

Rodrigo Modesto de Abreu

# Multi-Stage Continuous Integration

Leveraging Scalability on Agile Software Development

---

Helsinki Metropolia University of Applied Sciences

Masters of Engineering

Information Technology

Thesis

17<sup>th</sup> of May 2013

Author(s) Title	Rodrigo Modesto de Abreu Multi-Stage Continuous Integration
Number of Pages Date	121 pages + 2 appendices 17 <sup>th</sup> of May 2013
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	Mobile Programming
Instructor(s)	Dr. Antero Putkiranta, Senior Lecturer
<p>The objective of this thesis was to provide a detailed view of how a large-scale software organization is aligned with agile and lean concepts focused on a customer-driven approach by quickly adapting to changes.</p> <p>The multi-stage continuous integration is an agile practice which is implemented throughout disciplined agile process and tools to foster the scalability of teams. Thus, this thesis also aimed to provide essential facts that without the adoption of continuous integration the enterprise would struggle to achieve scalability of the purposes of agile principles.</p> <p>During the study case analysis and description, all aspects of a large-scale software product development, were emphasized, covering requirements management, release management, error management, cross-functional teams and mass customization. The study case described that all these areas were better integrated when they were interconnected throughout the multi-stage continuous integration system. It orchestrated and operationalized the interfaces of these functions enforcing agile and lean principles to the organization as a whole. On the same level, the study discussed the advantages of adopting such strategy when scaling out the software enterprise.</p> <p>The whole study was carried out through a qualitative study case analysis. It focused on first to provide theoretical fundamentals of agile and lean approaches and next applying these principles towards a large-scale enterprise, serving as a basis of the empirical qualitative observation.</p> <p>As a result, the thesis provided a practical approach of how to scale software organizations using agile and lean methodologies relying on multi-stage continuous integration. In this circumstance, it appeared to be the most effective practice to leverage scalability on software organizations.</p>	
Keywords	Multi-stage Continuous Integration, Large-Scale Software Development, Lean Software Development, Agile Software Development, Lean Organization, Scrum, Kanban, Debian Packages

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Research Method</b>	<b>3</b>
2.1	<i>Research Design and Material</i>	5
2.2	<i>Reliability and Validity</i>	7
<b>3</b>	<b>Theoretical Background</b>	<b>9</b>
3.1	<i>Continuous Integration</i>	9
3.1.1	Single Source Repository	9
3.1.2	Build Automation	10
3.1.3	Constant Integration Towards the Main Product	10
3.1.4	Easy to Reach Latest Executable	10
3.1.5	Improving Visibility	11
3.2	<i>Agile and Lean Development</i>	11
3.2.1	Value-Driven Development	13
3.2.2	Self-Organizing Teams	13
3.2.3	Frequent Delivery of Working Product	14
3.2.4	Sustainable Development Pace and Simplicity	14
3.2.5	Learning Frequently from Past Mistakes	15
3.3	<i>Agile Methods</i>	15
3.3.1	Scrum	16
3.3.2	Extreme Programming (XP)	21
3.3.3	Lean Software Development	23
3.3.4	Kanban	29
3.4	<i>Package-Based Systems</i>	32
3.4.1	Debian Packages	33
3.4.2	Package Dependencies	37
3.4.3	Package Repositories	38
3.4.4	Other Linux Package-Based Systems	39
3.5	<i>Scaling Agile Teams</i>	39
3.5.1	Defining Scalability	40
3.5.2	Agile Architecture	41
3.5.3	Feature Teams	41
3.5.4	Coordination	42
3.5.5	Continuous Integration Practices	44
3.5.6	Release Planning	47

<b>4 Case Company</b>	<b>48</b>
4.1 <i>History</i>	49
4.2 <i>Mobile Phones Market</i>	50
4.3 <i>MeeGo Organization</i>	52
<b>5 Leveraging Scalability throughout Multi-Stage Continuous Integration</b>	<b>54</b>
5.1 <i>Continuous Integration Process Overview</i>	57
5.2 <i>Continuous Integration Architecture</i>	59
5.3 <i>Requirements and Error Management</i>	68
5.3.1 Initial Development Phase	72
5.3.2 Gate Zero Loop	75
5.3.3 Staging Gate	77
5.3.4 Pre-Release Gate	80
5.3.5 Dependencies Graph	82
5.4 <i>Configurations</i>	84
5.4.1 Types of configuration packages	88
5.4.2 Configurations Inheritance	89
5.4.3 Maintenance in the Packages Configurations	91
5.5 <i>Mass Customization</i>	92
5.5.1 Creating Customization Enablers	95
5.5.2 Package Based Customization	96
5.5.3 Instantiation Process	102
5.5.4 End-to-End Customization Data Flow	103
5.5.5 Web Configurator	107
5.6 <i>Release Planning and Validation</i>	110
<b>6 Discussion</b>	<b>115</b>
<b>7 Conclusion</b>	<b>117</b>
<b>References</b>	<b>119</b>

## **Appendices**

Appendix 1. Agile Principles

Appendix 2. Debian Source Package Description

## 1 Introduction

Nowadays, in the current marketplace, organizations are challenged with new strategies to continue competitive and profitable. These challenges are forcing leaders to look for improved ways to sustain their companies in the market and gain market share. Enterprises that are fast to adapt to this new reality are the ones that will succeed in the transition from traditional ways of management to new levels of achieving productivity and innovation.

The shift from traditional to agile mainly requires a workplace to be completely rethought by giving emphasis on the knowledge worker intrinsic characteristics, which demands autonomy and ownership in a sense of the purpose of what they execute in their work routines, and as consequence increasing the rate of more engaged employees. Denning (2010, 8-15) calls this evolution radical management and focuses on not only being more productive than traditional management, but on liberating the energies, insights, and passions of people by creating workplaces that enable the human spirit.

According to Denning (2010, 8-15), this evolution is achieved through several shifts the organization must perform by getting the company goals right. Therefore, the actual enterprise goals are not merely to produce goods or services to make money for the shareholders, but instead the purpose of the work is to utmost delight customers. As a result of this new goal, people must be structured as self-organizing teams working in short cycles and driven by the client needs and problems. The sense of a continuous improvement, and higher levels of interactive communication are also changes required in the mindset of managers by pushing from top-down and command-and-control management to collaborative, servant leadership and the enabler of self-organized individuals bearing transparency and commitment to improvements as the main values.

Denning (2010, 8-15), also points out that software organizations deserve credit for advancing in achieving such agile evolution. The introduction of agile development in the form of Scrum was deliberately chosen to differentiate this way of developing software from the roles and practices of traditional management. The case company, Nokia, in this account, attempted to implement agile and lean practices by creating a

new endeavor in 2005, whose the main target was to boost and disseminate innovation to the whole company. The creation of OSSO (Open Source Software Operations) was the first step in this effort. The challenges to scale agility were clear impediments in 2008 when the organization focused on the creation of their first mobile phone device. Thus, this thesis analyses and observes how this demand has been addressed with the successful implementation of multi-stage continuous integration practices and tools. As a result, this thesis aims to answer the main research question imposed by these challenges:

How agile teams self-organized with the assistance of multi-stage continuous integration to achieve the scalability targets imposed by the organization lean goals?

Upon the definition of the purpose of this study, the study is carried out through a qualitative approach using a participative observation case study as the main method to collect information about the environment to answer to the research question. Therefore, the participative observation was carried out during the transition period when the scalability and significant changes in the practices were needed.

This report is mainly divided into four sections:

1. Theoretical background which focuses on providing the required knowledge regarding agile and lean software development as well as how those practices are scaled out to the enterprise by highlighting how a multi-level continuous integration can support such scalability. The last section covers package-based systems, detailing *Debian* package management system, the advent practiced by the empirical study.
2. The case company history, the reasons behind the decision to turn into an innovative organization and the results of this venture.
3. The study case in practice. It contains each of the topics and practices of an agile organization covering feature teams aspects and Release & Integration team relationship with the use on Multi-stage continuous integration tools architecture, processes and practices.
4. The final part of this thesis focuses on discussion of the results and conclusions concerning the study case, including topics for further study.

## 2 Research Method

This chapter covers the research methodology used to carry out this thesis work. It aims to classify the research characteristics throughout a methodology analysis and the reasons behind such a classification.

Research is a set of actions and proposals to seek for a solution to a problem. They are based on systematic and rational procedures. A research is therefore performed when there is a problem and as well as lack of information to solve it. Thus, the fundamental objective of a research is to discover answers for problems throughout scientific procedures.

In this sense, there are several ways to classify research due to the approach, procedures, objectives and data collection. In terms of an approach a research can be classified as qualitative, quantitative or both, when the analyzed scope relies also on data quantification and statistics. This research is qualitative once it interprets and observes the environment by investigating the meanings and results in an attempt to bring sense to the research or to the study field (Ritchie & Lewis 2003, 2-3). Investigating and observing agile and lean scaled software development practices, teams composition and structure, and the delivery process towards multi-stage continuous integration that comprises such a world gives to this research the qualitative intent.

The qualitative research utilized a single study case as the procedure to carry out the elements under observation. The procedure classification characterize the technique in which the research was undergone and concerning study cases, specifically, it focuses on the preferred strategy to answer questions such as “why” and “how” placing emphasis on the descriptive and explanatory objectives of the research (Yin, 1994, 1-15).

The justification for a single case study is because the phenomena was examined under an organization in the case company where, despite several teams being followed up, the study case aimed to analyze the collectiveness and collaboration of how teams self-organize using multi-stage continuous integration as the main practice to release the software product. In summary, the study case was carried highlighting the organization as a whole and not towards observation of specific teams or even considering a comparison between them, but in contrast, to examine the overall structure of an agile and lean organization through the scaled practices, roles and goals.

Therefore, I empirically investigate the processes, roles, tools, structure and decisions of which development teams have undergone in order to answer the questions imposed by this type of study case research. I aim to answer the following questions in this study:

- How did teams best benefit from agile and lean approaches when the organization was scaled up in order to succeed toward the delivery of large and complex software product?
- How did multi-stage continuous integration practices were used to support teams agility and complexity requirements in order to avoid extensive integration problems and assure high quality on the product deliveries?
- Why had the entire organization and people underneath it chosen agile and lean methods instead of traditional, well known approach?”
- Why had the organization set up in multi-stage continuous integration as the central practices in order to pursue quality and reach predictability for product releases?
- Why had teams decided to utilize a package-base approach as main delivery unit?

As previously mentioned, the participative observation was the main technique used to collect data for the case study. The reason for such an approach accounts for my extensive working experience occupying several roles in the Release & Integration team. I was responsible for gathering requirements, settling and leading the implementation of multi-stage continuous integration practices and tools among other assignments in the same team.

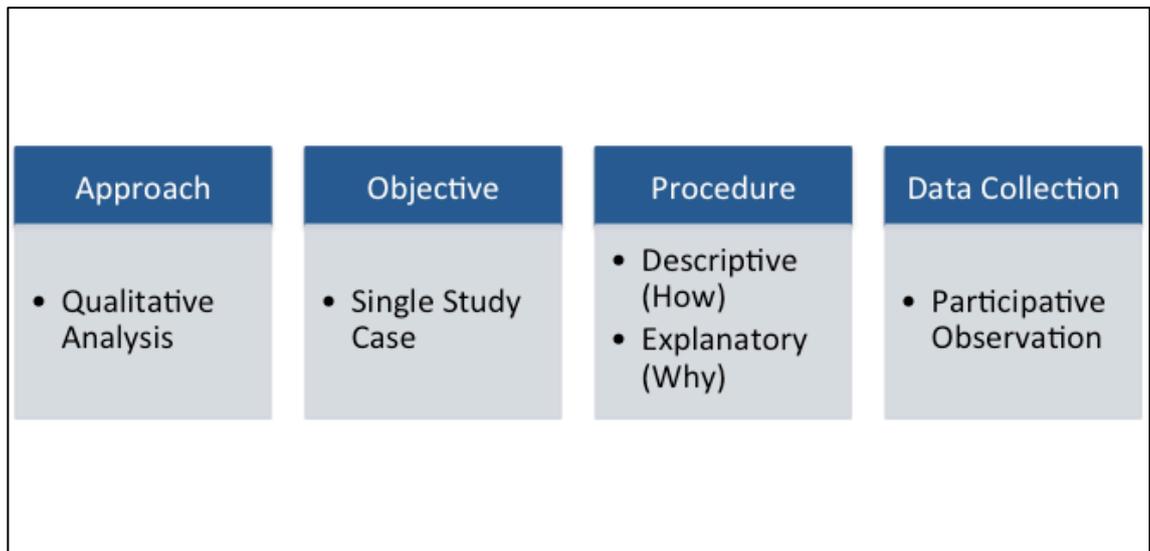


Figure 1: Classification of research methods

As a conclusion, Figure 1 summarizes how I categorized these thesis methods in this order: approach, objective, procedure and data collection.

## 2.1 Research Design and Material

As initially covered in the introduction, the aim of the thesis is to provide an applied approach for multi-stage continuous integration practices in order to substantially support scalability of agile software development teams. It is a vast theme once it deals with several elements regarding software engineering, alongside product development, leading me study and interrelate each of the elements with the main topic. Additionally, if the subject had not been thoroughly covered, it would have brought an inadequate idea about the theme, lacking crucial pieces of information essential to judge all aspects of large-scale and complex software products development.

Therefore, the theoretical framework is fundamental to provide a deep understanding of all facets of this study. It consists of a literature review comprising the theory that explains agile and lean methodologies emphasizing continuous integration. After that, this work reapplies this knowledge to provide the background of how to scale up these methodologies to a large enterprise by maintaining the same values and obtaining same the results as when applied to single, small teams. Additionally, this study provides valuable information regarding package-based systems and package management tools once the empirical work emphasizes such approach when dealing with mul-

ti-stage continuous integration. At the same level, package repositories are also covered as part of approval gates on the underlying system.

The empirical study relies on my participative observation and experience in such a scaled agile environment for five years in leading roles, participating in the development of Continuous Integration System among other responsibilities. It also relies on the case company intranet materials, such as wikis, as well as the error management system.

Basically, the theory implies several aspects which have been observed and corroborated during the case study data collection. As an example, several practices are described in the literature, detailing how organization scalability is supposed to occur along with several cases described by those authors. However, practice showed particularities that are not covered in the literature, mainly observed on the Multi-Stage continuous integration system setup, which brought much more flexibility to teams against what had been covered in the literature. Moreover, there is no similar approach in the literature that discusses package based continuous integration, bringing considerable improvements and advantages when the organization considers leveraging Multi-stage continuous integration.

Upon the selection of the case study and data gathering approach, the participative observation reported the experiences from the field. Consequently, the aim of this case study is not to infer findings from a sample to a population, but to engender patterns and linkages of theoretical importance (Bryman 1989, 144). In this sense, the “how” and “why” analysis focuses on the evaluation of how teams self-organized and the scalability decisions (“why”) taken in order to maintain agility, quality on deliverables and visibility.

Finally, as part of the report execution, I discuss the relevance of the empirical data observed against the current literature available, along with conclusions of such a study.

## 2.2 Reliability and Validity

Patton (1999, 1189) recognizes that "issues of quality and credibility intersect with audience and intended research purposes" (1999, 1189). In this sense, the criteria for a good research include:

- The researcher should use rigorous techniques and methods for gathering high quality data;
- The data should be carefully analyzed, based on triangulation;
- The credibility of the researcher such as training and experience including track record on the study field as well as status and presentation of self;
- Qualitative analysis should bear a creative inquiry, yet be methodical;
- Sufficient detail should be reported to allow others to judge the quality of the resulting product.

According to Patton (1999, 1191), statistical analysis follows formulas and rules while, at the core, qualitative analysis is a creative process, depending on the insights and conceptual capabilities of the analyst. While creative approaches may extend more routine kinds of statistical analysis, qualitative analysis depends from the beginning on astute pattern recognition. The patterns were identified during the research observing the self-organization of agile development teams towards the product common goal: to have software deliverables continuously integrated across an evolving product, demonstrating progress and adaptability during the course of the program. Along with such inductive analysis, I have looked on rival explanations on the corresponding literature in order to realize different findings or possibilities. In practice, I have found the full support in the literature concerning such observations.

However, Patton (1999, 1192) states that a single method adequately solves the problem of rival explanations; therefore, I have also included the triangulation as a method to corroborate the different aspects of the empirical study. In such an approach, it is possible to achieve triangulation within a qualitative inquiry strategy by combining different kinds of qualitative methods, mixing purposeful samples, and including multiple perspectives. Accordingly, it was achieved using multiple perspectives based on theo-

ries to interpret the data, that is, theory triangulation in a sense that theory and observations were aligned sustaining the analysis. It can be demonstrated in the following examples:

- Leffingwell, Griffiths and Ambler mention the importance of holding one single issues tracking system storing bugs, requirements, user stories, operational requests and many other sources of work items to support teams for further planning and dependencies realization.
- Leffingwell, Ambler, Larman and Gruver state about the advantages of feature teams when scaling up to the organization level by promoting the fast delivery of high value-added features.
- The ideas concerning Release Planning are similar among covered “agilist” authors referring to scalability issues when dealing with team planning and organization planning towards the final product and underlying features.
- Ambler and Gruver indicate the quality benefits and visibility enhancements when choosing a multi-level or stage approach for continuous integration practices and tools. The higher degree of automation described by such authors is the main advantage also observed in empirical observation.
- Gilmore reinforces the Mass Customization initiative addressed during the case study by mentioning the collaborative approach that customers and vendors work together targeting to identify customization requirements and market advantages on their products.

Patton (1999, 1205) points out the experience and competence of the researcher in the study field as a participant observer. The thread that runs through this discussion of researcher credibility is the importance of intellectual rigor and professional integrity (Patton 1999, 1205). There are no simple formulas or clear-cut rules to direct the performance of a credible, high-quality analysis. Thus, it is important to note that I had 5 years of experience in the study environment, participating in several roles as well as dealing with almost all teams in the organization and spending enough time with them to be able to judge, analyze and understand their intrinsic problems and how they were able to self-organize in order to overcome them. Likewise, I have contributed to fulfill organization needs to scale Continuous Integration system by creating the automation

tools and practices for multi-stage continuous integration, which had the function to interconnect and aggregate value to all teams' feature deliveries.

In terms of limits, Patton (1999, 1197) defines the limitations based by the nature of qualitative findings that are highly contextual and case-dependent. In this study, hence, the limitations on problems of temporal sampling, i.e., limitations resulting from the time periods during the observations took place. The explanation for such is that the case company has changed the organizational strategy by disrupting the case study organization. In this sense, the observing environment where the data has been gathered does not exist by the time of this thesis writing, raising questions about the validity and effectiveness of such observations. The reasons for this decision rely on economic and strategic challenges that case the company was undergoing. Such restructuring, led to a complete shift concerning Operating System development towards the responsibility of a licensing company.

### **3 Theoretical Background**

#### **3.1 Continuous Integration**

Continuous Integration is an agile software development practice where members of a team integrate their work frequently, leading to multiple integrations per day. Each integration is verified by an automated build (including automated tests) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. (Fowler, 2006.)

As part of Continuous Integration some key practices make up an effective process, which is described in the following:

##### **3.1.1 Single Source Repository**

All artifacts required to build the product should be placed in a single repository under revision control. The convention is that the system should be buildable from the artifacts under version control and not require additional dependencies.

### 3.1.2 Build Automation

Build is the process of turning source code into a binary code called compilation. In many systems it also involves the packaging of a binary code up to the executable or deployment generation of the system built.

When automating a build system, it is similar to include the source code into an assembly line, where several steps are automatically executed as an atomic operation, aimed to effectively produce a so called build or executable of the system. These steps include actions as compiling the source code, executing automated tests, generating documentation, inspection and deployment.

