, **mean=62%**)(n=163, **mean=64%**)  
 Role: all Role: Developer, Test engineer, Project manager

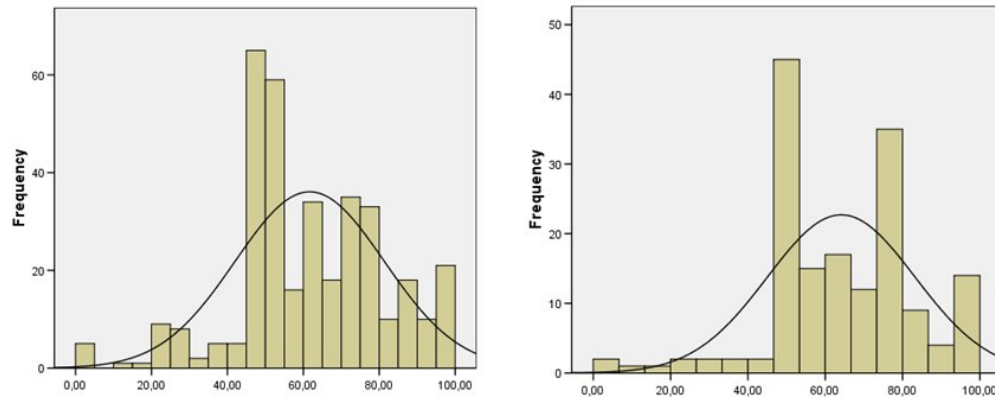


Figure 7.5: Graph showing improvement in features' quality

Figure 7.5 shows that an average 62% of general people and 64% of people having specific roles think that CI has helped improving the quality of features. This is because of finding errors at an early stage, through test automation process which has brought consistency in frequent test executions.

In my opinion, CI improves / has improved my team's efficiency [scale: disagree 0 - 100 agree] (n=338, **mean=60%**) (n=156, **mean=62%**)  
 Role: all Role: Developer, Test engineer, Project manager

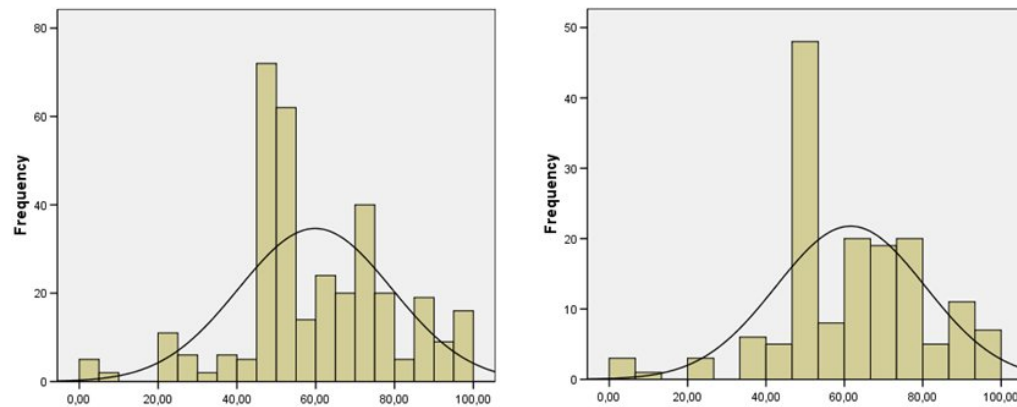


Figure 7.6: Graph showing improvement in efficiency of teams

Figure 7.6 shows that an average 60% of general people and 62% of people having specific roles think that CI has improved efficiencies of team work. This is certainly because of automation processes which have contributed their roles in getting things done faster and with regular pace.