The attempt of Continuous Integration is to keep the build time short. Thus, developers receive the feedback of their build submissions as fast as possible to fix issues or even solve integration problems. Therefore, this practice emphasizes the agile practice of *failing fast* by reducing costs of changes, and keeping and maintaining a releasable product free of errors constantly.

### 3.1.3 Constant Integration Towards the Main Product

The mainline in the source repository should be the place for the working version of the software or where the final product is built. It, in fact, reduces the number of conflicting changes. Committing all changes at least once a day is generally considered part of the definition of Continuous Integration. It is also preferred to have changes integrated rather than keeping several versions of the same product to be maintained simultaneously.

### 3.1.4 Easy to Reach Latest Executable

Anyone involved in a software project should be able to get the latest executable and be able to run it: for demonstrations, exploratory testing, or just to see what has changed from previous release either daily or weekly.

In agile development, processes explicitly expect feedback from working software instead of documentation and approved, formal, specifications. By making easy to reach the product software or executable increases the project progress visibility towards its

goals since any stakeholder is able to try out a fresh build from Continuous Integration System.

### 3.1.5 Improving Visibility

It should be easy to find out whether the build breaks and, if so, who made the relevant change, by ensuring everyone can easily see the state of the system and the changes that have been made to it. Thus, Continuous Integration fosters the communication of the mainline build state and consequently the project progress and impediments. It also provides a history of changes, allowing team members to get a good sense of a recent activity in the project as well. It makes visible what components have been changed, what has been changed, when and who made the changes.

In conclusion, adopting Continuous Integration practices favors the effect of finding and fixing integration bugs early in the development process. It saves both time and costs over the lifespan of a project. Also, it reduces risks, increases feedback as well as product visibility. It directly relies on test automation, the coverage and the range of the tests, from unit tests to regression tests. So, the success of implementation of Continuous Integration is tied to successful design and constantly improving tests in the project. Another important aspect is the possibility of CI to constantly release working software at any time, increasing the possibilities of user acceptance and progress visibility.

## 3.2 Agile and Lean Development

Different types of projects require different methods. Some projects, especially knowledge worker projects occurring in a fast moving or time-constrained environments, call for an agile approach. In the latest major revolution, the information revolution, workers are focused on information and collaboration rather than manufacturing. It places value on the ownership of knowledge and ability to use that knowledge to create or improve goods and services.

The information revolution relies on knowledge workers. These are people with subject matter expertise who communicate their knowledge and take part in analysis or development efforts. They are software developers, teachers, scientists, lawyers, doctors and many others. The communication and collaboration required for knowledge worker

projects are often more uncertain and less defined than in industrial work. Past projects have applied industrial work techniques to knowledge worker projects, causing frustration, and as consequence, failures increased.

Thus, Agile methods were developed in response. These methods collect knowledge worker techniques and adapt them for use in projects, mainly in software development initiatives, where the failure rates are quite high among the knowledge worker domain. The best example of these initiatives, are Scrum and Extreme Programming (XP) methodologies. Others have, as well, an important role in the agile scope such as Feature Driven Development (FDD), Test Driven Development (TDD) and Lean Software Development. However, it is not an agile methodology but its values are very closely aligned. (Griffths, 2012, 20.)

In February 2001, software and methodology experts met at the Snowbird, Utah resort, to discuss lightweight development methods. They published the *Manifesto for Agile Software Development* to define the approach known as agile software development. Some of the manifesto's authors formed the Agile Alliance, a nonprofit organization that promotes software development according to the manifesto's principles.

The Agile Manifesto reads, as follows (Agile Alliance, Online.):

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools  
**Working software over** comprehensive documentation  
**Customer collaboration** over contract negotiation  
**Responding to change** over following a plan

In this context, The Agile Manifesto is not a set of rules directing practitioners to proceed in a way rather than the other. It is subtler. It guides people to consider projects from a value-based perspective. In this sense, it will still be required processes, tools, documentation and plans to implement projects; yet, while dealing with these assets, the focus must be on the people engaged, the product being built, cooperation and flexibility. Agility is the capacity to execute projects while focusing the efforts on the left side of these values statements, rather than those on the right.

On the same level, the values have originated principles by guiding agile development that is listed in Appendix 2. It emphasizes the following practices in agile environment (Griffths, 2012, 22-30):

- Value-Driven development
- Self-Organizing teams
- Deliver working product frequently
- Sustainable development pace and simplicity
- Learn frequently from past mistakes

### 3.2.1 Value-Driven Development

The highest priority on agile projects is to satisfy the customer by delivering what brings more value to his/her business. As a result, it is important to note the focus on early and continuous delivery of working and valuable software rather than on descriptive plans and documentation. In practice it means close collaboration and feedback from stakeholders and business representatives following up by frequent demos of the working product. Both, the development team and business learn from each other and, as a consequence, build a common shared view about the goals of the project. This attitude reflects on welcoming and embracing change requirements, even in the late phases of the project.

By accepting that changes will happen and by setting up an efficient way to deal with them, a team can spend more time developing the project's end product. Rather than suppressing changes, agile methods work to create a well-understood, high-visibility way of handling changes that keep the project adaptive and flexible as long as possible.

### 3.2.2 Self-Organizing Teams

Self-Organizing teams mean finding an approach that works best for the team's methods, relationships and environment. Teams thoroughly understand and support the approach, because they helped create it. As a result they produce better work.

This also means that self-organization boosts motivation and it can make the biggest difference in whether a project will be delivered successfully and efficiently. Therefore, people work better when they are given the autonomy to organize and plan their own work, by promoting empowered teams, freeing the team from micromanagement and instead, emphasizing on craftsmanship, peer collaboration, and teamwork, which results in higher rates of productivity.

Nevertheless, it improves architectures, requirements and design results, since they were implemented by the ones who originated them along with the decisions made by the team itself proving the higher level of ownership and conviction. In effect, agile methods leverage the capacity to teams to best diagnose and enhance project technical aspects. After all, team members are the most informed about the project.

### 3.2.3 Frequent Delivery of Working Product

The aim of this principle is to emphasize the importance of releasing work to a test environment and getting feedback. It is the core idea behind the need of Continuous Integration tools and practices. The approach of agile teams is to receive brief and frequent feedback on what is being created and thus adapt against required changes before proceeding. The short timeframes also have the benefit of keeping the business engaged and in constant communication for opportunities to receive feedback and learn about new requirements or changes. That is the main reason why teams pursue the “working software” as the primary measure of progress. The importance of “working software” ensures the teams get acceptance of features by following their Definition of Done and create a result-oriented view of the project.

### 3.2.4 Sustainable Development Pace and Simplicity

In agile, simplicity means the art of maximizing the amount of work to be done. It is essential to what is important for the product rather than features that will be seldom used. Focusing on simplicity pushes the team attention to indispensable requirements, the ones that bring highest value to customers and business.

Therefore, agile methods seek the “simplest thing that could possibly work” and recommend this solution to be built first. This approach is not intended to preclude further

extension and elaboration of a product, it instead is merely expressing to build the plain vanilla first by mitigating risks and boosting sponsor confidence.

By keeping deliverables simple, the development team has to be mindful of keeping the design clean, efficient and adaptable to changes. Technical excellence and good design allow the team to understand and update the design easily, responding to changes faster. As a result, projects must have time to undertake refactoring. Refactoring is understood as the simplifications that need to be made to source code to ensure it is stable and can be maintained over a long term.

Accordingly, a project needs to balance its efforts of delivering high-value features with giving continuous attention to design of the solutions. This sustainable pace allows a system to deliver long-term value. Bearing the same sustainable pace idea, teams benefit to have a work-life balance, so that it creates a happier, stable and more productive team. It causes less tensions and relationships improve if teams pay close attention to the level of effort the team is putting forth to ensure a sustainable pace.

### 3.2.5 Learning Frequently from Past Mistakes

Agile projects employ frequent reviews or “look backs”, called retrospectives, to reflect on how facts are proceeding on the project and to identify opportunities of improvements. These retrospectives are frequently executed as part of several iterations executed across the project. One advantage of doing retrospectives often is that the details will not be forgotten when compared to traditional project management approaches, where lessons learnt are conducted once by the end of the project. Instead, in an agile project, the lessons learned are captured as the project progresses, so that the team realizes the relevance and is pushed to tune and adjust their behavior accordingly.

## 3.3 Agile Methods

Each agile methodology has a slightly different approach to implementing the core values from the Agile Manifesto. This thesis covers, in more detail, Scrum, XP and Lean Software Development approaches. However, it does not overlook other methodologies by tackling their characteristics whenever relevant for the context discussed.

Before going on specific aspects of Scrum and XP, it is important to note a crucial difference between them. Scrum is a framework for agile development processes and it does not include specific engineering practices. Conversely, XP focuses on engineering practices but does not include an overarching framework of development processes. That does not mean that Scrum does not recommend certain engineering practices or that XP has no process. This complementary view, and sometimes overlapping, of these two methods benefits projects to situational tailoring the process in practice by development teams due to complexity and sizes of the projects.

### 3.3.1 Scrum

The Scrum framework is a set of team guidance practices, roles, events and rules to execute projects. The theory behind Scrum is based on the three pillars of Transparency, Inspection and Adaptation (Griffths, 2012, 35-39):

- Transparency: This pillar involves giving visibility to those responsible for the outcome. This covers agreements and common definitions of what is expected from everyone in terms of commitment, behaviour and collaboration.
- Inspection: This pillar involves timely checks on how well a project is progressing toward its goals by looking for problematic deviations or differences that require adjustments to assert the target goals.
- Adaptation: This pillar affects the adjustments in the process to minimize further issues if inspection shows a problem.

The Scrum process plans four opportunities for inspection and respective adaptation in the framework: The Sprint retrospective, Daily Scrum meeting, Sprint review meeting and Sprint planning meeting. These opportunities are indicated in Figure 2:

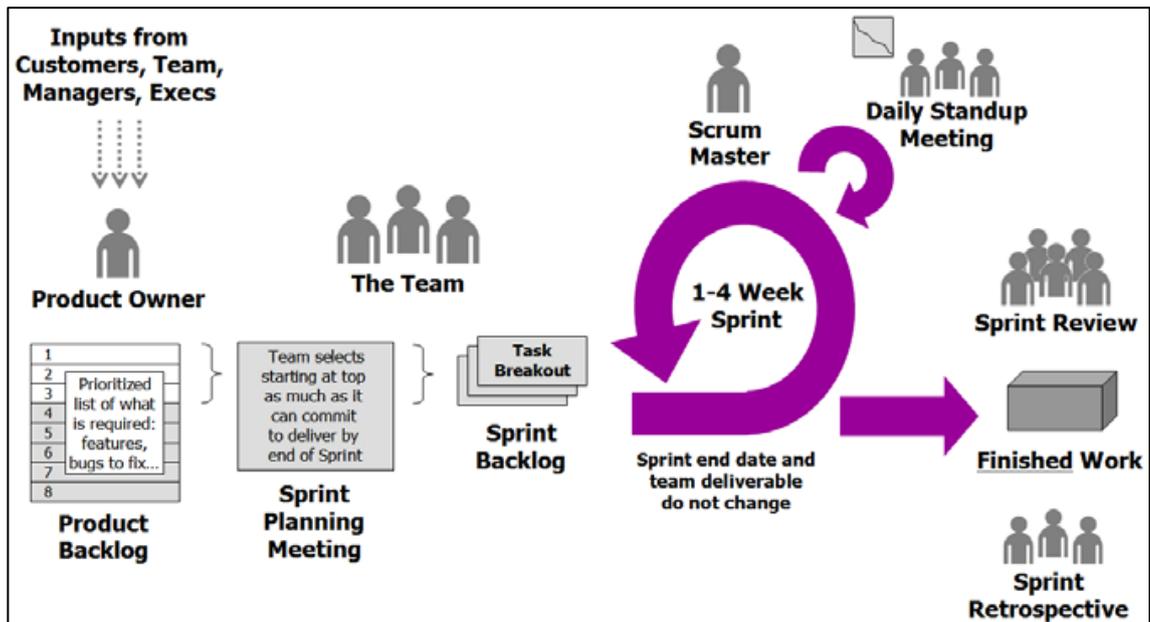


Figure 2: Scrum Process. Depicted from (Ramloll, 2010)

Figure 2 also depicts the three Scrum roles: Development Team, Product Owner and Scrum Master. These roles make up the Scrum Team of a project.

- Development Team: The group of professionals responsible to build the product increments in each iteration or “sprint”. The team is empowered to manage its own work and its members are self-organizing and cross-functional.
- Product Owner: Responsible for maximizing the value of the product. This person manages the product backlog including its prioritization, accuracy, shared understanding, value, and visibility.
- Scrum Master: Ensures that Scrum is understood and used acting as a servant leader to the development team, removing impediments to progress, facilitating events as needed and providing coach.

Similarly, Figure 2 also defines Scrum events:

- Sprints: A sprint is a timeboxed (time-constrained) iteration of one month or less to build a potentially releasable product. Each sprint includes a sprint planning meeting, daily stand-up meeting, the development work, a sprint review meeting and the sprint retrospective. During the sprint, no changes are made that would affect the sprint goal.

- Sprint Planning Meeting: A sprint planning meeting is used to determine what will be delivered in that sprint and how the work will be achieved. Thus, the Product Owner presents the backlog items and the whole team creates a shared understanding. The team in return forecasts what can be delivered based on estimates of past performances and capacity to define the sprint goal. The team then determines how the assigned functionalities will be built and how the team organizes to deliver the sprint goal.
- Daily Stand-up Meeting: It based on a 15-minute timeboxed daily meeting where activities are synchronized and issues raised, held usually at the same place and time. Each team member answers to the following three questions in this meeting:
  1. What has been achieved since the last meeting?
  2. What will be done before the next meeting?
  3. What obstacles or impediments are in the way?

The daily scrum is used to assess progress toward the sprint goal. The Scrum Master makes sure these meetings take place and helps remove any identified obstacles.

- Sprint Review: This meeting is held at the end of the sprint to inspect the increment as well as the evolving product. In this sense, the development team demonstrates the work that is regarded as done and answers any questions that could rise about the increment. Thus, the product owner decides and accepts what is really done. As the final step of this meeting, the team and product owner discuss the remaining product backlog and determine what to do next.
- Sprint Retrospective: At the end of the sprint, the team reflects on the process and look for opportunities for improvement. It occurs after the sprint review and before the next sprint-planning meeting. This meeting focuses on inspection on people, relationships, processes and tools. Likewise any traditional lessons learned session, the team and Scrum Master explore what went well by identifying opportunities for improvement which can be implemented in the forthcoming sprints.

In this regard, while the team focuses on the sprints targets and sprint backlog, the product owner takes care of prioritizing and envisioning the product backlog by creating a list of ordered items needed for the product. This list is comprised of requirements

that have a different level of refinement such as tasks, user stories, feature descriptions or even epics. They usually represent a hierarchy of the items, which the team or the product owner work to define higher level of details in order to explore specific aspects of features during planning activities, as seen on Figure 3. There is, actually, no single universal structure for the requirements hierarchy. The most common one is depicted in Figure 3, where epics represent a set of features that are broke down into functionalities such as features and a more detailed and measurable chunks as user stories and, in consequence fine-grained items which are more traceable and implementation-oriented as tasks. (Griffths, 2012, 129.)

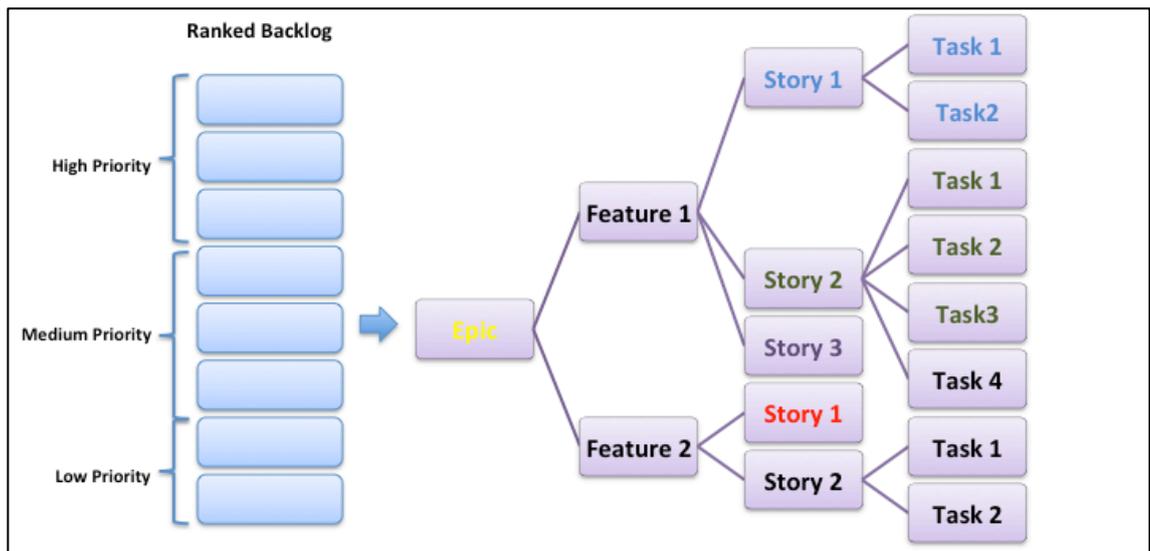


Figure 3: Requirements Hierarchy with epics over features.

While epics and features have a grouping role in the backlog, holding no actual implementation, user stories and tasks are more understandable pieces of business functionality reflected in the final product. Thus, the most frequent item in a backlog is the user story and is commonly written in the following template format:

*“As a <Role>, I want <Functionality>, so that <Business Benefit>.”*

The advantage is to mainly identify the stakeholder (who is asking for it) and the benefit to have the functionality implemented for accurate prioritization. User stories must have characteristics that evidence their accuracy and effectiveness when they are identified and composed. The *INVEST* mnemonic is often used as a reminder of how to write user stories. The mnemonic in detail means the following (Griffths, 2012, 127.):

- I – Independent: The attempt is to create independent user stories that can be selected on merit rather than being added in the backlog because other user stories are dependent on them.
- N – Negotiable: It should be possible that user stories provide a means of negotiating trade-offs based on cost and function. Negotiating user stories leads to an improved understanding of the true requirements, costs, and acceptable compromises.
- V – Valuable: The user story should explain the value or benefit of the requirement giving primarily hints of its priority among other user stories.
- E – Estimate: In order to rank user stories in the backlog based on cost and benefit trade-off, it is important to be able to estimate the effort to implement such item.
- S – Small: Small user stories are easier to estimate and test. Also, large stories are difficult to reprioritize and measure progress when they are under implementation.
- T- Testable: Having testable user stories, means, they can be validated and evaluated by stakeholders and therefore, later on, be accepted by stakeholders.

As the product Backlog, the sprint backlog is a set of items that were selected for emphasis in a specific sprint. It is a highly visible set of the work being undertaken and might only be updated by the development team. In such a manner, items are implemented and, in order to demonstrate progress during the current sprint, the burn-down chart is frequently updated to show the estimated remaining effort in the project while the burn-up charts evidence what has been delivered and accepted by the product owner. This means that as more work is completed, a burndown chart will show a progress indicator moving downward to indicate the reduced amount of work that still needs to be done. In contrast, the progress indicator in a burn-up chart will move upward, to show the increasing amount of work completed. The completed work is established when everyone has unambiguity in agreement towards the definition of done collectively created by the team itself. Therefore, frequent discussion of what “done” means is essential to avoid gaps in expectation or poorly accepted products. For software projects, there is a list of items that should be discussed and checked before de-

clarifying something is completed. This list means on verifying whether the implemented items are tested, coded, designed, integrated, installed, reviewed and accepted. Consequently, agile works best when teams make a little progress on every aspect of the project every day, rather than reserving the last few days of the increment for getting user stories completed. (Shore & Warden, 2008, 156-157.)