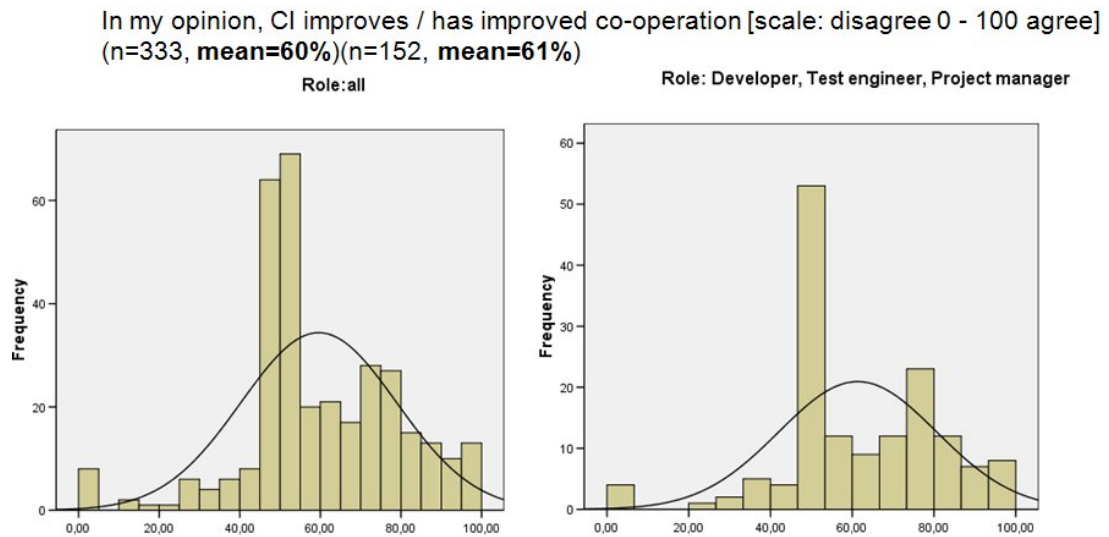


Figure 7.7: Graph showing improvement in communication and cooperation

Figure 7.7 shows that an average 60% of general people and 61% of people having specific roles think that CI has improved communication and coordination between the teams. This is due to the fact that visibility of processes has increased which helps knowledge sharing and enhances communication. Another role of CI in the development area is pair programming, which has increased same level of understanding between developers and close cooperation.

#### – F. Opinion about CI adoption

In this section of the questionnaire people were asked to write their comments in response to the questions asked below in F.1 through F.3.

F.1 In your opinion, how individual work habits have changed after CI adoption?

- Improved agility, quality, productivity and communication
- Increased efficiency
- Fastened development
- Attitude towards testing is getting better

F.2 In your opinion, how does CI support personal working style?

- Improved feedback, visibility, communication
- Early detection of error
- Build efficiency has increased through build automation



F.3 Please describe the benefits of CI adoption at The Company?

- Improves (productivity, visibility and transparency, time-to-market, communication, resource time and money)
- Reduces (Manual builds, testing costs and delivery slippages)

– G. Summary

Section G of the questionnaire is also based on the comments of the participants who were asked to write their opinions about the change in the working habits after the adoption of CI.

G.1 In your opinion, how individual work habits have changed after CI adoption?

- Good move to improve software quality
- Shortened development time
- Helpful; benefits can be seen
- More pressure on build teams

The rest of the survey on continuous integration and test automation with results included in a separate column along with the questions can be seen in Appendix D.

## 8. Conclusion

Continuous integration and the test automation processes have helped *The Company* in several different ways, to get development done faster, efficiently and with lower development and production costs. The process of transformation took several months or even a few years before a full and steady CI and test automation processes were in place.

New ways of working, renewal of the test code structure, new standards for the coding conventions and many other changes made it difficult for software developers to get used to these; but once they and many others, who were directly or indirectly concerned, had gone through these changes, benefits at last are becoming visible.

For continuous integration, the benefits have proven to be true that the defects, broken code, incompatibility, integration conflicts are all detected and fixed sooner rather than later in the development, because in the CI, the software builds are run and the code and functionality is tested at least once a day. CI builds also help leadership and managers forecasting their plans. Software quality is measurable and the product release date is predictable hence, less chances of delay which was often the case earlier.

Hence, the target of this project, to find answers to the questions, from continuous integration perspective of adopting and using combination of different tools and processes to enhance development and production, has now been answered. Build automation and test automation played a vital role in achieving this objective.

## References

1. Software Test, Q&A Support in the 2.0 World – (Testing Models). URL: <http://geekswithblogs.net/dthakur/articles/6475.aspx>. Accessed: December 29, 2009.
2. Software Testing, Testing Tutorial. [online]  
URL: [http://www.etestinghub.com/testing\\_models.php](http://www.etestinghub.com/testing_models.php). Accessed December 29, 2009 .
3. Wikipedia: Waterfall Model (Diagram file). [online]  
URL: [http://en.wikipedia.org/wiki/File:Waterfall\\_model.png](http://en.wikipedia.org/wiki/File:Waterfall_model.png). Accessed December 29, 2009.
4. Wikipedia: V-Model (Diagram file). [online]  
URL: <http://en.wikipedia.org/wiki/File:V-model.JPG>. Accessed December 29, 2009.
5. ETestingHub – Online Software Testing Tutorial. Spiral Model. [online]  
URL: <http://www.etestinghub.com/spiral.php>. Accessed December 29, 2009.
6. Myers Glenford. Revised and updated by Tom Badgett, Todd M. Thomas and Corey Sandler. The Art of Software Testing. 2<sup>nd</sup> ed. John Wiley & Sons Inc., Hoboken, New Jersey; 2004.
7. Bach James. "Risk and Requirements-Based Testing" (PDF); June 1999 [online]  
URL: [http://www.satisfice.com/articles/requirements\\_based\\_testing.pdf](http://www.satisfice.com/articles/requirements_based_testing.pdf).  
Accessed December 29, 2009.
8. Savenkov Roman. "How to Become a Software Tester". Roman Savenkov Consulting, 1<sup>st</sup> ed.; 2008.
9. Software Engineering – Testing. [online]  
URL: <http://www.openseminar.org/se/modules/7/index/screen.do>.  
Accessed December 30, 2009.
10. ApTest (software testing specialists) – Types of Software Testing. [online]  
URL: <http://www.aptest.com/testtypes.html>. Accessed December 30, 2009.
11. Mrlnik Grigori, Meszaros Gerard, Bach Jon. Acceptance Test Engineering, Volume I: Thinking. Microsoft Corporation; 2009.
12. Kolawa, Adam; Huizinga, Dorota. Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press; 2007 [online]  
URL: <http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>.  
Accessed January 06, 2010.
13. Glossary – Thiagarajan Veluchamy's Blog ('T' for Test Automation) [online]  
URL: <http://thiyagarajan.wordpress.com/glossary/>. Accessed January 06, 2010.
14. Metrics for Implementing Automated Software Testing. [online]  
URL: <http://www.methodsandtools.com/archive/archive.php?id=94>.  
Accessed January 06, 2010.
15. Fewster Mark, Graham Dorothy. Software Test automation Aontinuous – Effective use of test execution tools. ACM Press, Addison Wesley; 1999.