### 3.3.2 Extreme Programming (XP)

Extreme Programming is a software-development-centric agile method. While Scrum at the project management level focuses on prioritizing work and getting feedback, XP focuses on good practices in software development. In XP, lightweight requirements, user stories, are used in planning releases and iterations. Iterations are typically two weeks long, and developers work in pairs to write code during these iterations. All code developed is subjected to rigorous and frequent testing. Then up-on approval by the “on-site” customer, the software is delivered as small releases. The XP practices are indicated in Figure 4.

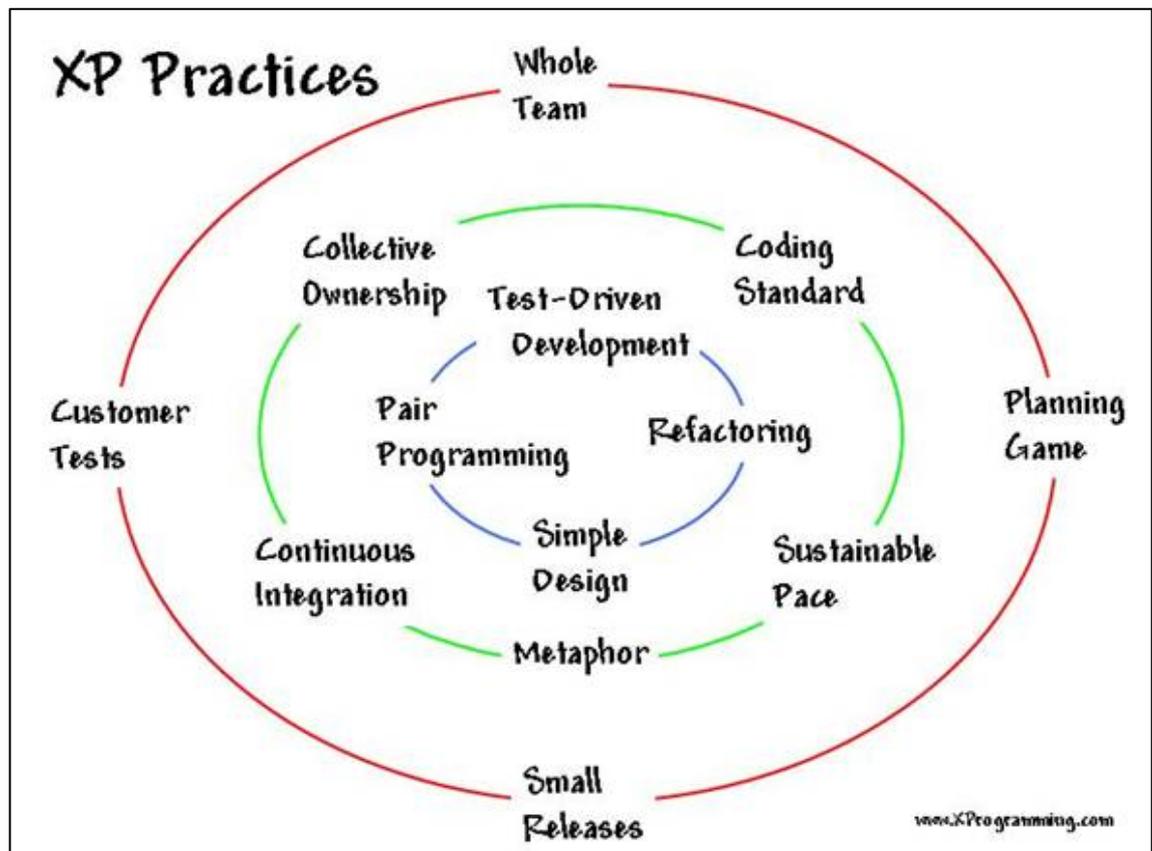


Figure 4: XP Core Practices. Depicted from (Jeffries, n.d.)

In Extreme Programming, every contributor to the project is an integral part of the “Whole Team”. The team forms around a business representative called “the Customer”, who sits with the team and works with them daily. Extreme Programming teams use a simple form of planning and tracking to decide what should be done next and to predict when the project will be done. Focused on the business value, the team produces the software in a series of small fully integrated releases that pass all the tests the Customer has defined. Team members work together in pairs and usually collocated as a unique group, with simple design and devotedly tested code, improving the design continually to keep it always just right for the current needs. The target of the team is to keep the system integrated and running all the time. The programmers write all production code in pairs, and all work together all the time. They code in a consistent style so that everyone can understand and improve all the code as needed. Therefore, Extreme Programming is about team responsibility for all code, for coding in a consistent pattern so that everyone can read everyone’s code, about keeping the system running and integrated all the time. Essentially, XP team shares a common and simple picture of what the system looks like. Everyone works at a pace that can be sustained indefinitely. (Griffiths, 2012, 39-43.)

At the same level of practices, there are three concepts that also explain the essence of how XP handles the software development projects that apply this method.

- Technical Debt: It is the total amount of less-than-perfect design and implementation decisions in a project. This includes “quick and dirty hacks” intended just to get something working and design decisions that may no longer apply due to business changes. Technical debt can even come from development practices such as an awkward build process or incomplete test coverage. The bad practices of poor formatting, unintelligible control flow, and insufficient testing are great indicators of systematic bugs in the code. The result debt often comes in the form of higher maintenance costs. Simple tasks that ought to take minutes may stretch into hours or afternoons. Unchecked technical debt makes the software more expensive to modify than to re-implement. It creates waste. Thus, XP takes an approach to technical debt by managing it being constantly vigilant, by avoiding shortcuts, using simple design and refactoring relentlessly in sum by constantly applying XP’s development practices.
- Continuous Integration: The ultimate goal of Continuous Integration in XP is to enable the team to deploy by creating the product at any time. This tells to the

team when they need to stop and fix issues as early as possible by avoiding integration delays and reworks. The target is to be technologically ready to release even if the functionalities are not ready to be released. In this sense the practice of small releases and to mainly avoid collisions in the code. Collisions are most likely when wide-ranging changes are implemented affecting everyone else deliverables. Integrating frequently and in small portions, it controls how changes are incorporated into the product and mainly, by making sure they are tested during the integration cycle.

- Spike Solutions: XP values concrete data over speculation by avoiding implementation based on assumptions which may lead to extensive re-work and as consequence, technical debt. Whenever the team is faced with a question, there is no speculation about the answer, it is conducted an experiment instead by identifying how to use the real data to make progress. Spikes can be conducted towards architectural or risk proof of concepts. It is usually reserved a period timebox period in the increments that will be used for further investigation in order to break wrong assumption main towards design and risk. The result might lead to improved refactoring, simple design and enhanced coverage of test cases.

### 3.3.3 Lean Software Development

Lean development covers the principles that have been taken from lean manufacturing approaches and applied to software development. Poppendieck and Poppendieck, (2006, 47-63) have translated these principles into seven core concepts in software development:

#### **Principle 1: Eliminate Waste**

Waste is considered everything that does not add value to end customer. Therefore, the first step to eliminate waste is to develop a keen sense of what value really is. Nevertheless, great software development organizations develop a deep sense of customer value and continually delight their customers. As a conclusion, waste is anything that interferes with giving customers what they value at the time and place where it will provide the most value. Anything that does not add customer value is waste, and any delay that keeps customers from getting the value when they want is also waste.

Table 1 compares the wastes in manufacturing and in software development by providing an idea of how waste can be identified and categorized in software. In order to have a thorough view of waste it is important to cover each of the seven forms of waste.

Table 1: The Seven types of Waste comparison between manufacturing and software development. Adapted from Poppendieck and Poppendieck (2006, 95 – Table 4.1)

<b>Manufacturing</b>	<b>Software Development</b>
In-Process Inventory	Partially Done Work
Over-Production	Extra Features
Extra Processing	Relearning
Transportation	Handoffs
Motion	Task Switching
Waiting	Delays
Defects	Defects

The inventory of software development is partially done work. Partially done software has all of the evils of manufacturing inventory: It gets lost, grows obsolete, hides quality problems, and ties up money. Moreover, much of the risk of software development lies in partially done work. When requirements are specified long before coding, of course they change. When testing occurs long after coding, test-and-fix re-work is inevitable. In a worst picture, these types of re-work are often just a precursor to the ever larger amendments created by delayed (aka big-bang) integration.

However, according to Poppendieck and Poppendieck (2006, 95-103), the biggest source of waste in software development is extra features. Only about 20 percent of the features and functions in typical custom software are used regularly. Around two-thirds of the features and functions in typical custom software are rarely used. If there is not a clear and present economic need for the feature, it should not be developed.

Waste is also realized when it is required to rediscover something once it was known and have forgotten. Nevertheless, our approach to capturing knowledge is quite often far too verbose and far less rigorous than it ought to be. Another serious waste is not to retain or fail to engage the outside knowledge people bring to the workplace. It is critical to leverage the knowledge of all workers by drawing on the experience that they have built up over time.

Handoffs and task switching are other two serious wastes in software development. Both rely on lost of tacit knowledge and increase of complexity of communication on the distracting activity of moving from one task to another. The waste created delays

the development flow, decreases quality and increases the amount of work being implemented causing extra risks to the project. The major result of these two previous wastes causes another waste, the delay. Mainly handoffs, forcing team members to wait until someone is free or even concerning decisions to be made about issues that rise during the development. Complete, collocated teams and short iterations with regular feedback can dramatically decrease delays while increasing the quality of decisions.

Defects are the seventh way of creating waste in a development cycle. They are closely related to how the code is tested and inspected across the development flow. In fact, a good agile team has an extremely low defect rate, because the primary focus is on mistake-proofing the code and making defects unusual. The secondary focus is on finding defects as early as possible and looking for ways to keep that kind of defect from reoccurring.

Another important factor in eliminating waste is to precisely identify the elements of waste that could be removed to improve the efficiency of a process. This technique is called Value Stream Maps (VSM) by visually creating visual maps of the process focusing on the customer needs and problems. The target is to reduce the time it takes from the customer's order up to have the deliverable handed over to him. All non-value-adding wastes are identified by usually looking for long delays, loop-backs and queues throughout the process and thus removing them by creating a new value stream flow.

### **Principle 2: Build Quality In**

Testing the product code early makes building quality into the code from the start as part of the goal. The target of tests, and the people that develop and runs tests, is to prevent defects, not to find them. A quality assurance organization should champion processes that build quality into the code from the start rather than test quality later on.

After all, there are two kinds of inspection: inspection after defects occur and inspection to prevent defects. If the team really wants quality, they just do not inspect after the fact, control conditions, so as not to allow defects in the first place. If this is not possible, then the product needs inspection after each small step, so that defects are caught immediately after they occur. When a defect is found, the proposed lean practice is to stop the line, find its cause, and fix it immediately.

Another aspect of defects prevention concerns to defect tracking systems which are queues of partially done work, queues of rework in fact. All too often, just because defects are in a queue, they are tracked and will be part of the product at some point. Nevertheless, in the lean paradigm, queues are collection points for waste. The goal is to have no defects in the queue. In fact, the ultimate goal is to eliminate the defect tracking queue altogether by building quality in.

Test-driven development (TDD) is an effective approach to improving code quality. The philosophy behind TDD is that tests should be written before the code is written. In other words, developers should first think about how the functionality should be tested and then write tests in a unit testing language before they actually start implementing the actual code. Initially tests will fail, since developers have not yet written the code to deliver the required functionality. Therefore, with TDD, developers initiate a cycle of writing code and running the tests until the code passes all the tests. (Poppendieck and Poppendieck, 2006, 46-48.)

### **Principle 3: Create Knowledge**

One of the puzzling aspects of "waterfall" development is the idea that knowledge, in the form of "requirements," exists prior and separate from coding. Software development is a knowledge-creating process. While an overall architectural concept will be sketched out prior to coding, the validation of that architecture comes as the code is being written. In practice, the detailed design of software always occurs during coding, even if a detailed design document was written ahead of time. An early design cannot fully anticipate the complexity encountered during implementation, nor can it take into account the ongoing feedback that comes from actually building the software. Worse, early detailed designs are not amenable to feedback from stakeholders and customers. A development process focused on creating knowledge will expect the design to evolve during coding and will not waste time locking it down prematurely.

According to Poppendieck and Poppendieck, (2006, 51-54) companies that have exhibited long-term excellence in product development share a common trait: They generate new knowledge through disciplined experimentation and codify that knowledge concisely to make it accessible to the whole organization.

Therefore, software projects are business and technology learning experiences. It is important to have a development process that encourages systematic learning throughout the development cycle, but also systematically improve that development

process. Because of a complex environment, there will be changes that require fast adaptation and continuous learning from these adaptations.

**Principle 4: Defer Commitment**

Deferring commitment in another words means deciding as late as possible. Lean software development practices delay freezing all design decisions as long as possible. So that, it is easier to change a decision that has not yet been made. This development technique emphasizes developing a robust, change-tolerant design, that accepts the inevitability of change and structures the system so that it can be readily adapted to the most likely kinds of changes.

The main reason software changes throughout its lifecycle is that the business process in which it is used evolves over time. Some domains evolve faster than others, and it is not possible to build in flexibility to accommodate arbitrary changes cheaply. Thus, deciding too early, most likely incurs risk issues, which it will be too costly to have it changed later on.

**Principle 5: Deliver Fast**

In the software development industry, in order to achieve high quality, teams have to "slow down and be careful". However, when an industry imposes a compromise like that on its customers, the company that breaks the compromise stands to gain a significant competitive advantage. As a consequence, a fast-moving development team must have excellent reflexes and a disciplined, stop-the-line culture. The reason for this is clear: development teams are not able to sustain high speed unless they build quality in.

The common mistake observed in software companies is in a quest for discipline. Many organizations develop detailed process plans, standardized work products, workflow documentation, and specific job descriptions. This is often pursued by a staff group that trains workers in the process and monitors conformance. The goal is to achieve a standardized, repeatable process which, among other things, makes it easy to move people between projects. However, a process designed to create interchangeable people will not produce the kind of people that are essential to make fast, flexible processes work.

In order to move fast, it is required engaged, thinking people who can be trusted to make good decisions and help each other out. In fast-moving organizations, the work is

structured so that the people doing the work know what to do without being told and are expected to solve problems and adapt to changes without permission. Lean organizations work to standards, but these standards exist because they embody the best current knowledge about how to perform the expected assignments. They form a baseline against which workers are expected to experiment to find better ways to do their job. Lean standards exist to be challenged and improved, for achieving a faster, higher quality and high value outputs.

### **Principle 6: Respect People**

In essence, respect people in software development means: respect the point of view of the people doing the work. Therefore there are three ideas of people behind this statement:

1. Entrepreneurial Leader: A company that respects its people develops good leaders and makes sure that teams have the kind of leadership that fosters engaged, thinking people and focuses their efforts on creating a great product.
2. Expert Technical Workforce: Any company that expects to maintain a competitive advantage in a specific area must develop and nurture technical expertise in that area. Wise companies make sure that appropriate technical expertise is nurtured and teams are staffed with the needed expertise to accomplish their goals.
3. Responsibility-Based Planning and Control: Respecting people means that teams are given general plans and reasonable goals and are trusted to self-organize to meet the goals. Rather than taking micro-management, respect their superior knowledge of the technical steps required in the project and allow them to make decisions.

### **Principle 7: Optimize the Whole**

The whole optimization is based on the idea that optimizing a part of a system will always, over time, sub-optimize the overall system. As already mentioned, by focusing on the entire value stream, not only the software delivery part, it dissects the whole organization as a system thinking and proposes changes to on parts that usually out of the software development scope but affects its releases. Throughout this idea, the products must be complete in a sense that they are created to solve customers' problems. Therefore, the focus must be on the overall project and organization, by optimizing everything as a whole.

The important practice in optimizing the whole is measurements. Rather than measuring effects of failing code or local team measurements, the metrics should focus on the overall business value, which is the reason the organization exist.

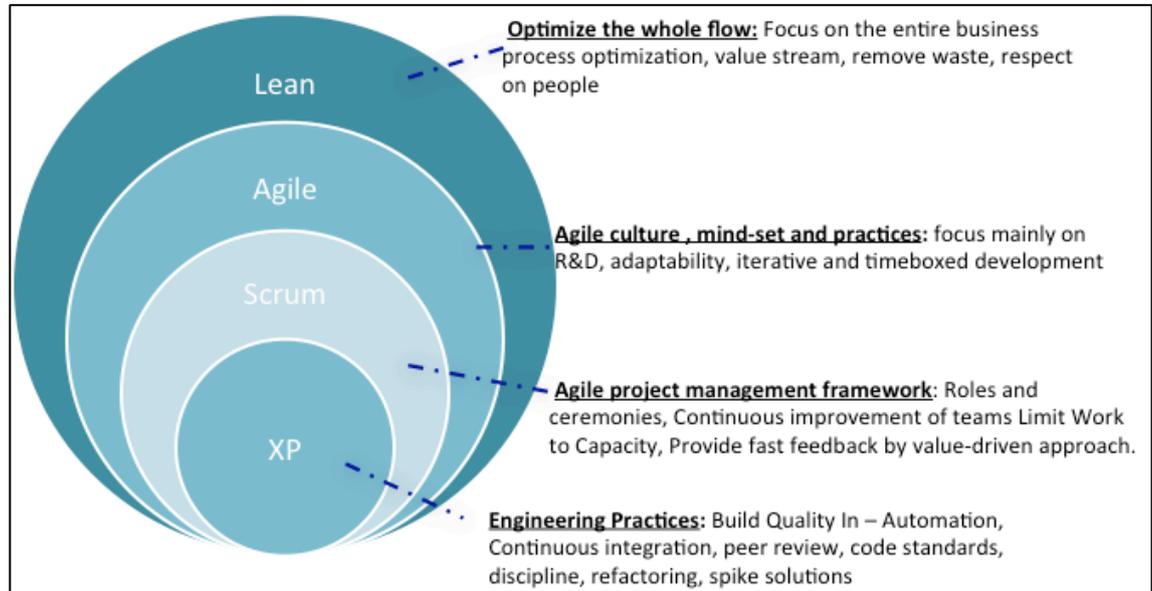


Figure 5: Visual explanation of how lean and agile methods relate to each other as complementary methods as well as to the enterprise.

In conclusion, Lean Software development provides a complete toolset of principles and practices that complement the implementation cycles and give hints how teams employ the customer value added approach in practice. Lean focuses on the whole of the organization changes in order to make it faster and same time with higher level of quality and people engagement. In this sense, changes outscope the team and embrace changes in a wider range of practices in the entire organization. These methods are the foundation of how agile can be scaled to the enterprise by providing guidelines towards people behavior: focus on value stream by reducing waste as well as targeting increased customer value added. Figure 5 depicts this information by making a relationship between methods and respective practices.

### 3.3.4 Kanban

Kanban is a new technique for managing a software development process in a highly efficient way. Kanban is based on Toyota's "just-in-time" (JIT) production system. Although producing software is a creative activity and therefore different to mass-

producing cars, the underlying mechanism for managing the production line can still be applied. Basically, a software development process can be thought of as a pipeline with feature requests entering one end and improved software emerging from the other end. Inside the pipeline, there will be some kind of process which could range from an informal ad hoc process to a highly formal phased process. Thus, this pipeline is restricted by the amount of software it delivers, the throughput. A bottleneck in a pipeline restricts its flow and most likely the processed work will be queued in some point of the pipeline. Therefore, Kanban is a tool to enable the identification and capable to reach the optimal flow of work within this pipeline. (Klipp, 2013) Therefore, Kanban utilizes three basic rules to reveal bottlenecks in the development flow: (1) visualize the workflow; (2) limit the work in progress; (3) measure and improve flow.

The visual representation of the workflow of the development process allows teams to organize themselves by tracking items under work and implementing along all phases of the process. The more complex a process is, the more useful and important creating visual workflow becomes. In practice, the visual representation is implemented by first mapping the production flow or the development phases the software product undergoes before it is delivered. The important details to note are how the working items flow in this map, how they get prioritized as well as the reasons of such decisions and how they get assigned. The bottom line is to describe and identify how items get completed in this flow.