16. Dustin Elfriede, Garrett Thom, Gauf Bernie. Implementing Automated Software Testing. Pearson Education Inc.; 2009.
17. Sommerville Ian. Software Engineering. 6th Edition, Pearson Education Limited, Addison Wesley; 2001.
18. Aptest, software testing glossary. [online]  
URL: <http://www.aptest.com/glossary.html>. Accessed December 16, 2009.
19. Meranetworks software glossary. [online]  
URL: [www.meranetworks.com/fundamentals/glossary](http://www.meranetworks.com/fundamentals/glossary). Accessed December 16, 2009.
20. Fowler Martin. Continuous Integration. [online]  
URL: <http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration>. Accessed December 11, 2009.
21. Duvall Paul, Matyas Steve, Glover Andrew. Continuous Integration – Improving software quality and reducing risk. Upper Saddle River, Addison Wesley; 2008.
22. DSL - A Specification Language Perspective – (Make). [online]  
URL: [http://phoenix.labri.fr/wiki/doku.php?id=an\\_overview\\_on\\_dsls](http://phoenix.labri.fr/wiki/doku.php?id=an_overview_on_dsls).  
Accessed December 13, 2009.
23. Apache Maven Project –Maven (Introduction to Build Lifecycle). [online]  
URL: [http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle\\_Reference](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference). Accessed December 13, 2009.
24. About Rake – Ruby Make. [online]  
URL: <http://rake.rubyforge.org/>. Accessed December 13, 2009.
25. Groovy – An agile dynamic language for Java platform. [online]  
URL: <http://groovy.codehaus.org/>. Accessed December 13, 2009.
26. About Apache Ant. [online]  
URL: <http://ant.apache.org/index.html>. Accessed December 13, 2009.
27. IBM – Rational Build Forge Enterprise Edition. [online]  
URL: <http://www-01.ibm.com/software/awdtools/buildforge/enterprise/>.  
Accessed December 13, 2009.
28. Hudson Wiki – Meet Hudson (What is Hudson). [online]  
URL: <http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson>.  
Accessed December 13, 2009.
29. BuildBot – (Manual 0.7.11). [online]  
URL: <http://djmitche.github.com/buildbot/docs/0.7.11/#Introduction>.  
Accessed December 13, 2009.
30. Astels David. Forewords by Jeffries Ron. Test Driven Development: A Practical Guide. Pearson Education Inc., Upper Saddle River, New Jersey; 2003.

31. Royce Winston W. Managing the development of large software system. [PDF]; 2003  
URL: <http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf>.  
Accessed February 12, 2010.
32. Agile Software Development. [online]  
URL: <http://agilemanifesto.org/history.html>. Accessed February 12, 2010.
33. Agile In a Flash – 12 Principles for Agile Software Development. By: Jeff Langar and Tim Ottinger. URL: <http://agileinaflash.blogspot.com/2009/08/12-principles-for-agile-software.html>. Accessed 12.02.2010.
34. Extreme Programming Rules. [online]  
URL: <http://www.extremeprogramming.org/rules.html>. Accessed February 14, 2010.
35. Build and Release Management – Electric Cloud, ElectricCommander and ElectricAccelerator. [online]  
URL: <http://www.electric-cloud.com/products/>. Accessed February 14, 2010.
36. Wikipedia: Spiral Model, Boehm; 1988. [online]  
URL: [http://en.wikipedia.org/wiki/File:Spiral\\_model\\_\(Boehm,\\_1988\).png](http://en.wikipedia.org/wiki/File:Spiral_model_(Boehm,_1988).png).  
Accessed February 12, 2010.
37. Agile Software Development Methodology. [online]  
URL: <http://www.celusion.com/Services/SoftwareDevelopment/tabid/193/Default.aspx>.  
Accessed March 3, 2010.
38. Scrum Overview. [online]  
URL: <http://consultingblogs.emc.com/ColinBird/>. Accessed March 7, 2010.
39. Extreme programming process model. [online]  
URL: <http://www.esic-solutions.com/services.html>. Accessed March 7, 2010
40. Test Driven Development. [online]  
URL: <http://www.softwaretestingprocess.com/testmethod/tdd.html>.  
Accessed March 7, 2010.
41. Symbian OS. [online]  
URL: [http://en.wikipedia.org/wiki/Symbian\\_OS](http://en.wikipedia.org/wiki/Symbian_OS). Accessed March 9, 2010.
42. About Symbian Foundation. [online]  
URL: <http://www.symbian.org/about-us>. Accessed March 9, 2010.
43. SBSv2 or Raptor Build System. [online]  
URL: [http://developer.symbian.org/wiki/index.php/Raptor\\_Build\\_System](http://developer.symbian.org/wiki/index.php/Raptor_Build_System).  
Accessed March 9, 2010.
44. Helium Build Framework. [online]  
URL: [http://developer.symbian.org/wiki/index.php/Build\\_machinery#Helium\\_.2F\\_Foundation\\_Build\\_Framework](http://developer.symbian.org/wiki/index.php/Build_machinery#Helium_.2F_Foundation_Build_Framework). Accessed March 9, 2010.
45. Symbian Foundation Build Machinery. [online]  
URL: [http://developer.symbian.org/wiki/index.php/Build\\_machinery#Physical\\_View](http://developer.symbian.org/wiki/index.php/Build_machinery#Physical_View).  
Accessed March 9, 2010.

46. Test Automation Framework. [online]  
URL: [http://en.wikipedia.org/wiki/Test\\_automation\\_framework](http://en.wikipedia.org/wiki/Test_automation_framework). Accessed March 9, 2010.
47. Symbian Test Frameworks and testing tools. [online]  
URL: [http://developer.symbian.org/wiki/index.php/Symbian\\_Test\\_Tools](http://developer.symbian.org/wiki/index.php/Symbian_Test_Tools).  
Accessed March 9, 2010.
48. MATTI User interface test framework. [PDF]  
URL: [http://developer.symbian.org/wiki/images/a/a1/SIG09\\_SF\\_Test\\_Tools\\_Packages.pdf](http://developer.symbian.org/wiki/images/a/a1/SIG09_SF_Test_Tools_Packages.pdf).  
Accessed March 9, 2010.
49. ATS Test Automation System – *How to guide*. [online]  
URL: [http://developer.symbian.org/wiki/index.php/How\\_ATS\\_is\\_used\\_in\\_Symbian](http://developer.symbian.org/wiki/index.php/How_ATS_is_used_in_Symbian).  
Accessed March 9, 2010.
50. Symbian Developer Community – ATS test generator Helium integration. [online]  
URL: [http://developer.symbian.org/wiki/index.php/ATS\\_Test\\_Generator\\_Helium\\_Integration](http://developer.symbian.org/wiki/index.php/ATS_Test_Generator_Helium_Integration). Accessed March 7, 2010.
51. Cost of change curve for agile software development. [online]  
URL: <http://www.agilemodeling.com/essays/costOfChange.htm>. Accessed March 15, 2010.
52. Competition quotes. [online]  
URL: <http://www.worldofquotes.com/topic/competition/index.html>.  
Accessed March 15, 2010.
53. Dilbert on extreme programming. [online]  
URL: <http://weblog.cemper.com/a/200312/08-dilbert-on-extreme-programming.php>.  
Accessed March 28, 2010.

## Appendix A – *MT\_CPbk2AddressSelect* Test Component's Files

*MT\_CPbk2AddressSelect* test component was used for the creation of a testdrop (AtsDrop.zip) package, as an example for the thesis. This component was provided by Nokia to Symbian Foundation and is an open source code for the public.

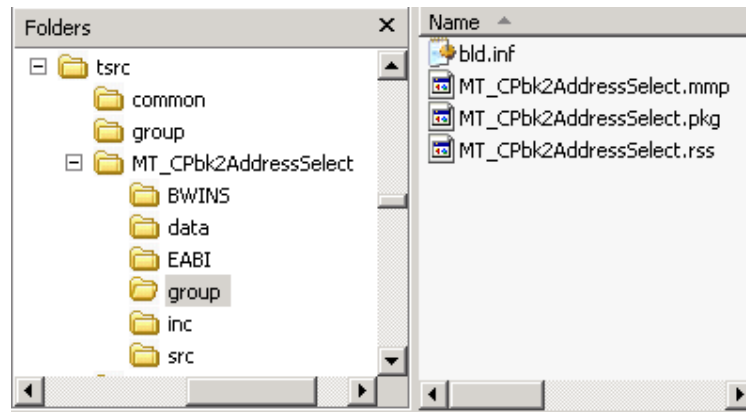


Figure A.1: Component "*MT\_CPbk2AddressSelect*" directory and file structure

### 1. *bld.inf* file:

```
/* Copyright (c) 2002 Nokia Corporation and/or its subsidiary(-ies).
 * All rights reserved.
 * This component and the accompanying materials are made available
 * under the terms of "Eclipse Public License v1.0"
 * which accompanies this distribution, and is available
 * at the URL "http://www.eclipse.org/legal/epl-v10.html".
 *
 * Initial Contributors:
 * Nokia Corporation - initial contribution.
 * Contributors:
 * Description:
 */
PRJ_PLATFORMS
DEFAULT

PRJ_TESTMMPFILES
MT_CPbk2AddressSelect.mmp
// End of File
```

### Description

For the test automation point of view, PRJ\_TESTMMPFILES project directive is a point of interest, under which, the mmp file is mentioned. It contains the details of the test components. There can be more than one .mmp files, bld.inf files and/or mixture of both the file(s) included and the hierarchy of multiple components is parsed by C pre-processor (CPP).