The Kanban board is the task board where the flow is represented. It employs a low-tech, high-touch approach by creating a simple method for easy engagement of all stakeholders to manipulate, understand and easily follow up the progress of projects using such tools. By adopting this primitive technique, teams avoid a tool-related data accuracy perception and allow more people to update the plans as appropriate for the reality of the project. Along with the board, the tasks are added and represent the value added contributions to the final product.

Whether a project is simple or complex or whether a team is small or large, there is an optimal amount of work that can be in the pipeline at one time without sacrificing efficiency. In another words, there is a significant amount or work that can be at same time that prevents overproduction and reveals bottlenecks. This is called work in progress (WIP), and setting limits of WIP in each of the steps in the flow has two major benefits: (1) It reduces the time to get any one thing done (reduces the lead time); (2) It improves quality by giving greater focus to fewer tasks. Thus, the WIP limits are also

part of the Kanban boards included in each of the phases of the flow by making all stakeholders aware of the limiting numbers.

Kanban works as a “pull” system. The term comes from the idea that one stage of the process pulls work from the previous stage signaling it is ready for the next batch. This approach limits WIP, as opposed to a push system, where each stage works as quickly as possible and then pushes work to the next task, no matter how much WIP already exists. In more general terms, pull means that when someone is ready to do work, they look at the board to see what needs to be done, and they pull their next task into the column representing the next step in the process.

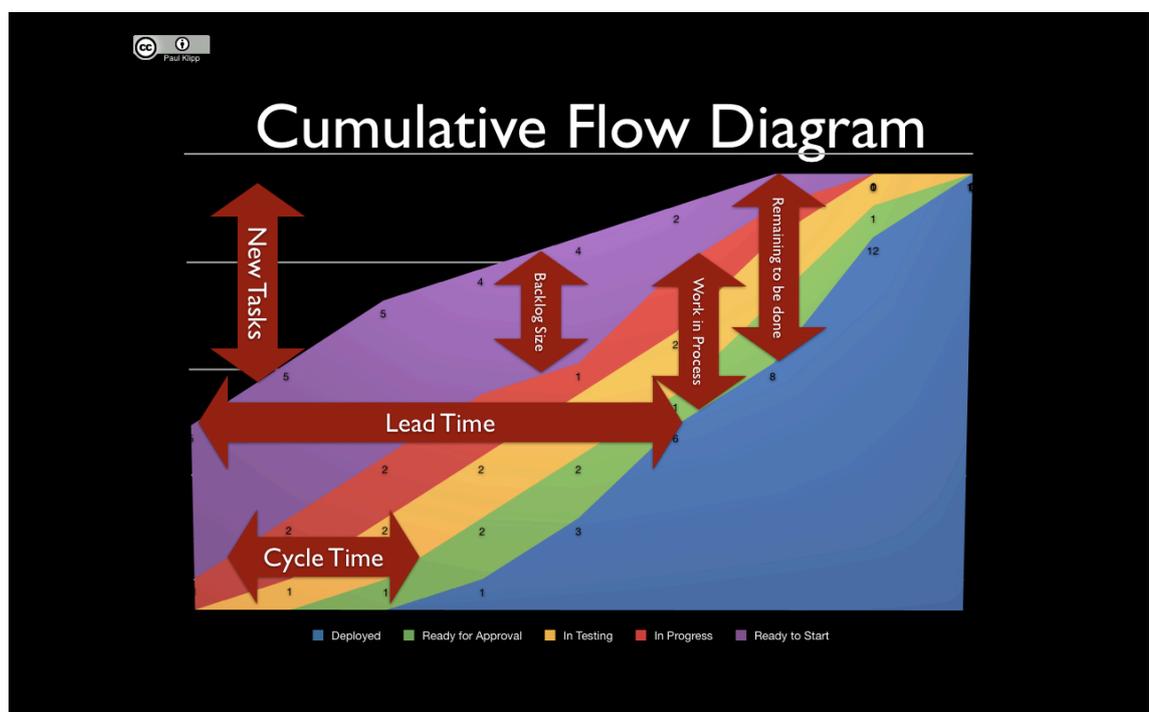


Figure 6: Kanban metrics calculated through Cumulative Flow Diagram. Copied from (Klipp, 2013)

Improvement should always be based on objective measurements, and Kanban follows the same approach. The measurements of lead-time and cycle time indicate how effective and predictable the process is towards customers needs. It makes easy to identify bottlenecks and give suggestions how to have them sorted out and to plan the overall workload in the process. In order to make the measurements clear, lead-time is basically how long it takes to get something done from the time someone asks for it until they receive it. Cycle time is how long it takes someone to finish a task once they have started it. Figure 6 shows a cumulative flow diagram of the measurements Kanban can

provide. Usually, the smooth flow from left to right indicates the WIPs are being followed and the process provides a predictive throughput. However, spikes in what represents a bottleneck are causes excessive WIP in a specific phase. (Klipp, 2013)

### 3.4 Package-Based Systems

In software, a package management system, also called package manager, is a collection of software tools to automate the process of installing, upgrading, configuring, and removing software packages for a computer's operating system in a consistent manner. It typically maintains a database of software dependencies and version information to prevent software mismatches and missing prerequisites.

Packages are distributions of software, applications and data. Packages also contain metadata, such as the software's name, description of its purpose, version number, vendor, checksum, and a list of dependencies necessary for the software to run properly. Upon installation, metadata is stored in a local package database. (Wikipedia, 2013.)

Software Packages and Package Managers were initially utilized on Linux-based systems in order to supply, along with Linux Kernel, a large suite of accompanying software to complete the various tasks for which users typically employ Linux. Thus, Linux distributions were born to achieve such requests, including several thousands of applications; suites and configurations bundled into packages. This situation required an effective way of managing packages for building a productive working environment, providing ways to deliver such packages on different Linux distributions and upgrading transparently installed packages with new versions.

Nowadays, many Operating Systems and large software applications benefit from package management systems. Making use of package management systems, several advantages can be identified, such as:

- Reduced compilation time. It is faster and less complex to compile a single package, seen as a small chunk of the whole system, instead of a monolithic system comprised by thousand files and configurations.

- It is easier to port single packages instead of large systems. This advantage is observed on Linux distributions, which today there are several of them in the market. Distribution developers can build, maintain and distribute different packages for different distributions only recompiling and rebuilding the packages under such environments. In this sense, users of these systems work with binary packages by skipping the complexity of bootstrapping and recompiling the whole application under each environment.
- It helps to scale development and to maintain system architecture simple along with high visibility to the whole project. Developers focus on the development and maintenance of packages under their domain, taking care only of the interfaces to external dependencies such as libraries or frameworks. It increases software reusability by decreasing development time.
- Package management systems turn the task to upgrade, update, install and remove software from the system simpler and faster rather than dealing with several files and their interdependencies in separate.
- Dependencies between packages are easier to identify and manage by increasing the architecture visibility. Package Management Systems have tools specially developed to build and organize such dependencies which, in turn, assist the systems upgrades and new installations automatically, identifying missing or outdated dependencies.
- Packages group several files together and add flexibility to include configuration operations before, during and after packages' installation reducing the complexity of dealing with these files separate.
- It makes it easier to understand and manage component new versions. In several circumstances, once the dependencies are automatically calculated during package installation, the versioning complexity can be hidden from users.

#### 3.4.1 Debian Packages

*Debian* GNU/Linux, which includes the *GNU* OS tools and the Linux kernel, is a popular and influential Linux distribution. It is distributed with access to repositories (as seen

in section 3.4.3) containing thousands of software packages ready for installation and use. Debian is known for relatively strict adherence to the philosophies of free software as well as using collaborative software development and testing processes. (Raymond, 1999). Debian can be as well used on a variety of hardware, from laptops and desktops to phones, and servers. It focuses on stability and security and is used as a base for many other distributions. (Wikipedia, 2013).

There are two types of *Debian* packages. The source packages and the binaries generated from the build process of a *Debian* source package. Regularly users install into their systems the binary packages, which are files ending in `.deb`, while source packages end in `.dsc`.

Source packages provide all of the necessary files to compile or otherwise, build the desired piece of software. A source package is comprised of:

- Main file, the `pkg_debver.dsc`: Meta-data file that contains the name of the package, dependencies to build, and checksums. The Appendix XX depicts a `.dsc` file example.
- `pkg_ver.orig.tar.gz`: The upstream source, which means the original source code from a version control system where the software is originally maintained (where there is no distribution specific changes). An upstream code is the “unflavored” version of the application and creating the *Debian* package out of it means “debianising”, i.e. converting a regular source code package into one formatted according to the requirements of the *Debian* Policy.
- `pkg_debver.debian.tar.gz` : tarball with the *Debian* changes made to upstream source, plus all the files created for the *Debian* package. These changes indicate the specific porting to *Debian* system, making it possible to compile, build the binary package and fetch the respective dependencies to be able to be executed under the target *Debian* system.

In the description above, `pkg` means the application name from where the package name is referred. `debver` is the *Debian* version which designates the versioning under *Debian* Systems, while `ver`, means the original upstream version. The difference be-

tween two versions is mainly to differentiate the changes made in both systems that might follow different release schedules and bug fixes.

*Debian* System provides tools to compile and build the source package into one or several *Debian* packages to specific hardware architectures. The build process aims to:

1. Identify and install build dependencies required to be present in the system to make possible the proper compilation of the source files and their respective configuration.
2. Compile source files, by building the installation structure and generating binary code.
3. Execute automated checks in order to find bugs and policy violations. It contains automated checks for many aspects of *Debian* policy as well as some checks for common errors.
4. Generate the binary package, the `.deb` file and a `.changes` file. The `.changes` file describes all changes made in `debian/changelog` and `debian/control` so that when the package is uploaded to a package repository it designates a newer version of it is included for repository re-indexing purposes. Afterward, the package is ready to be installed in the target system.

The standard format of a binary package is `package_version-revision_arch.deb`. In the name structure, replace the *package* part with the package name, the *version* part with the upstream version, the *revision* part with the *Debian* revision, and the *arch* part with the package hardware architecture, as defined in the *Debian* Policy Manual. (Debian, 2012) Also, the *Debian* Policy requires every package to provide copyright and change log information. The copyright file is supposed to contain all the necessary licensing information, as well as the author, the location where to obtain the source, and ideally the maintainer who packaged it, and when.

Therefore, the basic structure of a binary package is split into two sections: the files to be installed under Linux directories and the control files to guide installation by requesting further dependencies and configuring the application during the installation.

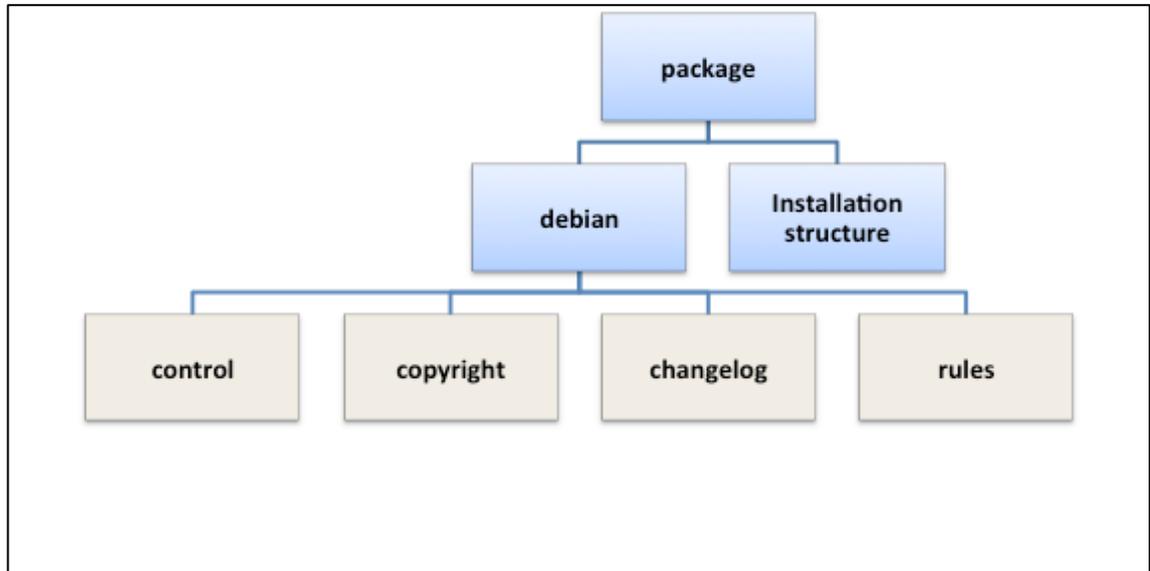


Figure 7: Binary package basic structure depicting the *debian* directory files

The required files in *debian* directory under the binary package are described as follows:

- control: describes source and binary packages names, dependencies to build and to install the package as well other metadata information required by Debian Policy.
- copyright: contains information about the copyright and license of the upstream sources.
- changelog: has a special format used by programs to obtain the version number, revision, distribution, and urgency of the package. It is also important, since it is where all changes are documented for each package version.
- rules: specifies how to build the package following certain procedures called “targets” likewise in `Makefile`. For example, the *clean* target calls scripts that will clean all compiled, generated, and useless files in the build-tree and the *build* target will build the source into compiled programs and formatted docu-

ments in the build-tree. There are several other targets possible to be added into `rules` file, but they are not included in the scope of this work.

### 3.4.2 Package Dependencies

*Debian* system is armed with powerful tools that assist users to effectively manage packages installation. The set of programs and tools comprise the so-called “Advanced Package Tool or APT. APT, which basically resolves problems of dependencies and retrieves the requested packages, works with another tools, which handle the actual installation and removal of packages (applications). When APT is used to retrieve packages from repositories, it builds a dependency resolution tree and identifies which packages and respective versions available have to be installed in order to fulfil packages dependencies.

As stated in section 3.4.1, the `control` file specifies the dependencies a package needs to be installed as well as built. Dependencies can be defined as strict which package absolutely requires another package to function correctly after installation. Another type of dependency is defined as recommended, where the presence of such packages is not mandatory, but useful if installed in the system. The *pre-depends* is also another way to define a dependency between packages. It is actually reserved to cases when the system demands a package to be fully installed and configured before attempting to install the actual package. Therefore, the “*pre-depends*” package must be installed in a separate procedure, before installing the actual package.

An important dependency relationship is defined as *provides*. It is related to Virtual Packages and its intrinsic definition. Sometimes, there are several packages that offer more-or-less the same functionality. In this case, it is useful to define a *virtual package* whose name describes that common functionality. (The virtual packages only exist logically, not physically and this is the reason why they are called *virtual*.) The packages with this particular function will then *provide* the virtual package. Thus, any other package requiring that function can simply *depend* on the virtual package without having to specify all possible packages individually. The *provides* statement is also defined as part of the `control` file by creating the virtual package name on the *Provides* field. (Debian, 2012.)

### 3.4.3 Package Repositories

In order to have software packages retrieved and installed on major distributions, an archive called Package Repository is used to hold these packages. *Debian* Package Repositories are managed by APT that handle installation, upgrades and removal of packages in a distribution as well as dependencies resolution under the repository. Thus, APT fetches the required packages to install in a system from the repository, downloads and adds the installation result into a local database evidencing it was successfully installed.

Essentially, APT repositories consist of directory hierarchy with Debian source and binary packages in it. Frequently, repositories tools provide ways to group or classify packages for security or convenience reasons under the directory structures. APT repositories are indexed in order to provide ways for querying package versions and interdependencies. Therefore, it is possible to retrieve all versions of the same package and its dependencies using tools for managing such repositories. To summarize, package repositories are used as a single storage database to store packages for better handling the operations of installing, updating and removing packages from a system.

On package-based development, repositories are used as the final destination, or even as maturity stages of developed binary packages, hence, used also as the source for building the final product. This strategy enables the usage multiple version control system tools by development teams. This approach does not enforce distributed development teams to rely on a single version control tool or even being forced to commit to the project mainline. In fact, the project mainline runs under the repository where the tested, inspected and accepted packages are stored.

In this perspective, package repositories become part of Continuous Integration process using packages stored in the repository as source for building other packages via dependencies or to build the final product by retrieving and installing packages in a deployment environment. In a package-based development, repositories help to scale Continuous Integration as well as scaling agility in large software development projects.

#### 3.4.4 Other Linux Package-Based Systems

There are several other Package Based systems including also other Linux initiatives such as *Fedora* distribution. The packaging system and repository management tools in *Fedora* are different than *Debian*, but in essence, their targets are closely interrelated. It therefore uses the RPM (RPM Package Manager, a recursive acronym) as package management system and binary packages end in `.rpm`.

The main difference between *Fedora* and *Debian* is about the environment stability and security it provides for a development and production environment. While *Fedora* focuses on the cutting edge distribution suites, which may have flaws, *Debian* targets stability and security of its releases and moreover, it advantages of a large free software community based on collaborative development processes making the portings to other platforms easier and more straightforward. Furthermore, APT in practice offers more simplicity and flexibility than RPM based tools. Thus, this is the main reason the case company has chosen *Debian* as the base of its product development.

#### 3.5 Scaling Agile Teams

As seen on agile development on section 3.2, small teams applying agile practices can create substantial improvements in productivity and customer satisfaction at the local team level. On an enterprise scale, however, the challenge of achieving the full benefits of agile is significant, and CEOs, VPs and other sponsors should recognize the likelihood that serious changes in the existing organization must be addressed.

As the enterprise grows, organizational patterns, policies and procedures grow with it. Substantially, many of them run directly the opposite way to the philosophies that characterize agile. Worst than that, many of these patterns resist changing while it is fundamental to achieve creativity and productivity. Moreover, when it comes scaling agile methodologies to the enterprise there are two classes of challenge that must be addressed. First, the challenges inherent in agile itself, which limit the methodology, may impose to scaling itself, and second, those imposed by the enterprise, seen as impediments will otherwise prevent the successful application of the new methods.

The focus of this study is to provide ways to succeed achieving results in the first class, while the second is covered as required changes in the organizations governance for

successfully adopting agile. Without reasonably achieving such transformations it is too difficult to overcome the agile limitations to large-scale projects.

### 3.5.1 Defining Scalability

Ambler and Lines (2012, 4) define scalability of an organization by a set of factors that an agile team may face. This includes team size, geographical distribution, organizational distribution (people working for different groups or companies), regulatory compliance, cultural or organizational complexity, technical complexity and Enterprise disciplines. The scalability is addressed through the full delivery lifecycle by adding appropriate lean governance to balance self-organized teams and risk-driven viewpoint to the value-driven approach. Ambler and Lines (2012, 19) also note that risk-value driven approach as an extension of value-driven approach, once it focus on a strategy that reduces delivery risk, by producing potentially usable products on a regular basis. In this perspective, the requirements in the teams backlogs that are of highest value from the perspective of the stakeholders but also consider features related to risk as high priority items, not just high-value features.

In this sense, Ambler and Lines, (2012, 40) propose a hybrid process framework that builds upon the solid foundation of many core agile methods previously discussed on chapter 3.2. Therefore, the project teams will adopt and tailor these complementary practices to the scope and complexity of the scaled project and organization. The main points of scalability are defined below:

- Agile Architecture
- Feature Teams
- Coordination
- Continuous Integration practices
- Release Planning

### 3.5.2 Agile Architecture

Larman and Vodde (2010, 288) state that agile architecture comes from the behavior of agile architecting — hands-on programmer architects, a culture of excellence in code, an emphasis on pair programming coaching for high-quality code/design, agile modeling design workshops, test-driven development and refactoring, and other hands-on-the-code behaviors. This statement, therefore, questions the role of system architects in traditional organizations that slowly lose touch with the reality of source code so high up and abstracted from the code (real system) leveraging the whole architecture in degradation over time. In this sense, the skillful way to achieve good architecture is mainly having it emerging from the teams, the ones who own the actual software code.

The most important practice towards this idea is to adopt joint design workshops along with all involved feature teams (see section 3.5.3). It means involving all people with skills in programming, system engineering, architecture, testing, UI design, database design, and so forth in the initial phase of implementation of a new feature. It may involve modeling everything related to the upcoming goals and overall system architecture. Larman and Vodde (2010, 292) note that the intention of design workshops is actually to encourage teams to have a conversation — to explore and discuss together and come to a shared understanding about designs and requirements, to help develop a shared mental model, and learn together. This conversation while sketching influences ongoing discussions on each iteration to evolve the design according to the feedback of what actually has been coded, tested, integrated and agreed with stakeholders.

### 3.5.3 Feature Teams

According to Larman and Vodde (2010, 549) a feature team is a long-lived, cross-functional, cross-component team that completes many end-to-end customer features — one by one.

The characteristics of a feature team are:

- Long-lived teams: the team stays together so that they can succeed for higher performance by taking on new features over time.

- Cross-functional and cross-component teams dealing with vertically features under the product scope
- Work on a complete customer-centric feature, across all components and disciplines (analysis, programming, testing, architecture)
- Composed of generalizing specialists
- Size, typically in Scrum, from 7 to 10 team members

However, Larman and Vodde (2010, 550) point out that features are not randomly distributed over the feature teams. The current knowledge and skills of a team are factored into the decision of which team works on which features. It means that each of the feature teams has a specific backlog of items belonging to the features they are responsible for under the specific domains of their current specialized domain. This is mainly because feature teams exploit speed benefits from specialization, but when requirements do not map to the skills of the teams, learning is forced, breaking the overspecialization constraint. Therefore, feature teams balance specialization and flexibility.

In this sense, Larman and Vodde (2010, 550) also observe that continuous integration is essential when adopting feature teams. Continuous integration facilitates shared code ownership, which is a necessity when multiple teams work at the same time on the same components.

#### 3.5.4 Coordination

Large-scale agile organizations focus on cross-functional and cross-department teams synchronizing through departments interfaces based on agreements and commitments focused on maintaining visibility. The coordinator of this department teams is often part of a product owners team responsible to work with other product owners in the product organization to establish the prioritization of the common product backlog. He does not have final decision-making power nor is he involved in the development itself; his focus is purely on cross-department synchronization. In this sense, the functional departments do not exist by increasing the cross-functional responsibility inside department teams, as stated by Larman and Vodde (2010, 190).

In such structure, teams must be aware of their context and actively manage their boundaries by undertakings, such as synchronizing work on a shared code or clarifying cross-team requirements. Scrum Masters, in this regard, are not the project managers and in fact facilitate coordination by reminding the teams' focus is the overall product and thus ensuring team members of different teams know each other and have practices and means in place to synchronize their work.

Much of the coordination adopted in scaled agile teams is mainly to promote and facilitate communication between teams. According to Larman and Vodde (2010, 200) coordination techniques can be classified into centralized and decentralized approaches. The centralized techniques consist of meetings in which people from multiple teams assemble to coordinate their work while decentralized strategies let teams figure out their dependencies by themselves and expect them to resolve these in a decentralized, networked manner.

Specially concerning centralized coordination the Scrum of Scrums meetings focus on having representatives of different teams to discuss and explore cross-team topics. It is not intended for only Scrum Masters but open to every scrum team member that realizes the importance of getting aware of coordination issues among every other team.

Towards decentralized coordination, the Community of Practice (CoP) intends to improve the organization a whole by creating groups of volunteers to tackle important and specialized issues within the organization such as test driven development, Continuous integration, architecture or even discussing about coordination and communication itself.

Another important aspect that directly affects coordination and consequently communication among cross-functional teams is the multisite environments, a common impacting issue when dealing with scaled organizations. One way to establish and improve coordination among those sites is to create a virtual shared space using tools such as wikis, forums, IRC (Internet Relay Chat) or instant messages.

### 3.5.5 Continuous Integration Practices

An important statement for this work comes from Larman and Vodde (2010, 351) who mention that Continuous integration (CI) is essential for scaling lean and agile development:

Our conclusion is that there is no inherent reason why Continuous Integration and Automated Build processes won't scale to any size team. In fact... [they] become more essential than ever.

With CI, developers gradually grow a stable system by working in small batches and short cycles—a lean theme. This enables teams to work on shared code and increases the visibility into the development and quality of the system. This idea can be also evolved in a way that when a project starts to scale, it is easy to be deceived into thinking that the team is practicing continuous integration just because all of the tools are set up and running. If developers do not have the discipline to integrate their changes on a regular basis or to maintain the integration environment in a good working order they are not practicing continuous integration.

Larman and Vodde (2010, 361) still note to have a CI system scaled is the first action point to fully automate build and test procedures. These are mandatory steps to achieve full benefits from the lean practices and promote CI principles such as speed of integration as well as speed of feedback cycle including in this scope the automated tests execution. However, there are obstacles for scaling CI System. Once there are more people producing more code and tests, the probability of breaking the build increases with more people checking in code. Also, an increase in code size leads to a slower build and thus a slower CI feedback loop.

The solution is to (1) speed up the build (and tests execution) and; (2) implement a multi-stage CI system. For speeding up the build there are several solutions: Add hardware and parallelize build by creating a build farm that is a collection of several servers set up to compile and build source code remotely. Another approach is to handle the components as packages, so that only changed packages need to be recompiled. This approach also requires that dependencies to be well managed, improving the overall structure of the product.

The second option is to implement a multi-stage CI. Multi-stage continuous integration takes advantage of a basic unifying pattern of software development: software moves

in stages from a state of immaturity to a state of maturity, and the work is broken down into logical units performed by interdependent teams that integrate the different parts together over time. What changes among projects is the number of stages, the number and size of teams, and the structure of the team interdependencies. (Wikipedia, 2013.)

A multi-stage CI system splits the build and executes it in different feedback cycles. At the lowest level, it has a very fast CI build containing unit tests and some functional tests. When this CI build succeeds, it triggers a higher-level build, containing slower system-level tests. Larger products have more stages.

Also, according to Larman and Vodde (2010, 353) standard CI has the ultimate goal to “stop and fix” as early as possible when a defect is detected. The first priority is to fix the defect and corresponding root cause sharing the *jidoka* lean concept. However, despite the fact this attitude being absolutely needed, it doesn’t mean that all the work should blindly stop. On large, cross-functional and distributed teams, adopting multi-stage CI is still high priority to identify problems early and act on them. Nevertheless, avoid affecting all other participants, allowing them to proceed towards their integration targets. Consequently, only if the problem turns out to be really serious, does the integration flow have to be stopped.

Therefore, Larman and Vodde (2010, 364) point out a few elements to take into account when building a multi-stage CI system:

- A developer build: Developers should be able to run unit tests for subsets of the system before checking-in the code.
- Component or feature focus: A traditional multi-stage CI system is structured around components. The lowest level builds one component, the next level a subsystem, and the highest level builds the whole product. Alternatively, CI system can be organized around components and the output of this triggers multiple-feature CI systems running higher-level acceptance tests in parallel.
- Automatic or manual promotion: A higher-level CI system needs to be triggered by an announcement that the component can be used. Such an announcement is called a promotion and is done by labeling (or tagging) the component. Promoting a component can be done automatically or manually. With automatic promotion the lower-level CI system promotes a component after it passed the

intended the automated test suites. Manual promotion is when the team decides when the component is “good enough” and promotes it. During manual promotion, it may also include manual acceptance and validation tests over the promoting components.

- Event or time triggers: Every CI system is triggered by either an event or by time. The low-level CI systems are always triggered by an event — a code check-in. For higher-level CIs, the trigger is either the promotion of a component or time.
- The number of stages: The size and complexity of the product, the number of teams and developers sharing the code base as well as the amount of features determine the number of stages. Stages a CI system has, the slower is the integration process. Many of the below stages are not required. It will highly depend on how teams self-organize towards the features implementation. Larman and Vodde (2010, 366) define the following stages:
  - *Fast component-level* — a very fast low-level CI system for quick feedback. It runs unit tests, code coverage, static analysis and complexity measurements.
  - *Slow component-level* — a slower low-level CI system. It runs integration or slow component-level tests.
  - *Product stability-level* — a very fast product-level CI system for quick feedback on the basic product stability. It runs fast functional tests (smoke-tests).
  - *Feature-level* — a slower high-level CI system. It runs functional and acceptance tests.
  - *System-level* — a slow high-level CI system. It runs system-level tests, which often take hours.
  - *Stability-performance-level* — a very slow high-level CI system. It continuously runs stability and performance tests, which often take days, if not weeks.

On the other hand, Gruver et al. (2013, 55) groups these stages into three levels of testing:

1. **Integration testing**, where pre-commit and commit tests are automatically executed running unit level tests and smoke tests on top of components (small grouping of code base coming out from development teams) and features. Autorevert is possible at this stage once it is reported integration failures.
2. **Stability testing**, which is intended as a quick feedback loop to find broad-based failures from new commits in as narrow a commit window as feasible.
3. **Regression testing**, which is full regression test suite of all automated tests. It kicks off at midnight daily and provides complete view of the quality of the system.

Larman and Vodde (2010, 367) still highlight the importance of CI systems to include visual management (a lean principle). When the build breaks, a visual signal indicates failure, fostering investigation of integration problems and creating a solid quality system to avoid errors to propagate to the final product.

In conclusion, CI provides a systematic enterprise-level solution for ensuring the fundamental agile paradigm of always having the code base stable and ready for release. This actually requires carefully thinking through deployment pipeline and the stages of automated testing.

### 3.5.6 Release Planning

As already mentioned on section 3.5.3, Leffingwell (2011, 301) also reinforces that teams and activities are organized around an ongoing series of incremental releases. The releases may be internal and used for evaluating the system as a whole or may be made external, in that they are made generally available to customers or major stakeholders.

Leffingwell (2011, 303) also points out that releases need to build on top of common rules applied to all teams with the purpose of driving strategic alignment and therefore institutionalize the product development flow. These rules enforce the teams to produce towards the product global targets to a common direction as well as to achieve cadence through periodic planning and synchronization through continuous integration.

Primarily, releases are planned ranking the release objectives by business value and consequently aligned with teams' commitment to deliver such objectives. Teams therefore, break the business objectives (usually seen as epics) into features and stories by applying the agreed cadence by planning their releases in a timeboxed fashion. It is also important to note that teams work together also to identify their interdependencies by laying stories into the iterations available in the release scope.

The results of release planning are used to update the product Roadmap, which provides a sense of how the enterprise hopes to deliver increasing value over time. The Roadmap consists of a series of planned release dates, each of which has epics, a set of objectives, and a prioritized feature set. The Roadmap, then, represents the enterprise's current "intent" for the next and future releases. However, it is subject to change as development facts, business priorities, and customers may change.

In many organizations, Leffingwell (2011, 71) mentions about the Release Management Team in addition to agile teams once, even though empowered, the agile teams do not necessarily have the requisite visibility, quality assurance, or release governance authority to decide when and how the solution should be delivered to the end users. Members of this team may include key stakeholders of the Program level of the enterprise, and this team meets frequently to address issues like to make sure teams still understand their goals and what is being built as well as tracking the statuses of the current release and impediments to be facilitated.

#### **4 Case Company**

Nokia Corporation (Finnish: Nokia Oyj) is a Finnish multinational communications and information technology corporation that is headquartered in *Keilaniemi*, Espoo, Finland. Its principal products are mobile telephones and portable IT devices. It also offers Internet services including applications, games, music, media and messaging, and free-of-charge digital map information and navigation services through its wholly owned subsidiary *Navteq*.

Nokia has around 97,798 employees across 120 countries, sales in more than 150 countries and annual revenues of around €30 billion. It is the world's second-largest mobile phone maker by 2012 unit sales (after *Samsung*), with a global market share of 22.5% in the first quarter of that year. Nokia was the world's largest vendor of mobile

phones from 1998 to 2012. However, over the past five years it has suffered a declining market share as a result of the growing use of smartphones from other vendors, principally the *Apple iPhone* and devices running on *Google's Android* operating system.

Since February 2011, Nokia has had a strategic partnership with Microsoft, as part of which all Nokia smartphones will incorporate Microsoft's Windows Phone operating system (replacing *Symbian* and *MeeGo*). Nokia unveiled its first Windows Phone handsets, the *Lumia 710 and 800*, in October 2011. (Wikipedia, 2013.)

#### 4.1 History

Over the past 150 years, Nokia has evolved from a riverside paper mill in southwestern Finland into a global telecommunications company. During that time, Nokia has made rubber boots, car tires, generated electricity and even manufactured TVs.

Nokia history splits into three phases: Industrial conglomerate, Networking Equipment and Mobile Phones and Internet Services. The first phase, began in 1865 when it was founded as a paper mill company manufacturing paper in two Finnish Cities, Tampere and Nokia. Close to the 20<sup>th</sup> Century, Nokia added to its business activity along with rubber manufacturing products such as galoshes, the production of cables like telephone, telegraph and electrical cables. By the end of the Second World War the two business units were merged into a new industrial conglomerate. The new company was involved in many industries, at one time producing paper products, car and bicycle tires, footwear (including rubber boots), communications cables, televisions and other consumer electronics.

In the 1970s, in the second phase, Nokia became more involved in the telecommunications industry by developing a digital switch for telephone exchanges, which became the workhorse of the network equipment division. Later in 80's Nokia was one of the key developers of GSM (Global System for Mobile Communications), the second-generation mobile technology that could carry data as well as voice traffic. NMT (Nordic Mobile Telephony), the world's first mobile telephony standard that enabled international roaming, provided valuable experience for Nokia for its close participation in developing GSM, which was adopted in 1987 as the new European standard for digital mobile technology.

In the late 1980s and early 1990s, the corporation ran into serious financial problems, a major reason being its heavy losses by the television manufacturing division and businesses that were just too diverse. Therefore, probably the most important strategic change in Nokia's history was made in 1992, however, when the new CEO *Jorma Ollila* made a crucial strategic decision to concentrate solely on telecommunications. Thus, during the rest of the 1990s, the rubber, cable and consumer electronics divisions were gradually sold as Nokia continued to divest itself of all of its non-telecommunications businesses. By 1998, Nokia's focus on telecommunications and its early investment in GSM technologies had made the company the world's largest mobile phone manufacturer, a position it would hold for the next 14 consecutive years until 2012. Between 1996 and 2001, Nokia's turnover increased from 6.5 billion euros to 31 billion euros. Logistics continues to be one of Nokia's major advantages over its rivals, along with greater economies of scale.

In February 2011, Nokia announced it was joining forces with Microsoft to strengthen its position in the smartphone market. The strategic partnership sees Nokia smartphones adopting the new Windows 7 operating system, with the Symbian platform gradually being side-lined. (Nokia, 2013.)

#### 4.2 Mobile Phones Market

Nowadays, modern mobile phones support a wide variety of other services such as text messaging, MMS, email, Internet access, short-range wireless communications (infrared, Bluetooth), business applications, gaming and photography. Mobile phones that offer these and more general computing capabilities are referred to as smartphones.

Smartphones are mobile electronic devices capable of running an advanced operating system that is open to installing new applications, are always connected to the Internet and provide very diverse functionality to the consumer. Smartphone manufacturers make contracts with the network service providers for exclusivity of certain phones, and the providers in turn subsidize the cost of the smartphone for the consumer. Every smartphone user must purchase service with their smartphone, or the value of the smartphone is significantly diminished.

The smartphone device itself is made up of two primary parts: the hardware (consisting of the screen, processor, memory, keyboard (if it has one), radio, packaging, etc.), and the software that runs on the hardware. While every smartphone firm is involved in the design and manufacture of the hardware of their phones, they are not necessarily heavily involved in the primary development of the software that runs on their phone.

In this sense, smartphone operating systems come in three forms: proprietary, licensable, and open source. Smartphone manufacturers strategically chose which OS model to follow based on their core strengths. Open source OS's give the smartphone manufacturer access to an existing operating system that is free, and freely customizable. The most popular open source OS is *Android*, but others include *Symbian* OS and *MeeGo*.

Software applications, or "apps," are a significant part of the smartphone market today. Every smartphone operating system has an online store where apps can be purchased and downloaded to the smartphone to extend the functionality of the smartphone. These purchases can be made directly from the phone, and include very diverse functionality: games, over-the-internet radio, exercise trackers, maps and GPS navigation, note taking, word processing, etc. Apps constitute a complementary product to the smartphone and are a source of revenue for the company that runs the app store.

In general the smartphone market is rapidly changing, with constant product introductions. It is characterized by quickly evolving technology and designs, short product life cycles, aggressive pricing, rapid imitation of product and technological advancements, a highly price sensitive consumers. No one vendor in the market has sufficient market share to control prices, resulting in strong rivalry and competitive pricing. Although the technological innovation is rapid in the smartphone market, any edge a particular firm might obtain, whether it is technological or industrial, is diminished by rapid imitation. With some exceptions, smartphones from most manufacturers at any given time have relative feature parity in many respects, making substitution relatively easy from the consumer's perspective. As a result, smartphone manufacturers compete heavily on price, and small changes in price often result in increases in sales. (Cromar, 2010, 5-31.)

In conclusion, mobile operating systems and the ecosystems evolving around them are shaping the way forward for the mobile industry because of their potential to attract users and drive new business opportunities in data services. *Stephen Elop*, current

Nokia CEO, has brilliantly summarized nowadays smartphones market in order to justify his decision towards Windows Phone disruptive strategy (Ahonen, 2011).:

"The battle of devices has now become a war of ecosystems, where ecosystems include not only the hardware and software of the device, but developers, applications, ecommerce, advertising, search, social applications, location-based services, unified communications and many other things. Our competitors aren't taking our market share with devices; they are taking our market share with an entire ecosystem. This means we're going to have to decide how we either build, catalyze or join an ecosystem."

#### 4.3 MeeGo Organization

In order to introduce MeeGo organization, since 2005, a very small group of engineers with limited resources at Nokia was developing a Linux based *Maemo* operating system and devices based on it. The team was known as OSSO (Open Source Software Operations) and the goal was to produce a product that would change the world. The OSSO team was renamed as *Maemo* team in 2007, and as a consequence of Nokia's and Intel's partnership in 2010, it was renamed as the *MeeGo* team. Thus, at the Mobile World Congress in Barcelona in February 2010, Nokia and Intel announced that they would combine their Linux-based operating systems in development into a new joint project; *MeeGo*.

Joint with *MeeGo* initiative, Nokia decided to continue developing *Maemo* 6, codename *Harmattan*, and to make it as compatible with *MeeGo* as possible. In this sense, *Harmattan* was supposed to act as a bridge between *Maemo* and *MeeGo*, which was being developed in cooperation with Intel. (Kurri, 2012.) The result from *Harmattan* development was a sole product, Nokia N9.

The scope of this thesis is to focus on *Harmattan* teams scalability, agile structure and collaboration while pushing efforts to accomplish N9 launch. Additionally, as the central role, fastening all teams, the Release & Integration unit with the utmost responsibility to deliver and customize the final product, support program management team with error management tools and processes as well as release planning and deliverables follow up.