### 2. *MT\_CPbk2AddressSelect.mmp* file:

```
/*
 * Copyright (c) 2002 Nokia Corporation and/or its subsidiary(-ies).
 * All rights reserved.
```

```

* This component and the accompanying materials are made available
* under the terms of "Eclipse Public License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.eclipse.org/legal/epl-v10.html".
*
* Initial Contributors:
* Nokia Corporation - initial contribution.
*
* Contributors:
*
* Description:
*
*/

#include <platform_paths.hrh>
#include <data_caging_paths.hrh>

TARGET          MT_CPbk2AddressSelect.dll
TARGETTYPE      dll
CAPABILITY      ALL -TCB
TARGETPATH      RESOURCE_FILES_DIR
UID             0x1000af5a 0x01700000

RESOURCE        MT_CPbk2AddressSelect.rss

SOURCEPATH      ../src
SOURCE          MT_CPbk2AddressSelectDllMain.cpp
SOURCE          MT_CPbk2AddressSelect.cpp
SOURCE          ../../common/CSimulateKeyBase.cpp
SOURCE          ../../common/CSimulateKeyEvents.cpp
SOURCE          ../../common/CPbk2TestSuiteBase.cpp

USERINCLUDE     ../inc
USERINCLUDE     ../../common

SYSTEMINCLUDE   ../../../../phonebook2/inc
APP_LAYER_SYSTEMINCLUDE
SYSTEMINCLUDE   /epoc32/include/Digia/EUnit

LIBRARY         EUnit.lib EUnitUtil.lib
LIBRARY         VPbkEng.lib Pbk2Presentation.lib Pbk2UiControls.lib
LIBRARY         euser.lib eikcoctl.lib eikctl.lib eikcore.lib baf1.lib
LIBRARY         AknSkins.lib avkon.lib
LIBRARY         cone.lib CdlEngine.lib commonengine.lib cntmodel.lib
LIBRARY         efsrv.lib ecom.lib

// End of File

```

### Description

Among this information, a few keywords are related to test automation, for example:

- TARGET specifies a filename of the file which should be created after the compilation of the component.
- TARGETTYPE specifies about the type of the file either DLL or EXE. In this case it is a DLL
- LIBRARY includes information about associated libraries attached to the component. In most cases the harness of the component is selected if certain library files are mentioned. In this case there is a library EUnit.lib tells about the harness of the test which is EUnit. (*harness was discussed in section 7.1, paragraph 2*).



### 3. *MT\_CPbk2AddressSelect.pkg* file:

```
; Copyright (c) 2009 Nokia Corporation and/or its subsidiary(-ies).
; All rights reserved.
; This component and the accompanying materials are made available
; under the terms of "Eclipse Public License v1.0"
; which accompanies this distribution, and is available
; at the URL "http://www.eclipse.org/legal/epl-v10.html".
;
; Initial Contributors:
; Nokia Corporation - initial contribution.
;
; Contributors:
;
; Description:
;
; Languages
&EN,FR

;Header
#{ "MT_CPbk2AddressSelect_en", "MT_CPbk2AddressSelect_fr" }, (0x12345678),
1,0,0, TYPE=SA

;Localised Vendor name
%{ "Nokia EN", "Nokia FR" }

;Unique Vendor name
: "Nokia"

;Files to install
"\epoc32\release\armv5\urel\MT_CPbk2AddressSelect.dll"-
"!:\sys\bin\MT_CPbk2AddressSelect.dll"

"\epoc32\data\Z\resource\MT_CPbk2AddressSelect.rsc"-
"!:\resource\MT_CPbk2AddressSelect.rsc"
```

#### Description

The package (.pkg) file is used for many other purposes than just to copy a file into the device. Only ‘;Files to install’ section is of importance for the test automation process. In the section there can be one to many files with a special syntax like

```
"<source of the file>"-"<destination of the file in a device>"
; <comments>
```

In this example file, there are two files, which should be picked from some location <source> and install into the device to location <destination>.

### 4. *MT\_CPbk2AddressSelect.rss* file:

#### Description

This is a data file and is mentioned in the package file. This file will be included in the testdrop package and during the test execution the file will be installed on the device at a location

```
"c:\resource\MT_CPbk2AddressSelect.rsc".
```

## Appendix B – Configuration Files

Configuration files are used to configure the build and test automation process. Test component which is to be built and include in a 'testdrop' is mentioned in package\_definition.xml file whereas, build.xml file is used to for setting up other properties for the test.