The Release & Integration entity was comprised of the following teams:

- Build and Integration Team: Made up of developers in charge of implementing and maintaining the build, continuous integration and configuration tools broadly used by the whole MeeGo organization with the aim to have software delivered and integrated into the product. The people in this team were also supporting in various integration tasks developers would require as well as defining policies and enforcing them throughout automation.
- Test Automation Team: Developers responsible for building and implementing the test automation environment executed along with Continuous Integration System.
- Information Technology (IT) Support Team: The main scope of IT is to maintain, upgrade, scale and secure the build farm infrastructure as well as the Version Control Systems maintained under *MeeGo*.
- Configuration and Customization Team: This team is responsible to manage the several configurations and variants as part of product delivery to external entities at Nokia such as Product Managers, Customer Account Management team and Local Sales Offices. It also works close to System Architects in order to establish proper dependencies and core assets as part of product configurations. It respectively defines policies and procedures on how secondary configurations are created and maintained by Configuration Owners.
- Error Management Team: People responsible for defining the policies and process regarding bugs and requirements assignment, resolution and traceability. This team is also accountable for bug and requirement tracking tools known as Defect Tracking Systems and how asset teams should act towards the tool and processes best practices.
- Release Planning Team: The intent of release managers is to plan and follow the execution of epics and features that would build the product. This team negotiates with development teams among the assets priorities and mandatory requirements. It has a key role in supporting the program management team in order to align the deadlines of all asset teams in each release.

- System Integration Validation Team: Testers responsible for validating the daily releases and several asset promotions executing manual smoke, and exploratory tests validating releases are sanely build and assuring test automation process was sustainable.

The empirical part of this thesis aims to detail the collaboration of Release & Integration team along with several other asset development teams under *MeeGo*, to emphasize the Multi-Stage Continuous Integration architecture and practices as well as procedures of how teams scaled an agile and lean environment in order to successfully deliver the target product.

## **5 Leveraging Scalability throughout Multi-Stage Continuous Integration**

Although this thesis presents common ideas as the ones previously mentioned chapters 2-4, it introduces a new scaled agile model which is focused on multi-stage continuous Integration approach where the whole organization targets to integrate their software assets bundled into packages towards to a common, single repository. In such an Integration-centric approach, teams target delivering their own releases guided through a common product release planning that encompasses the synchronized targets of all development teams.

In this perspective, teams work following their own agile and lean practices as well as Version Control Systems tools and processes. Nonetheless, they share the same practices and tools for handling requirements and errors, to integrate and release software. Hence, the Release & System Integration team takes the central role for managing integration requests and release targets for all other development teams.

Figure 8 provides a simplified understanding concerning several teams' relationship along with Release & Integration by pointing out the main processes and tools under this organizational area.

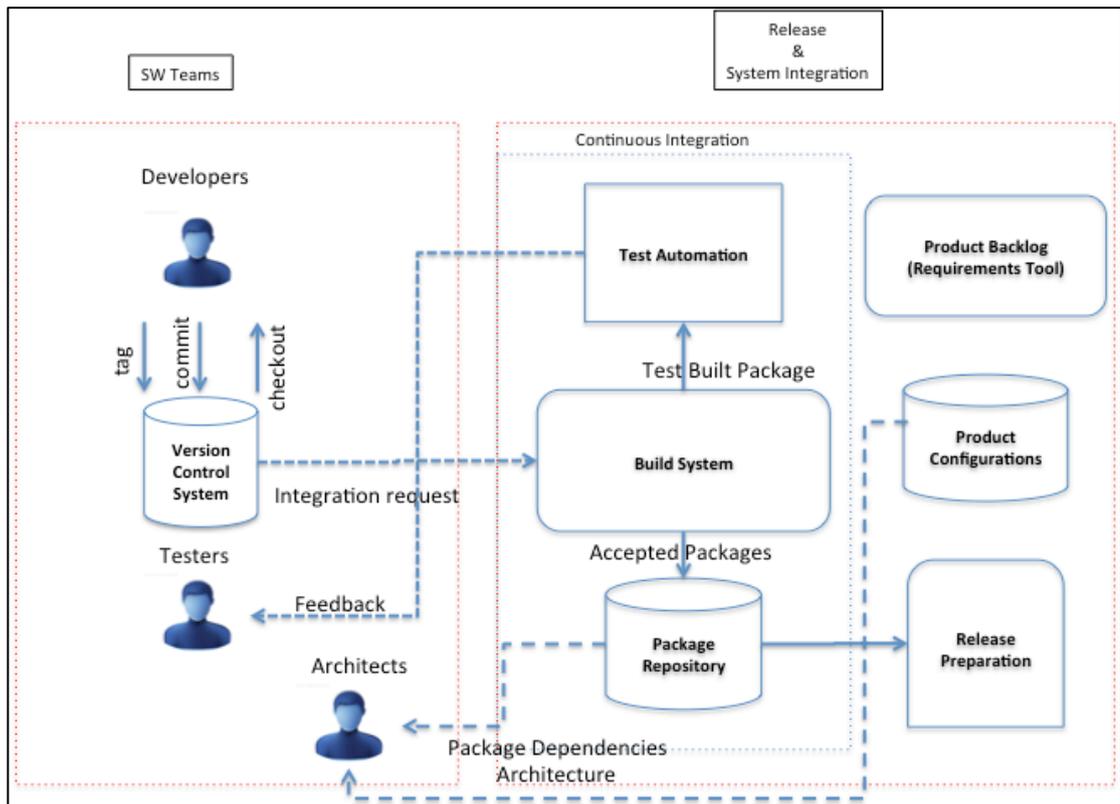


Figure 8: Systemic view of the relationship between development teams and Release & Integration.

In this environment there are several processes and practices in order to achieve the main targets that are:

- Integrate software packages by maintaining consistency in the delivery flow;
- Create and announce product weekly release by sharing a common product view and promoting transparency;
- Be able to continuously verify progress from final product by continuously building the product, several times a day, whenever needed;
- Keep risks and issues at a low and manageable level, by avoiding regressions;
- Promote agile and lean thinking practices as much as possible in the organization. Constantly learn from previous iteration mistakes and adapt fast to changes.

Therefore, these targets translate into key processes and practices listed as follows:

- Execute all tasks of a Multi-Stage Continuous Integration System;
- Manage Requirements and Errors throughout a single backlog shared for the entire organization and hooked to Continuous Integration System;
- Leverage product builds for every integration request;
- Effectively perform Agile Release Management;
- Keep product different configurations and customizations continuously evolving along with packages integration.

## 5.1 Continuous Integration Process Overview

The multi-stage continuous Integration process works as depicted in Figure 9.

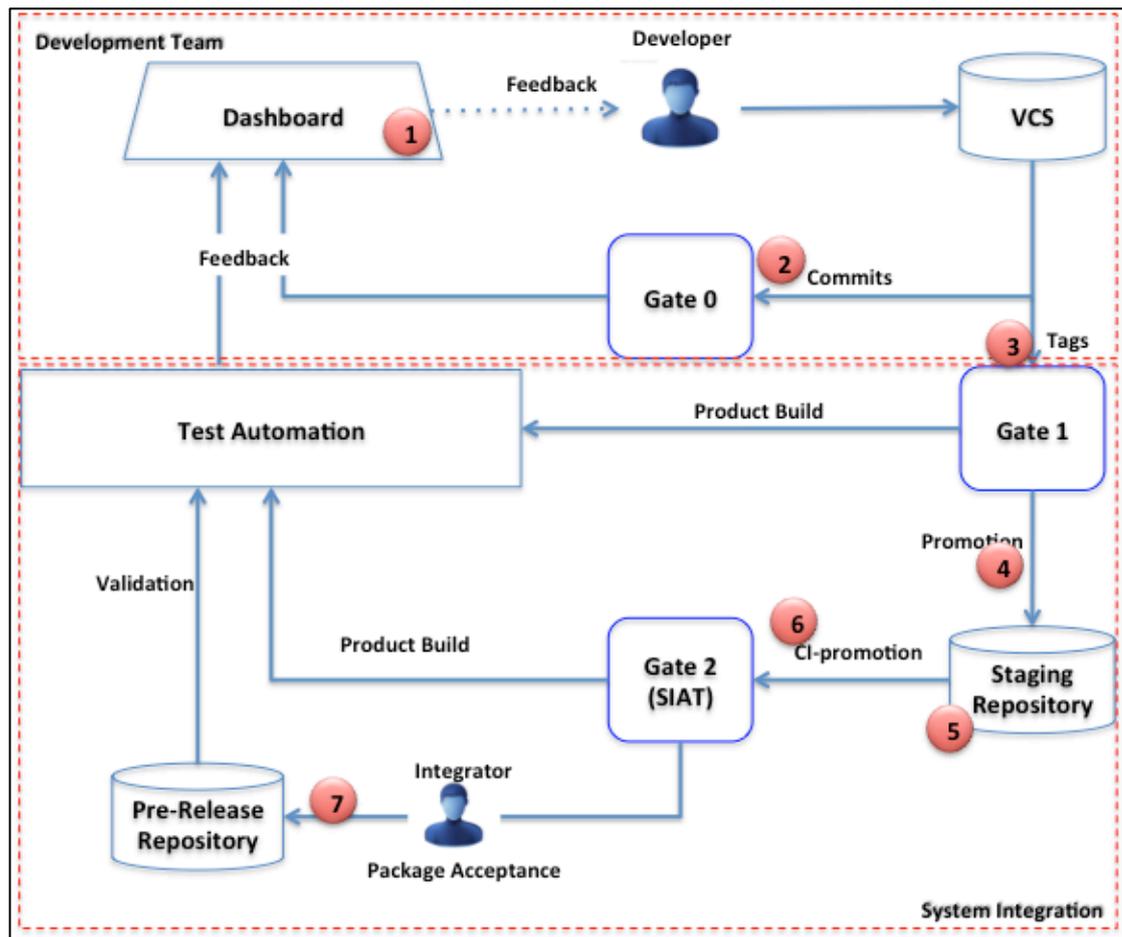


Figure 9: Continuous Integration stages and steps taken towards successful package integration.

The Continuous Integration stages are introduced in the following steps as numbered on Figure 9:

1. Dashboard set up
2. Gate 0: First Feedback loop, at team asset scope; fast feedback
3. Gate 1: Second Feedback loop, broader range of unit and integration tests executed on the product build; implemented package towards integration into main repository
4. Staging Repository promotion

5. Staging Repository: package is ready for acceptance procedure to be promoted to pre-release repository
6. Gate 2: Third Feedback loop; packages from Staging repository are grouped and automatically tested for acceptance. The CI-promotion system performs the packages' grouping according to package category previously determined by team's architect.
7. After acceptance tests, Integrator manually promotes accepted package into Pre-Release Repository. Further Validation, exploratory and more detailed tests are performed from product nightly builds generated from this repository.

The first step in this environment is part the dashboard setup by developers. The dashboard aims to create visibility of the build and test statuses of each software package monitored by the development team. The dashboard allows the follow-up of each monitored package towards its integration to Pre-Release Repository. While setting up the dashboard, developers connect the source package available in the team's version control system. This action hooks up the package source repository in the VCS to Continuous Integration System and hence any commit or tagging into source repository triggers the Continuous Integration steps for each package.

Thus, on every commit into VCS, a package build is triggered under CI system targeting the Gate 0 feedback loop. This is a local fast feedback (target is to be less than 5 minutes) which developer receives from CI system. For each package build, corresponding unit tests are also executed, providing results on the dashboard. On every tagging into VCS, the package promotion is triggered to Staging repository. The tagging practice means: submit package to production to be part of the final product.

The package is built again and added to an intermediary product build being then submitted to the test automation system. Integration tests are executed in order to identify errors or regressions concerning the addition of this new package or a newer version of it. The whole test execution is a bit longer in Gate 1 (also known as Staging Gate) than Gate 0, which has to consider the time spent on integration system to generate a product build and corresponding tests execution.

If errors are found, the feedback system will report to the developer throughout the dashboard and the CI System reject the failed packages. However, if tests successfully pass, package will automatically be promoted to the Staging Repository being ready Acceptance for the Tests phase.

The ci-promoter is a monitoring tool that runs automatically every 5 minutes on packages that have reached the staging repository. It takes all packages flagged as promotable, the ones that passed Gate 1 tests, and group mandatory flagged packages to build a new product. As well as Gate 1, the product build is submitted to Test Automation System for acceptance tests execution.

Packages that fail tests during acceptance phase are manually removed from the group by promotion maintainer and another product build is manually triggered. An error is raised to the responsible team informing the corresponding package has failed the acceptance phase. Therefore, a newer fixed version of the package needs to be resubmitted to CI for approval. Packages that pass the acceptance phase are manually moved to the Pre-Release Repository by the integrator.

By the end of the day, a nightly product build is created from the Pre-Release repository by, among other objectives, providing product progress. The resulted nightly build reports which packages have been successfully promoted on the previous day, new bugs found and providing the latest product version for further validation and testing for all involved teams in the organization.

## 5.2 Continuous Integration Architecture

In order to support a large-scale development environment with several teams committing the code to the central repository in parallel to the CI infrastructure, it has to be scaled in order to be capable of providing required CI practices such as fast feedback and test-automation covering all the submitted code.

The CI architecture is built on top of several dozens of build servers working independently known as build farm. Each of the build servers has a schedule task controller that executes the build automation procedure on every few minutes. The build procedure starts by picking up build requests submitted by development teams that are queued waiting for building. Thus, each of the builder can pick several build requests

up to its calculated capacity and the whole farm increases this capacity to process several hundreds of requests in parallel.

These requests comprise manual or automated submissions of code towards continuous integration system. In this sense, for a package to become part of the product repository, a request containing details of the package such as package VCS information is created and submitted to the build queue. Each request can be monitored throughout its state after submission as indicated in Figure 10:

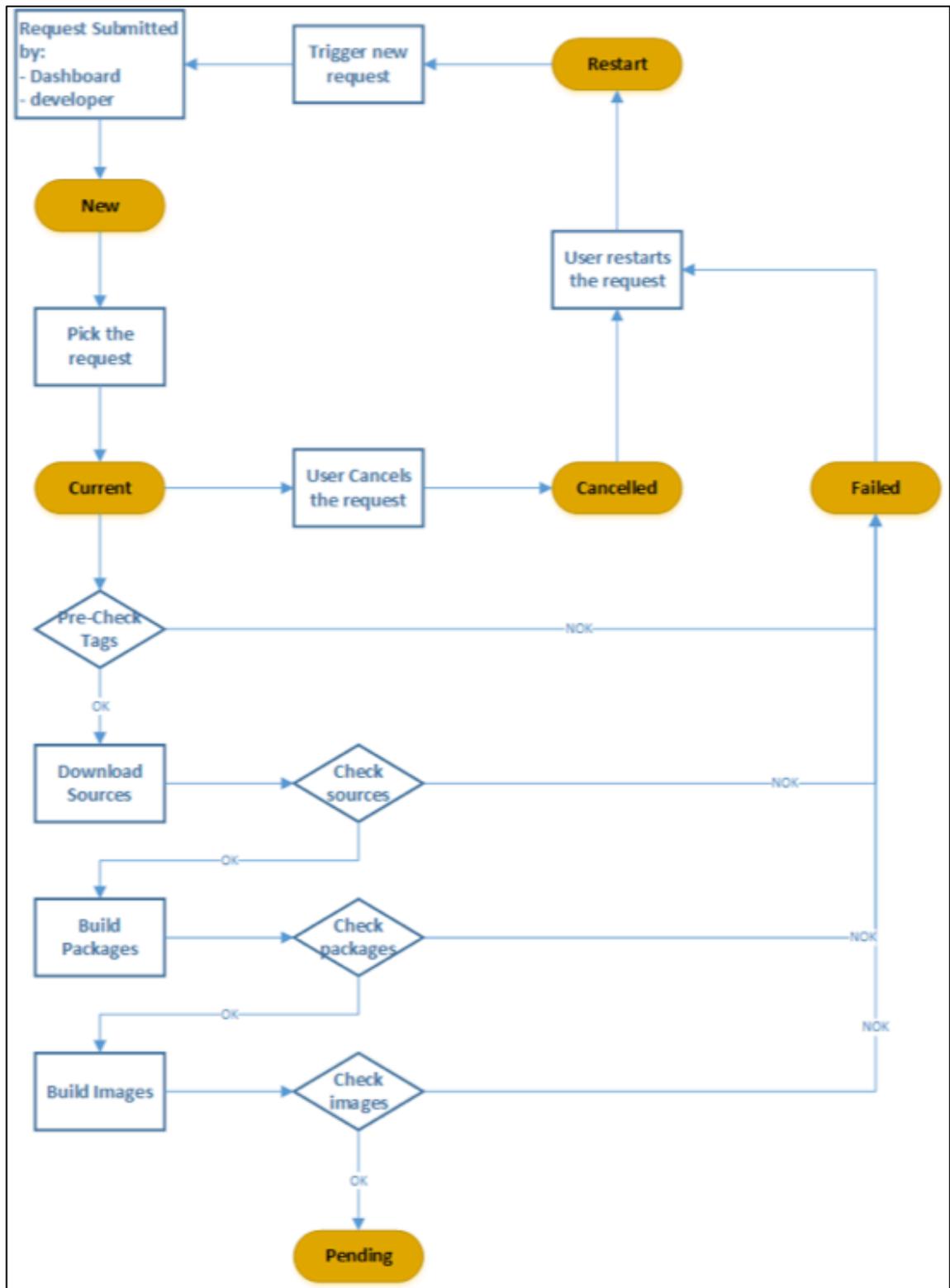


Figure 10: The process of handling build requests either submitted manually by developers or automatically when the source code is monitored in developers' dashboard

As depicted in Figure 10, the builder has the sole responsibility to:

1. Perform sanity checks into the submitted packages such as version check and several other rules automated to assure package build consistency in the product.
2. Download sources from VCS.
3. Compile the source code by setting up a clean build environment and generates the binary package.
4. Generate the product build image.

As also shown in Figure 10, requests can be submitted by mainly:

- Through Web Tool front end, where developers can visually follow-up the request build states through the same tool;
- Through dashboard where developers setup the packages they want to have watched for submitting automatic requests either through VCS commits or tags. Behind the dashboard there is a mechanism called *VCS-runner* that watches every commit or tag into VCS and then submits a request automatically. *VCS-runner* is designed to hook up on each source code added by the developers in the dashboard and starts tracking commits and tags in such code in order to trigger automatic requests into build system.

In this sense, developers usually setup the dashboard and start development. On each commit, the fast feedback cycle is executed by having the *vcs-runner* fetching the commits from VCS and trigger new build requests towards CI. Tagging represents the intention of the developer to have the source code integrated following the full CI cycle. Once requests are automatically created, they are queued waiting for one of builders of the build farm to have it digested. At this point is important to note the value stream of the CI while queuing such requests and carefully analyze it to remove waste. In this sense, it is important to observe the attention of Release & Integration team to keep the queue as lower as possible all the time to avoid delays and bottlenecks in the development flow. The cheapest way to maintain the request queue size low is to add more hardware in the build farm and in consequence increasing the frequency the builders were capable to grab new requests to build. Another alternative is to keep

packages architecture simple with minimum dependencies as possible and therefore fast to build.