### a. package\_definition.xml

```
<?xml version="1.0"?>
<SystemDefinition name="phonebook" schema="1.4.0">
  <systemModel>
    <layer name="unit_test_layer">
      <module name="tain">
        <unit unitID="tain_test" name="eunit_tests"
          bldFile="\s60\MT_CPBk2AddressSelect\group" filter="" mrp="" />
      </module>
    </layer>
  </systemModel>
</SystemDefinition>
```

### b. build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="ATS" default="ats-test">
  <dirname file="${ant.file.Build}" property="Build.dir" />

  <property environment="env"/>
  <property name="product.family" value="Helium" />
  <property name="product.name" value="ATS_TEST" />
  <property name="eunitexerunner.flags" value="/E S60AppEnv /R Off" />
  <property name="ats.email.list"
value="firstname.lastname@domain.com"/>
  <property name="ats.flashfiles.minlimit" value="2" />
  <property name="ats.plan.name" value="plan" />
  <property name="ats.product.hwid" value="AB123" />
  <property name="ats.script.type" value="import" />
  <property name="ats.target.platform" value="armv5 urel" />
  <property name="ats.test.timeeout" value="60" />
  <property name="ats.testrun.name" value="test_run" />
  <property name="ats.trace.enabled" value="false" />
  <property name="ats.ctc.enabled" value="false" />
  <property name="ats.ctc.host" value="10.0.0.1" />
  <property name="ats.trace.enabled" value="false" />
  <property name="ats.obey.pkgfiles.rule" value="0" />
  <property name="release.images.dir" value="j:/output/images" />
  <property name="ats.drop.location" value="10.1.2.3" />
  <property name="ats.drop.location" value="j:\output\ats" />
  <property name="ats.username" value="userid" />
  <property name="ats.password" value="password" />
  <import file="${helium.dir}/helium.ant.xml" />
  <fileset id="reference.ats.flash.images" dir="${release.images.dir}">
    <include name="**/${build.id}*.core.fpsx"/>
    <include name="**/${build.id}*.rofs2.fpsx"/>
    <include name="**/${build.id}*.rofs3.fpsx"/>
  </fileset>
</project>
```

## Appendix C – The ‘testdrop’ and the *Test.xml* File

The ‘testdrop’ structure, created by Helium for ATS, for the above component is shown in figure C.1.

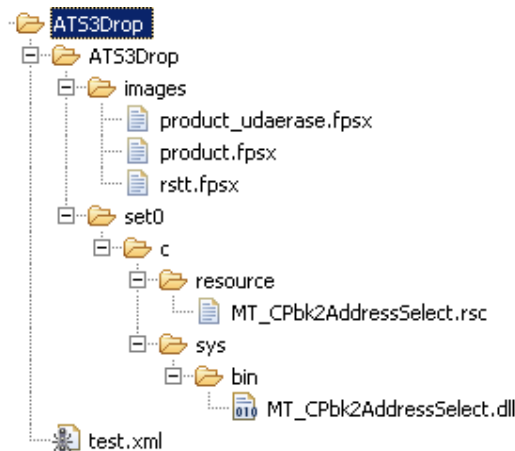


Figure C.1: Structure of a testdrop.zip file

Below is the test.xml file, generated by Helium for the component which was discussed in this thesis.

```

?xml version="1.0" encoding="ISO-8859-1"?>
<test>
  <name>helium_MyDevice_tain</name>
  <buildid>http://publish.results.com/publish/builds/39865/</buildid>
  <target>
    <device alias="DEFAULT_EUNIT" rank="none">
      <property name="HARNESS" value="EUNIT"/>
      <property name="TYPE" value="MyDevice"/>
    </device>
  </target>
  <plan passrate="100" harness="EUNIT" enabled="true"
name="helium_MyDevice_tain Plan" significant="false">
    <session passrate="100" harness="EUNIT" enabled="true"
name="session" significant="false">
      <set passrate="100" harness="EUNIT" enabled="true" name="set0-
j:\s60\MT_CPbk2AddressSelect\group" significant="false">
        <target>
          <device alias="DEFAULT_EUNIT" rank="master"/>
        </target>
        <case passrate="100" harness="EUNIT" enabled="true" name="set0
case" significant="false">
          <flash images="ATS3Drop\images\product.fpsx" target-
alias="DEFAULT_EUNIT"/>
          <flash images="ATS3Drop\images\product_udaerase.fpsx"
target-alias="DEFAULT_EUNIT"/>
          <flash images="ATS3Drop\images\rslt.fpsx" target-
alias="DEFAULT_EUNIT"/>
          <step passrate="100" harness="EUNIT" enabled="true"
name="Create EUNIT log dir" significant="false">
            <command>makedir</command>
            <params>

```