After the build is finished the resulting state of the request is updated (pending or failed) as indicated in Figure 11. Figure 11 shows the complete architecture of the build system:

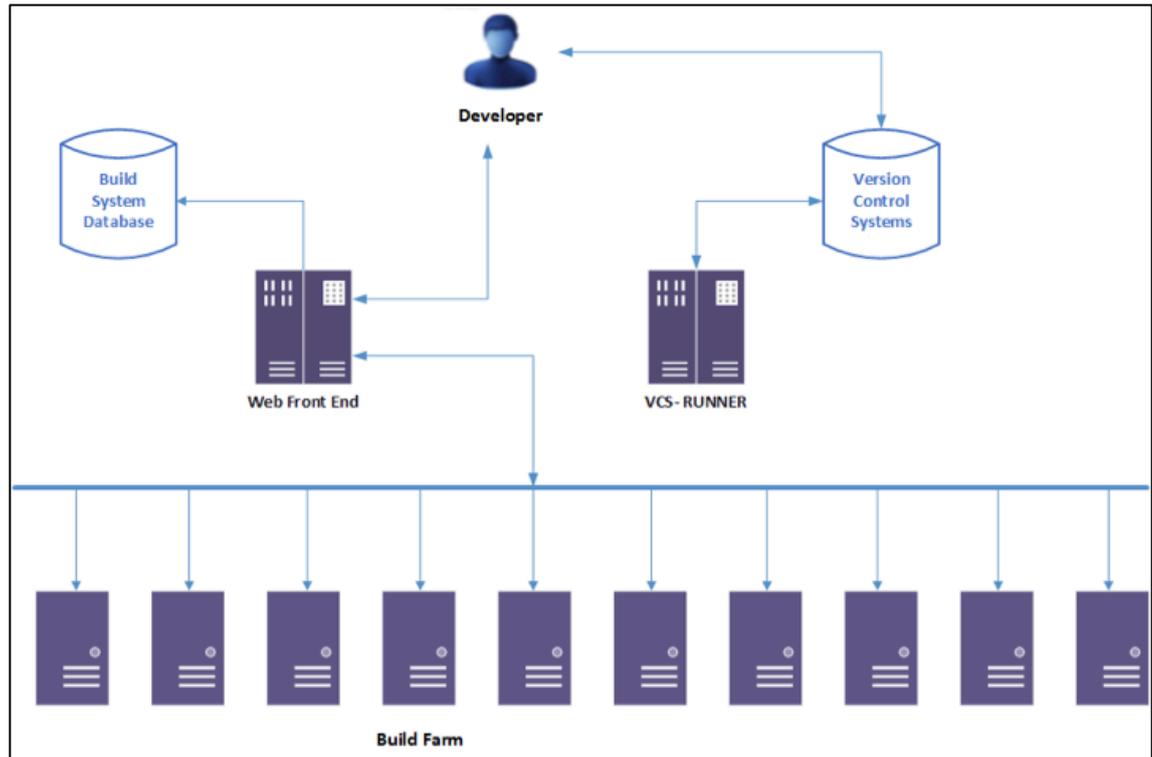


Figure 11: Build System Infrastructure architecture

Along with build system, it is important to highlight how the case company has implemented test automation as part of CI process. Before detailing such implementation agile testing underlies on Test Driven Development (TDD) process. With TDD, Crispin and Gregory (2009, 5) define that the developer writes a test for a tiny bit of functionality, sees it fail, writes code that make it pass, and then moves on to the next tiny bit of functionality. In the same perspective, developers also cover integration tests code to make sure the small units of code work together as intended. The test not only validates at a confirmatory level that the code works as it is expected it also effectively specifies the work in detail on a just-in-time (JIT) basis. In order to have this schema working smoothly, developer must have immediate feedback from the test execution they are evolving.

However, when scaling TDD to enterprise level the fast feedback target commonly turns difficult to reach at all levels. The support of multi-level CI enables such scalability by also deploying levels of complexity into the automated tests executed when teams try to integrate their source code. As indicated on chapters 2-4, the case organization has split test automation into four levels:

1. Unit Tests at team level, achieving fast feedback towards developer code (Gate 0)
2. Feature and Component Tests (along with Unit Tests) executed in Gate 1 when achieving features and components integration along with several teams.
3. Acceptance and System Integration level tests executed in Gate 2 during promotion to Pre-Release Repository.
4. Validation and Exploratory Tests executed on daily release creation after packages reach pre-release repository assuring the whole product implementation is being sanely created and following up pre established release plan.

However, regardless of the test scope, tests are created in a similar fashion also encapsulated into packages defining a common set of rules to have tests executed in a standard and consistent way while in Continuous Integration. In this sense, the test package has some special features that support its execution and proper coverage.

The first element is a test definition XML file that defines the automated test plan including:

- A hierarchical structure of tests such as: suite, set and case. The suite is the collection of tests to be executed under a specific domain that defines a set of target features to be tested. Each feature contains one or more case that is the unit of testing defining what needs to be tested or which testing scripts or functions to call during test execution.
- Information about the test scope referred to what feature or sub-feature the test cases are being created.

- The test type concerning the viewpoint or the quality purpose the test has been created. Thus, it can be classified as Unit, Functional, System, Performance, Security or Benchmark.
- The information to which level the test belongs to such as Component, Feature, System or Product respectively.
- The execution instructions defined under step element. A test step corresponds to a single operation executed in the test environment either on test device hardware or on an emulated environment. Each test step specifies a command line that is executed in the corresponding environment. A special test step is defined under pre\_steps and post\_steps, which are executed before or after the actual test cases in a test set. These sections are responsible for common environment initialization and cleanup for the test cases inside a test set.

Figure 12 depicts the Test Definition XML file as described above.

```
<?xml version="1.0" encoding="UTF-8"?>
<testdefinition version="1.0">
  <suite timeout="90" name="mwts-dataflow-cita" domain="AutomatedTesting">
    <set feature="Bluetooth" name="Bluetooth">
      <pre_steps>
        <step>mkdir -p /tmp/min_confs ; mv /etc/min.d/mwts-*.conf /tmp/min_confs/</step>
        <step>btup=`hciconfig hci0|awk '/UP / { printf "1" }' ; if [ x"$btup" == x1 ]; then touch /tmp/
mwts_dataflow_cita_bt_was_up; fi</step>
        <step>hciconfig hci0 up</step>
      </pre_steps>
      <case name="FUTE-BT-Basic_lookup">
        <step>source /tmp/session_bus_address.user; MWTS_LOG_PRINT=1 MWTS_DEBUG=1 min -x /
usr/lib/min/scripter.so:/usr/lib/min/mwts-bluetooth.cfg -c -t FUTE-BT-Basic_lookup</step>
        <get>
          <file>/var/log/tests/FUTE-BT-Basic_lookup.log</file>
          <file>/var/log/tests/FUTE-BT-Basic_lookup.result</file>
        </get>
      </case>
      <post_steps>
        <step>if [ ! -f /tmp/mwts_dataflow_cita_bt_was_up ]; then hciconfig hci0 down ; fi</step>
        <step>rm -f /tmp/mwts_dataflow_cita_bt_was_up</step>
        <step>mv /tmp/min_confs/* /etc/min.d/ ; rmdir /tmp/min_confs</step>
      </post_steps>
    </set>
  </suite>
</testdefinition>
```

Figure 12: XML Test Definition example defining automated test cases for Bluetooth feature.

Along with the XML file, the `debian/control` file of the test package must contain two important meta-data pieces of information for the CI System:

- Which functional packages the test package is aimed to cover. It uses the `CI-Packages` keyword to list binary package covered by this test package.
- At what stage of CI the testing should be performed. This includes the steps discussed above: (1) fast [for fast feedback]; (2) staging [for component or feature test]; (3) Acceptance; (4) Validation. It uses the `CI-Stage` keyword to list the stages the test package is activated.

Once test package is ready to be integrated and executed containing mandatory meta-data as explained above, the developer can either locally use test packages along with local development or integrate tests making them part of CI System. For the second option, whenever functionality is integrated under CI, the system identifies which tests belong to this functionality and execute them appropriately according to what is defined into test package meta-data.

The system under CI responsible to execute tests is the Continuous Integration Test Automation (CITA). This system is receives test requests from CI build process in order to trigger different sorts of tests and also provide a reporting functionality in a way to provide just-in-time feedback to developers. The communication between CI and CITA are synchronous in a sense that each build request in CI may trigger (according to developer's targets) a test request to CITA keeping the build request in CI on hold while tests are executed.

There are two tests environments under CITA: One that emulates the product execution focused on backend and non-functional features while another on a target-hardware which User Interface tests are executed.

Under CITA, the definition XML file is rendered and the system starts following the test execution case by case storing the results to be later posted in the build request, after test execution is finished. According to the automated execution results CITA informs CI System whether tests passed or failed by also failing the build request in a case of test errors were found.

Figure 13 highlights the cooperative communication between CI and CITA.

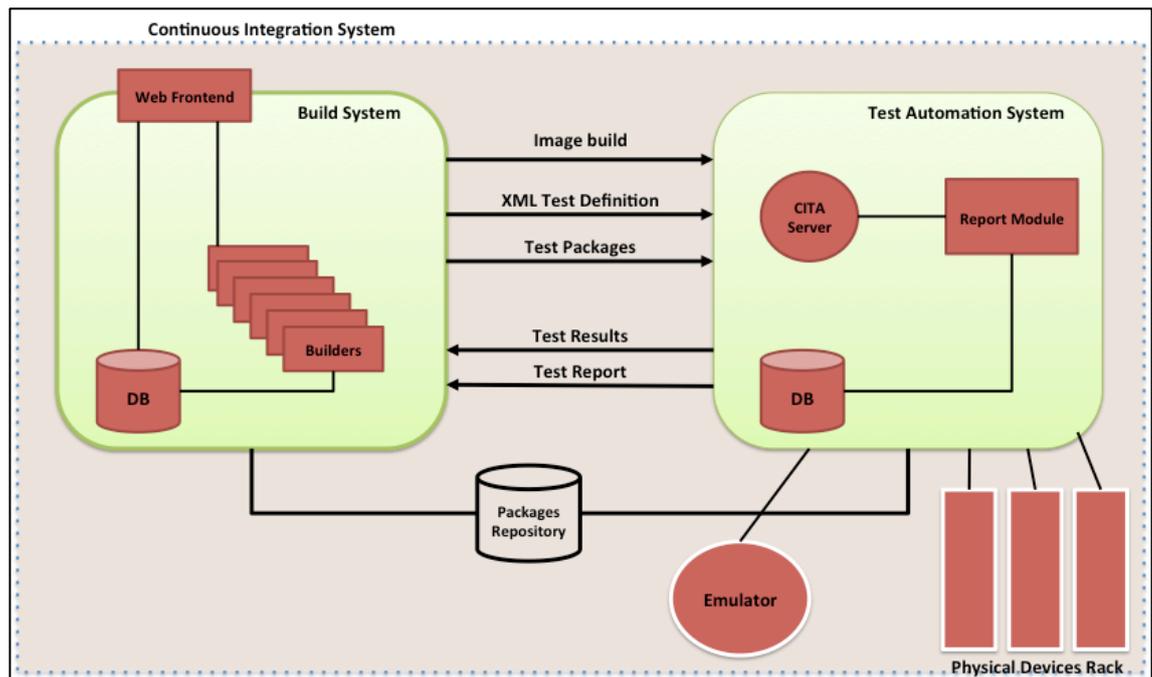


Figure 13: Interfaces between Build System and Test Automation System making up the core part of Continuous Integration architecture

Figure 13 depicts the information exchanged between Build System and Test Automation System (CITA). In this sense:

- Build System first sends build request information to CITA. It includes the image build based on the confirmation chosen to create the image and the options to CITA perform the test execution such as emulator test or physical device test more focused on UI test cases. The request also includes test packages that may be stored, after build, in the request itself or retrieved from repository. Along with test packages the XML test definition makes part of the set of information sent to CITA as well.
- After performing the test execution CITA returns to Build System the results of each test cases in a form of web report. If tests fail execution, the request, as whole, also fails and developers can access the test report throughout the web frontend. The test results are stored in a database maintained by CITA System.

### 5.3 Requirements and Error Management

In such a scaled agile environment, every requirement, story, epic, bug, change request and operational task is considered as part of one single backlog. Each of these items is classified by the domain nature they belong as part of the product architecture and assigned to the appropriate component under a team responsibility. For example, Bluetooth domain and respective sub-components such as protocols, profiles, user interface hold all backlog information concerning the thorough implementation of Bluetooth feature in the product including stories, raised bugs and tasks. Teams' responsibility is to coordinately implement and deliver such items according to the Product Release Plan.

The main advantage to keep all backlog items under one single source of information is the improved visibility and ability to link dependencies between items under the same domain or even cross-feature domains. It offers transparency and control over trade-offs, because the relative priority of the work is clearly shown by dependencies (vertical and cross-feature) in order to provide a better understanding of the remaining work, critical issues and assistance for enhanced prioritization.

Dependencies are easily created by any team member while detailing the feature corresponding tasks or even throughout design identifying parts of the feature that would require the component to be properly delivered. Vertical dependencies are set when detailing features or stories into smaller, more tangible and estimable, pieces of work named tasks. Therefore, features rely on stories, and stories on tasks to be fully implemented. Meanwhile, cross-feature dependencies are discovered through out features or epics completeness.

In the example above, the Bluetooth user interface can be only delivered to the product when the application framework component and underlying APIs are delivered as well as the Interface layouts, which, in turn, are responsibility of other feature teams. In such scenario, component (or domain) owners are responsible to identify, communicate and set cross-feature dependencies by understanding the order they have to be implemented in accordance to release planning, product roadmap as well consequent priorities. Actually, there is no strict rule to create such dependencies. There are situations when the backlog item does not require any link to other items having a clear implementation target. The focus is to act towards requirements implementation and fixes

as pragmatically and lean as possible by delivering value rather than to create and follow formal procedures.

Bugs are discovered at any time during product development and are usually blockers bugs of one or more features. They can be assigned to any level, either straight to the feature itself, stories or even tasks. A single feature is regarded complete when all inherited dependencies and blocking bugs are also delivered to pre-release repository and verified towards to the main product. In such agile environment, which test-driven approach has a key aspect concerning product development, bugs are regarded as not an error in logic, but simply, it is a test not covered by corresponding automated test cases. At the same time, when continuous integration is capable to provide fast feedback against components stability, bugs are usually found before final component or feature integration. This helps comprehensive delivery of bug-free components. In this sense, each bug does not only lead to an issue fix, but also to the discovered opportunity to implement corresponding test cases to permanently avoid such issue to repeat.

Although all efforts are added to early discovery as well as recovery of errors, they might happen when a feature is already released and accepted. This is not only about problems in the code, yet due to the high visibility and product build availability, it is constantly verified by a large group of stakeholders responsible to accept and discuss whether the deliveries are according to their expectations. If not, anyone is allowed to file a new backlog item by adding the change request proposition to be assessed by team whether that change makes sense or not. Change Requests, valid or not, are regarded in practice as bugs. This concept removes waste of forcing teams to handle a different type of issue that in fact, in its essence, there is no much difference than a bug. It can be faced, generically, as a proposal for changing or adding something in a component or feature.

All too often, bugs are classified as regressions. It is when a bug makes an accepted feature stop functioning after a system upgrade of newer versions of underlying components. It can be introduced locally in the affected component or remotely, where a change is a part of a depending module, which breaks another feature functionality.

Regressions are seen as errors that escaped to final product acceptance and regarded as the most dangerous issues in large-scale agile development once it disrupts the lean principle of continuous delivery flow and reduce product's quality and visibility.

Moreover, if the delivery flow is producing too many regressions, it is an indication that agile and lean principles are not taken to practice and the root cause must be found and fixed, frequently by improving acceptance and other suites of tests in the integration chain. Usually, despite of the regression severity, they are all regarded as blockers and raised as the highest criticality to be fixed as soon as possible. Therefore, the message to responsible team is: stop everything and focus to fix the regression. In this sense, feature requirements and bugs share the same implementation lifecycle. Figure 14 shows the workflow states for implementing requirements and bugs as well as the acceptance stages under Continuous Integration System.

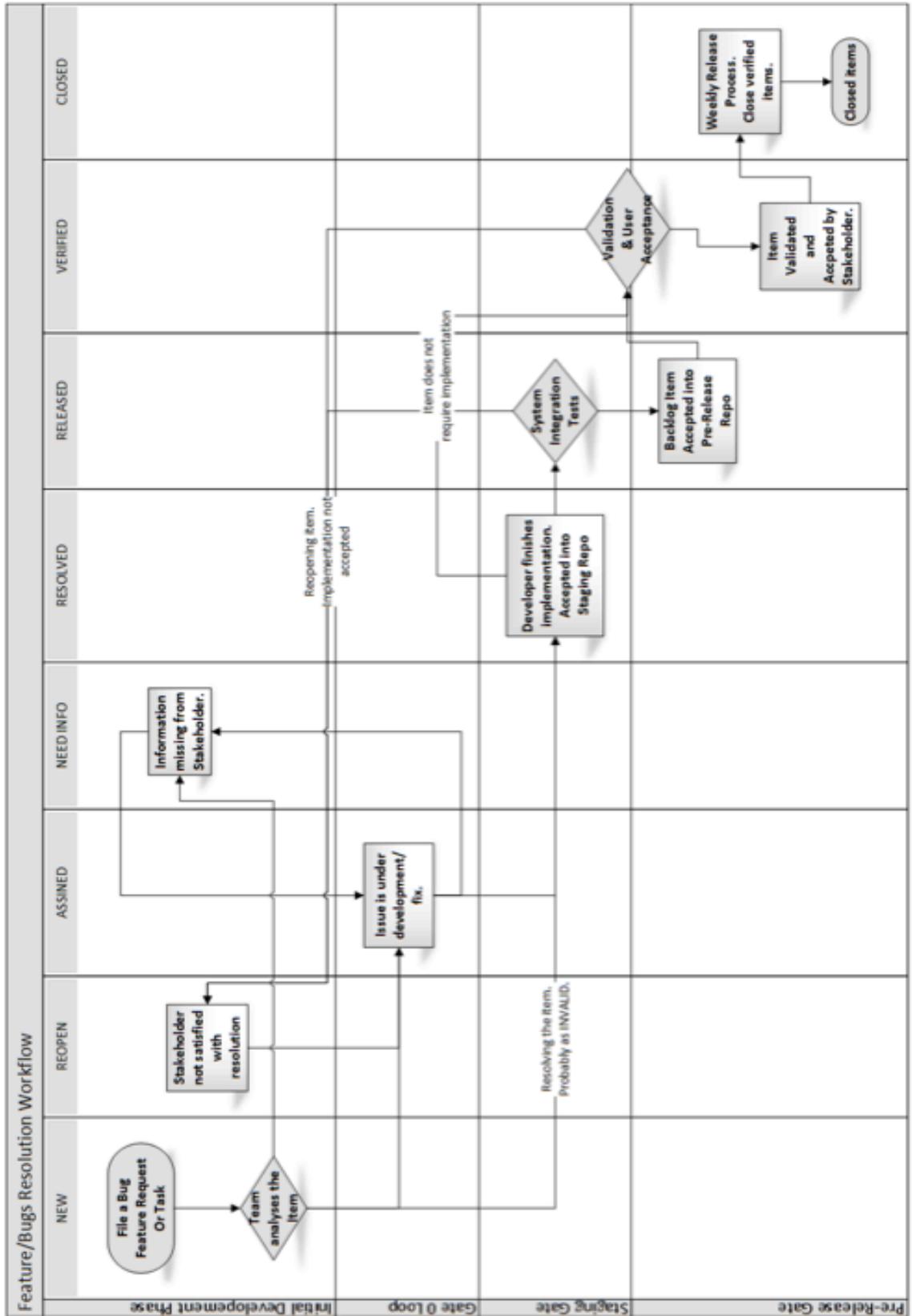


Figure 14: Cross-Functional workflow for Features and Bugs resolution under Continuous Integration System

### 5.3.1 Initial Development Phase

As mentioned on section 5.3, anyone can file requirements or bugs into backlog. The aligned practice with teams is to properly create an item under the correct domain and sub-component. There is no strict rule for such; it goes for the best product architecture understanding and teams collaboration of where to place the backlog item accurately. The hierarchy is as follows:

1. Select the **product**
2. Select the **domain**
3. Select the **sub-component** under each domain.

When creating the item, there is also some metadata that makes part of it:

Severity: It defines the urgency of how an item has to be prioritized due to the negative impact the error or lack of implementation is causing to the final product. It can be set as:

- **Blocker**. It has the highest severity. When a backlog item is set as blocker, it means, the issue is drastically impeding the delivery continuous flow of the release production. It's affecting release generation, product build creation or any of the core assets delivery, part of the project critical path. Often, software regressions are classified as blockers or critical. This is the only severity item that requires special grants to set over the backlog item. Once it's seen as a red alert flag, responsible teams ought to solve it as soon as possible due to drastic product interruption it causes.
- **Critical**. The issue causes dependencies to break in one or more asset. Usually, blocks the delivery of components from one or more teams. It requires high attention of responsible teams to have it sorted out.
- **Major**. The issue breaks dependencies of the same component and causes impediments to delivering of corresponding components.