```

        <param dir="c:\Shared\EUnit\logs"/>
    </params>
</step>
    <step passrate="100" harness="EUNIT" enabled="true"
name="Install testmodule: MT_CPbk2AddressSelect.dll"
significant="false">
        <command>install</command>
        <params>
            <param
src="ATS3Drop\set0\c\sys\bin\MT_CPbk2AddressSelect.dll"/>
            <param dst="c:\sys\bin\MT_CPbk2AddressSelect.dll"/>
        </params>
    </step>
    <step passrate="100" harness="EUNIT" enabled="true"
name="Install data: MT_CPbk2AddressSelect.rsc" significant="false">
        <command>install</command>
        <params>
            <param
src="ATS3Drop\set0\c\resource\MT_CPbk2AddressSelect.rsc"/>
            <param dst="c:\resource\MT_CPbk2AddressSelect.rsc"/>
        </params>
    </step>
    <step passrate="100" harness="EUNIT" enabled="true"
name="Execute test: MT_CPbk2AddressSelect.dll" significant="false">
        <command>execute</command>
        <params>
            <param file="z:\sys\bin\EUNITEXERUNNER.EXE"/>
            <param result-
file="c:\Shared\EUnit\logs\MT_CPbk2AddressSelect_log.xml"/>
            <param parameters="/E S60AppEnv /R Off /F
MT_CPbk2AddressSelect /l xml MT_CPbk2AddressSelect.dll"/>
            <param timeout="60"/>
        </params>
    </step>
    <step passrate="100" harness="EUNIT" enabled="true"
name="Fetch test module logs" significant="false">
        <command>fetch-log</command>
        <params>
            <param type="text"/>
            <param delete="true"/>
            <param path="c:\Shared\EUnit\logs\*" />
        </params>
    </step>
</case>
</set>
</session>
</plan>
<postAction>
    <type>DiamondsAction</type>
    <params/>
</postAction>
<postAction>
    <type>FileStoreAction</type>
    <params>
        <param name="to-folder"
value="\abcd001.rst.domain.com\groups\store\Results\helium\builds\5.0
/helium\${RUN_NAMES}\${RUN_START_DATES}\${RUN_START_TIMES}\ATS3_REPORT"/>
        <param name="report-type" value="ATS3_REPORT"/>
        <param name="date-format" value="yyyyMMdd"/>
    </params>
</postAction>

```

```

        <param name="time-format" value="HHmmss"/>
    </params>
</postAction>
<postAction>
    <type>FileStoreAction</type>
    <params>
        <param name="to-folder"
value="\\abcd001.rst.domain.com\groups\store\Results\helium\builds\5.0
/helium\${RUN_NAMES}\${RUN_START_DATES}_${RUN_START_TIMES}\EUNIT_REPORT"/>
        <param name="report-type"
value="EUNIT_COMPONENT_REPORT_ALL_CASES"/>
        <param name="run-log" value="true"/>
        <param name="date-format" value="yyyyMMdd"/>
        <param name="time-format" value="HHmmss"/>
    </params>
</postAction>
<postAction>
    <type>SendEmailAction</type>
    <params>
        <param name="subject" value="ATS3 report for ${RUN_NAMES}
${RUN_START_DATES} ${RUN_START_TIMES}"/>
        <param name="type" value="ATS3_REPORT"/>
        <param name="send-files" value="true"/>
        <param name="to" value="firstname.lastname@domain.com"/>
    </params>
</postAction>
<files>
    <file>ATS3Drop\images\product.fpsx</file>
    <file>ATS3Drop\images\product_udaerase.fpsx</file>
    <file>ATS3Drop\images\rstt.fpsx</file>
    <file>ATS3Drop\set0\c\sys\bin\MT_CPbk2AddressSelect.dll</file>
    <file>ATS3Drop\set0\c\resource\MT_CPbk2AddressSelect.rsc</file>
</files>
</test>

```

## Appendix D – The Survey on CI and TA

The questionnaire and the survey results are presented here. They were conducted by The Company and VTT. Only a few relevant questions are being included in the following questionnaire from the original one. A summary of the responses is included in the last column of the questionnaire.

### The Questionnaire

The response rate of the questionnaire was

- 895 respondents started the questionnaire.
- 499 interrupted the questionnaire.
- The rest of the responses are taken into analysis, the total number is 396.

Question	Type of answer / options	Responses
<b>A. General knowledge about CI</b>		
A.1 My knowledge or experiences of CI are based on	MULTISELECTION i. I don't have any knowledge / experience of CI ii. CI-eLearning module at The Company iii. Some other CI training courses or studies, please describe iv. Personal interest and trials, please describe v. Earlier CI experiences from other organisations, please describe vi. CI work experiences at The Company, please describe	out of 392 i. 169 ii. 127 iii. 39 iv. 43 v. 27 vi. 79
A.2 In my opinion, Continuous Integration (at The Company) means that integration is done (please define how many times)	SINGLE SELECTION i. Several times a day, please define how many times during a day ii. Daily iii. Several times a week, please define how many times during a week iv. Weekly v. Biweekly vi. I don't know	Out of 395 i. 132 ii. 167 iii. 12 iv. 18 v. 9 vi. 57
A.3 What types of builds are done in your team?	MULTISELECTION i. Private builds (personal or team internal builds) ii. Feature builds (technology program environments) iii. Integration builds iv. integration domain builds v. Product builds vi. Symbian builds vii. No builds are done viii. Other	Out of 389 i. 198 ii. 109 iii. 114 iv. 4 v. 74 vi. 7 vii. 81 viii. 15
<b>B. CI practices and tools</b>		
B.1 Please define which of the following practices/processes (that are related to CI) are used in your team?	MULTISELECTION i. Developer Gate ii. Feature Integration iii. Test Automation iv. Test Driven Development, TDD	Ou of 384 i. 110 ii. 128 iii. 155 iv. 53

Question	Type of answer / options	Responses
	v. Mainline Gate vi. Definition of Done, DoD vii. Pair programming viii. None ix. Other, please describe	v. 108 vi. 129 vii. 37 viii. 95 ix. 18
B.2 Please, select the CI tools that are used within your team?	MULTISELECTION i. ASTE ii. ATS3 iii. BlackTusk iv. CMT++ (complexity analysis) v. Code Scanner vi. CTC++ (test coverage measurement) vii. Dashboard viii. EUnit ix. Helium x. Memory leak tool, please specify xi. Own build scripts xii. STIF xiii. Subversion xiv. Code repository and version control xv. Some proprietary tool, please describe xvi. Other, please describe	Out of 348 i. 72 ii. 72 iii. 59 iv. 47 v. 162 vi. 81 vii. 27 viii. 67 ix. 110 x. 35 xi. 84 xii. 100 xiii. 54 xiv. 238 xv. 13 xvi. 37
B.3 What kind of environment(s) your team has for CI (How many and what subsystems you have integrated in the same CI environment?)	FREE TEXT FIELD	A few are... ATS, ASTE, CruiseControl, Helium, Subversion, test automation, etc.
<b>C. General assumptions about CI's benefits, weaknesses, and challenges</b> (The responses in this section were presented in section 7-3)		
C.1 In my opinion, CI improves / has improved the response time to errors	SCALE disagree – agree and comment	
C.2 In my opinion, CI improves / has improved features' quality	SCALE disagree – agree and comment	
C.3 In my opinion, CI is challenging to adopt in multi-site or multi-team environment	SCALE disagree – agree and comment	

Question	Type of answer / options	Responses
C.4 In my opinion, CI improves / has improved my team's efficiency.	SCALE disagree – agree and comment	
C.5 In my opinion, CI improves / has improved co-operation.	SCALE disagree – agree and comment	
C.6 In my opinion, CI improves / has improved the visibility of the development's status progress	SCALE disagree – agree and comment	
<b>D. Feedback loop</b>		
D.1 How do you get feedback (build and/or test results) about the changes you have made	FREE TEXT	A few are...  Email, dashboards, build logs, agile teams, scrum, sprint reviews, myself etc.
D.2 How long it takes to get feedback for your changes (test results)?	SINGLE SELECTION  i. Minutes ii. Hours iii. Days, please specify how many days? iv. Weeks, please specify how many weeks? v. I don't make any changes vi. I am not interested in those	Out of 324  i. 48 ii. 96 iii. 64 iv. 13 v. 77 vi. 26
D.3 How long it takes to get feedback for your changes (build logs)?	SINGLE SELECTION  i. Minutes ii. Hours iii. Days, please specify how many days? iv. Weeks, please specify how many weeks? v. I don't make any changes vi. I am not interested in those	Out of 275  i. 49 ii. 75 iii. 38 iv. 9 v. 77 vi. 27
D.4 How many times per day a developer in your team commits changes to version control tool in average?	MULTISELECTION  i. Several times a day ii. Daily iii. Several times a week, please define how many times during a week iv. Weekly	Out of 302  i. 67 ii. 78 iii. 23 iv. 13 v. 4



Question	Type of answer / options	Responses
	v. Biweekly vi. Monthly vii. I don't know	vi. 1 vii. 116
<b>E. Testing</b>		
E.1 What types of testing are included in your team's work?	MULTISELECTION  i. Unit testing ii. Module testing (API testing) / excluding domain SDK API testing iii. Functional testing iv. Memory Leak analysis v. Regression testing vi. Static analysis, please describe what kind of tools are used vii. Performance / reliability testing viii. Dynamic analysis, please describe what kind of tools are used ix. Other please? Describe. x. We don't do any testing	Out of 326  i. 133 ii. 113 iii. 205 iv. 88 v. 156 vi. 76 vii. 101 viii. 25 ix. 48 x. 45
<b>Specifics_ Testing</b>		
E 1.1 Please, give your estimate how many percent of the (selected testing type) is automated?	SCALE:  Data (i - x) from previous question  0...100%	[No. of responses (mean in %)] i. 127 (60) ii. 108 (51) iii. 197 (33) iv. 85 (45) v. 150 (33) vi. 98 (42) vii. 25 (37) viii. 75 (67) ix. 41 (33)
<b>F. Your opinion about CI adoption</b> (The responses in this section were presented in section 7-3)		
F.1 In your opinion, How individual work habits have changed after CI adoption?	FREE TEXT	
F.2 In your opinion, How does CI support personal working style?	FREE TEXT	
F.3 Please describe the benefits of CI adoption at The Company?	FREE TEXT	
<b>G. Summary</b> (The responses in this section were presented in section 7-3)		
G.1 Please, give your free comments about CI at The Company	FREE TEXT FIELD	