- **Normal.** *The issue is a Non-major loss of function.* Usually reflects on documentation missing, misleading, inaccurate, or contradictory information in the documentation, but successful task completion is probable.
- **Minor.** It is classified as a minor cosmetic issue, but still worth registering either for further discussion or analysis. Indicates low priority and low value to customer.
- **New work.** The item is faced as a new enhancement request, entirely new feature or requirement. By reporting a new work, stakeholders are creating in practice change requests, which require further analysis and prioritization from affected teams. Differently than bugs, these items require more careful planning evaluating the impact, dependencies and sprint in which this implementation can be included.

As much as new work backlog items, some project managers decide to manage the workload by including not critical errors also under sprint planning and prioritize the work altogether allowing interruptions only if blocker bugs are raised. At the same time, another observed approach is to deal with bugs and new work in separate threads. Many project managers opt to have two different management strands by choosing Scrum to deal with new work items and Kanban to handle bugs and operational tasks. Different nature of work, design considerations and resolution time explain project manager decision.

It is important to note there is no pre-established defined time to have issues fixed, based on a standard Service Level Agreement (SLA). The resolution time is given upon teams collaboration and interoperability to jointly define and priority most critical issues and dependencies. Under unsolved conflict, the product program and release team is scaled to decide which item to prioritize relying on teams' feedback.

Summary: It refers to a one-sentence summary of the problem. This line is used, along with item sequential number, in package changelog and release system to identify the item itself. As good practice, teams agree to add keywords for better classification and to easy determine backlog type.

If a regression is reported, for example, it is added [REG] as initial statement of the summary, [FEA] for features, [STO] for stories and [TASK] for tasks.

*Description:* The description represents the full details of the backlog item. If a bug, the practice is to provide as much information as possible to reproduce the bug, such as:

- Software Release and Hardware version was issue has found;
- Pre-conditions;
- Setup environment;
- Steps leading to problem;
- Expected outcome;
- Actual Outcome;
- Frequency of occurrence;
- Associated test cases or test-suite that lead to bug creation.

However, if the description is associated to feature, story or task. The practice is to provide clear business functionality. See theory about Scrum and User Stories on chapter 3.3.1 for understanding how user stories are supposed to be created in an agile environment.

*Dependency:* If a backlog item cannot be implemented unless other items are fixed (depends on), or it stops other issues being fixed (blocks), their numbers are recorded into a specific field. The parent backlog can be only set to VERIFIED when all depending items are also VERIFIED. For a visual representation of dependencies see Figure 19.

Other Backlog Items Metadata:

- **Votes.** While a backlog item remains in NEW, stakeholders can vote for it as an indication of how much value and importance it has for the product. As many votes the items receive, the higher are the items priority when compared to others with fewer votes. This attribute helps product owners or

product managers to understand the importance of an issue or requirement for the feature the one maintains.

- **Keywords.** The product program team defines keywords that can be used to tag and categorize backlog items. The categorization is useful to mark items that belong to cross-feature domains like customization and localization, for instance. Therefore, it is easier to track features or bugs that require localization or customization implementation along with standard, required, feature implementation. This list can be extended using any criteria approved by product program and can be also used to track customer specific requests, by adding customer's name as keyword as well.
- **Attachments.** It is possible also to include attachments as part of backlog items, either bugs or features. It might be useful whether for problem's scope definition or stakeholder understanding. It serves as both, history for approved prototypes, tracing or logs as part of problem resolution and also for communication mechanism between teams.

### 5.3.2 Gate Zero Loop

Just before reaching Gate 0 (Zero), backlog items are, therefore, ordered into their priorities under each component. In this sense, each component holds a backlog list, ranked according to business and customer value.

In fact, requirements priority is part of the product roadmap that establishes the release planning and stories map for the product. Each release includes the features that apply to specific epics that guide the deliverables for that specific release. Thus, the first release, for example, generally focuses on the core assets implementation epics. All features that belong to the selected epics are part of the release planning (see chapter 5.6 for Release Planning). All domains and respective components affected by this plan rank and split the work into sprints, part of the release.

The prioritization activity is regularly part of continuous collaboration between teams, and strongly directed by dependencies resolution. So, teams must identify what needs to be delivered, tested and verified first in order to have other elements implemented in an order that will not break the product cohesion. Moreover, legal and certification re-

quirements are under non-negotiable features. If they are not implemented, the product will not be able to be commercialized. Global-range products as mobile devices commonly face this particularity. They have, for example, to comply with local legal requirements that change from country to country, adhere to network specificities of hundreds different carriers and have certified hardware components. These requirements are covered in more detail on section 5.5, when discussing mass customization. Nevertheless, these requirements also rely on core asset features, increasing drastically the complexity of ranking and prioritizing backlog items for large-scale products.

As mentioned in section XX, teams organize their work under Scrum and Kanban frameworks by relying on ranked list of items. Thus developers start working moving bugs or requirements from NEW to ASSIGNED. Therefore, ASSIGNED state means: under work. If questions are raised or feedback required from any of the involved stakeholders, it is moved to NEED INFO. Hence, all backlog items on NEED INFO state is impeded waiting for a clarification or blocking issue to be sorted out.

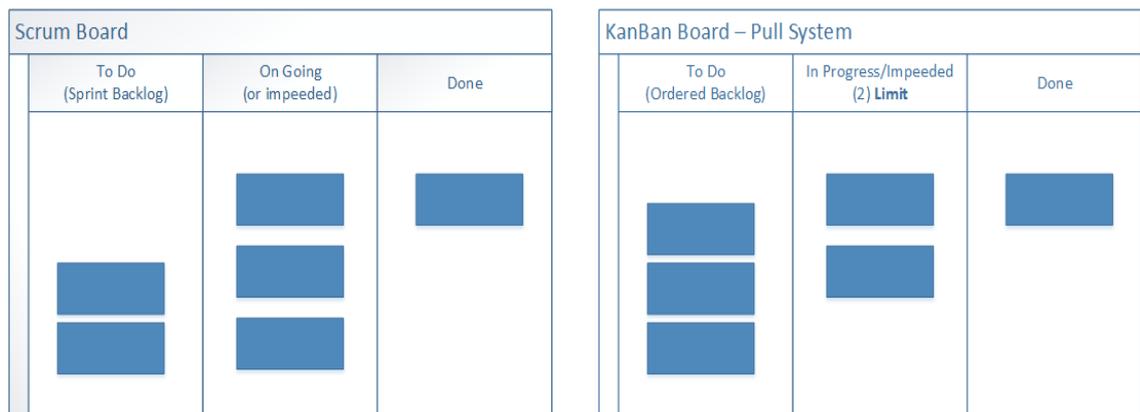


Figure 15: Scrum and Kanban boards handle differently WIP items and respective limits.

ASSIGNED and NEED INFO are regarded as Work In Process (WIP) items. WIP is the term given to work that has been started but has not yet been completed. WIP is the basic idea of Kanban, by limiting the “in progress” items and something new should be started only when an existing piece of work is delivered. Such idea assigns explicit limits to how many items may be in progress at each workflow state in every team.

Both, Scrum and Kanban approaches limit WIP, but in different ways. Scrum teams usually measure velocity – how many items get done per iteration. Once the team knows their velocity, it becomes their WIP limit. Therefore, in Scrum, all items can be at either impeded or “in progress”, differently than Kanban that explicitly constrains the

amount of work. In Scrum, though, teams understand this procedure can lead to too many on going items and evolve a culture of trying to get the current items done before starting new items. (Kniberg and Skarin, 2010, 15-17.)

Regardless the methodology, Scrum or Kanban, teams plan their own work and take on only the amount of features that the measured velocity indicates they can achieve. This forces the input rate (negotiated release objectives) to match capacity (what the teams can do in the release). The current timeboxed release prevents uncontrolled expansion of work. The global backlog pool, consisting of features and epics for the whole product, is constrained by the local WIP pools, which reflects the team's current backlog as driven by the current increment.

In this sense, there are several advantages for the business limiting WIP such as: (Griffths, 2012, 91)

- WIP consumes capital investment and delivers no return until it is converted into an accepted product. It represents money with no return.
- WIP hides bottlenecks in process that slows overall workflow (or throughput) and masks efficiency.
- WIP represents risk in the form of potential rework, since there may still be changes to items until those items have been accepted. If there is a large pool of WIP, there in turn may be a lot of scrap or expensive rework if a change is required.
- WIP Increases response time to new, higher-priority activities.

### 5.3.3 Staging Gate

While in Gate 0 loop, the developer focuses the work on mainly unit test verification throughout fast feedback under the team's component and domain in the first stage in Continuous Integration process. As mentioned in section 5.1, the software package is built on every source code commit and selected test cases are executed to make sure this stage of development is sane. When the implementation can be integrated into the

rest of the product for final acceptance, the developer follows the steps described below:

*Resolving the Backlog Item:* Set the backlog item to RESOLVED and assigns the resolution type as FIXED. Classifying the resolution as FIXED means the problem is fixed or task finished implementation.

There are other resolution types that indicate the team decision concerning the analysis done on each backlog item, either a bug or requirement:

- WONTFIX: The item is decided not to be implemented;
- DUPLICATE: There is a similar already in the backlog;
- WORKSFORME: Issue not reproducible;
- ALREADYFIXED: Issue fixed as part of another backlog item;
- INVALID: Project Manager decides to invalidate the bug after careful consideration. When invalidating an item, it can be for several reasons: out of product scope, feature has been dropped from product, the request or error is invalid in its essence.

By setting to any of these categories, the backlog reporter has to agree on the reasons given for such decision. If there is no source code implementation under corresponding item, thus the releasing phase is skipped by allowing the responsible stakeholder for the item to set the backlog item state to VERIFIED. However, if there is no agreement, the backlog record is REOPEN by stakeholder, reinitiating the whole phase.

By reopening a task or error by rejecting team's decision brings an execution impact in the planning. Moreover, making this practice a common behavior for recurring discussions of whether something is valid or not misleads the backlog proposal and worsens teams agility. Although it is recommended approach to keep all development history, decision-making and discussions under backlog items, maintain an invalid-reopen battle between teams is counterproductive and affects the continuous flow and WIP limits. Teams end up deviating the attention from the lean principle to sustainably delivering value fast to focus on win-loose conflicts to see whether an item is whether invalid or not.

Product Managers, Product Owners and Release Managers have the ultimate decision concerning an item should whether or not be implemented. This determination is part of the pre-planning phase in order keep only valid items selected for sprints or Kanban boards. It requires, once again, teams collaboration and technical expertise to assert the thoroughness of the item reinforcing the agile and lean needs in such phase of the development.

Tagging the source: The developer tags the source code into VCS by providing a coherent version to delivering package. Continuous Integration System watches the tagging and automatically submits the source to build towards Staging repository. The product build is also generated utilizing accepted packages from Pre-Release repository. The product is then submitted to test automation system where a wider range of unit and integration are executed. The developer and team keep following results from dashboard. By a successful execution of all pertinent tests, the package is automatically promoted into Staging repository. The package, in-turn, keeps waiting for ci-promoter to group more packages and execute the next stage of promotions towards to pre-release repository.

An important consideration to note in this phase is the changelog details into delivering Debian package added by the developer. More than one backlog item can be included into the changelog, which corresponds to one single version upgrade of the package. It follows the format: `Fixes: NB#<backlog item number>` as the example in Figure 16:

```
account-plugin (0.7-24) unstable; urgency=low

* Fixes: NB#221640 - <comment omitted>
* Fixes: NB#221699 - <comment omitted>
* Fixes: NB#207403 - <comment omitted>
* Fixes: NB#222038 - <comment omitted>

-- <Developer name> <e-mail> <date>
```

Figure 16: Changelog snippet indicating 4 implemented backlog items. The addition of such history improves traceability to into backlog when those items are integrated.

The example in Figure 16 shows changelog of account-plugin package version 0.7-24 implementing 4 backlog items. Comments have been omitted for convenience. When

the package is processed in Continuous Integration System, during promotion to Staging repository phase, an extra comment similar the one on Figure 17 is added into each backlog item to indicate tracing information of which package name, version, VCS URL tag id and CI request number originated such implementation.

```

--- Comment #36 from ci-promotion bot 2012-09-24 16:03:16 EEST ---
Accepted on <Release ID> 'Staging Repo' request (953586):
<URL to CI request>

Fixed in package list:
account-plugin 0.7-24 (VCS URL#tag-id)
revision: 8f1d1691cfdc624e90f9805438d6ea7elec9134c)

```

Figure 17: Automatic comment added into backlog item by linking the item itself, the continuous integration request and VCS tag id providing full traceability of integrated items.

The information displayed in Figure 17 is useful for tracing and history purposes when developers are troubleshooting or debugging issues in the product and to indicate the correct source code position a change has been introduced.

#### 5.3.4 Pre-Release Gate

The CI-promoter groups accepted packages on Staging repository and execute the automated process to release packages by promoting them to pre-release repository. Product build is created as part of release promotion process along with its submission to System Integration Automated Test phase covered in section 5.2.

Once packages are successfully accepted, Integrator executes scripts to move packages from Staging to Pre-Release repositories. These scripts are responsible to:

- Copy packages from Staging to pre-release repositories by re-indexing the pre-release database to understand newer package versions or newly added packages
- Include a new comment into backlog item (as depicted in Figure 18):

```
--- Comment #38 from CI-promotion bot 2012-09-24 21:00:09 EEST ---  
Released in pre-release repository version <daily release id>
```

Figure 18: Additional comment added into backlog item to track which daily release the item is part of.

- Move item state from RESOLVED to RELEASED.

Being in RELEASED state means, the package is part of daily release builds generated from pre-release repository. Further automated tests including acceptance and regression tests are applied on this build to verify the sanity of the build. Besides, smoke and exploratory tests are manually executed as part of advanced tests to make sure end-to-end functionalities work as expected.

In theory, the product owner or product manager is the responsible for submitting each of the backlog items and also accountable for verifying and accepting if the implementation is successful. In practice, it is a shared activity for whoever is interested in the implementation accounting for dependencies or any other blocking issues that by verifying the item would terminate such impediment.

During weekly release creation, backlog items are automatically moved from VERIFIED to CLOSED automatically during the process of release finalization. It is important for all teams to be proactive and VERIFY the backlog items assigned to their domain and components. If a task or bug is still under release state, it cannot be included in the release notes generation and is therefore skipped from the release.

It is important to note that several of the backlog items do not go through the automated process. Many of the tasks involve operational work not requiring software development, but may impede the progress of such and therefore, must be part of the dependency to have an epic or feature delivered. As an example, certification, hardware or even prototype activities are part of the same backlog, but not concern to software development and are equally relevant to successful product development. The slight difference is regarding the approval process. Once the item is set to RESOLVED, it's simply moved to VERIFIED by interested stakeholders and later CLOSED by weekly release finalization scripts.

### 5.3.5 Dependencies Graph

As mentioned in section 5.3.2, dependencies discovery and realization are the foundation of the successful scalability of this environment. By having dependencies visually followed by the organization improves awareness and simplifies the progress rate. This provides the idea of how complex it may be and supports further planning as well as organizing and estimating the work.

The tool used for the enterprise and product development backlog provides such an advent as shown in a generic example of Figure 19:

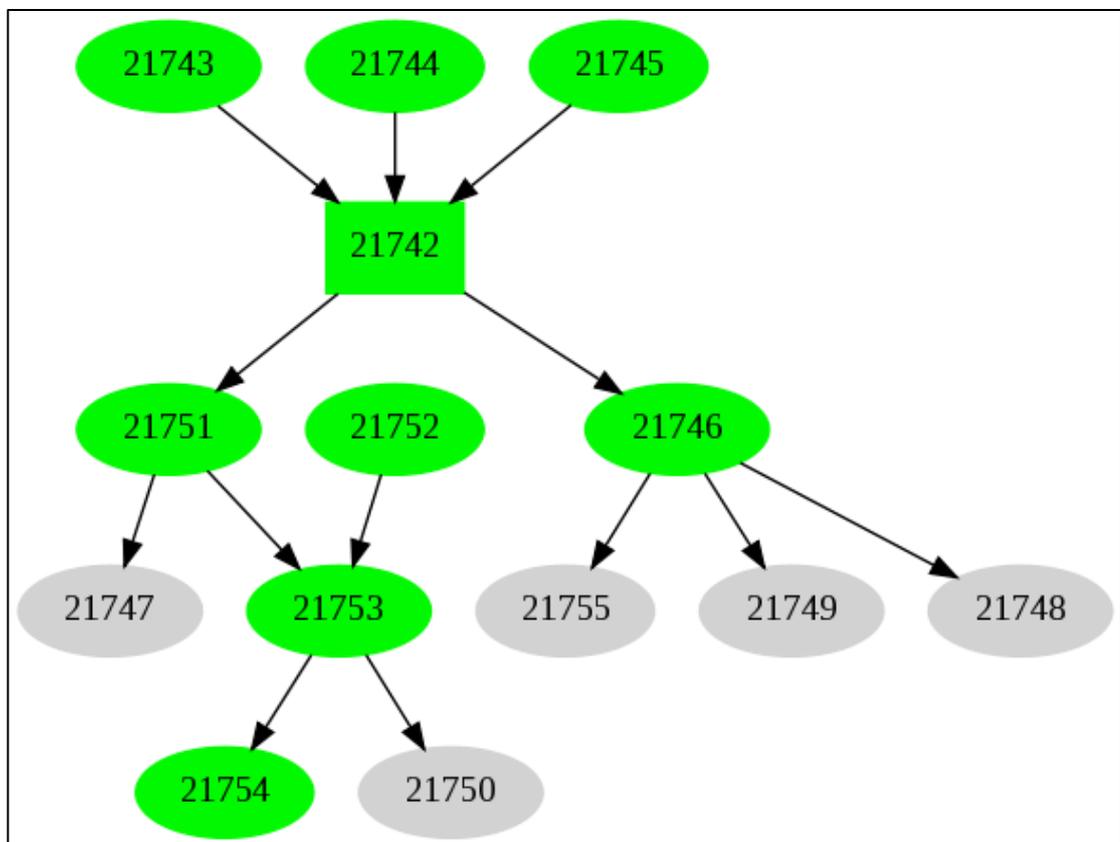


Figure 19: Backlog dependency tree. Green ellipses denote items new or under work. Grey ones designate finished elements.

Gray ellipses represent items that have been dealt either on any of the resolution states as RESOLVED, RELEASED, VERIFIED or CLOSED). Green ones are either NEW, ASSIGNED or NEED INFO.

In the contrived example, in Figure 19, main focused backlog item is 21742 (in square shape) that items 21751 and 21746 are directly blocked by it. On the other hand, item 21742 depends of the implementation of 3 items, 21743, 21744 and 21745.

Items as 21751 and 21746 might represent stories, which have underlying tasks linked to them. In this situation, such items are regarded as meta-informational and many times do not represent any actual implementation but only the grouping support. As a result, they are set to the corresponding state when the respective tasks have finished implementation.

In such representation might have items under several different components and each under their corresponding Scrum or Kanban boards. Product Managers and Architects are mainly responsible for setting up those dependencies relying on continuous collaboration on teams to keep them sane and correct throughout product development. It's essential to encourage self-organization, experiment with teams deciding among themselves — by interest, negotiation, or skill. This helps reducing decision-making effort by the Product Owner. Excelling in this practice, will boost the delivery of a successful product under large-scale, high complexity entity, reason for it to be regarded the most sensitive and critical effort.

Another representation for dependencies is similar to Figure 20. It first displays impeding items to selected task and next the ones block by it. Summary names are only illustrative. Strikethrough items represent resolved items.

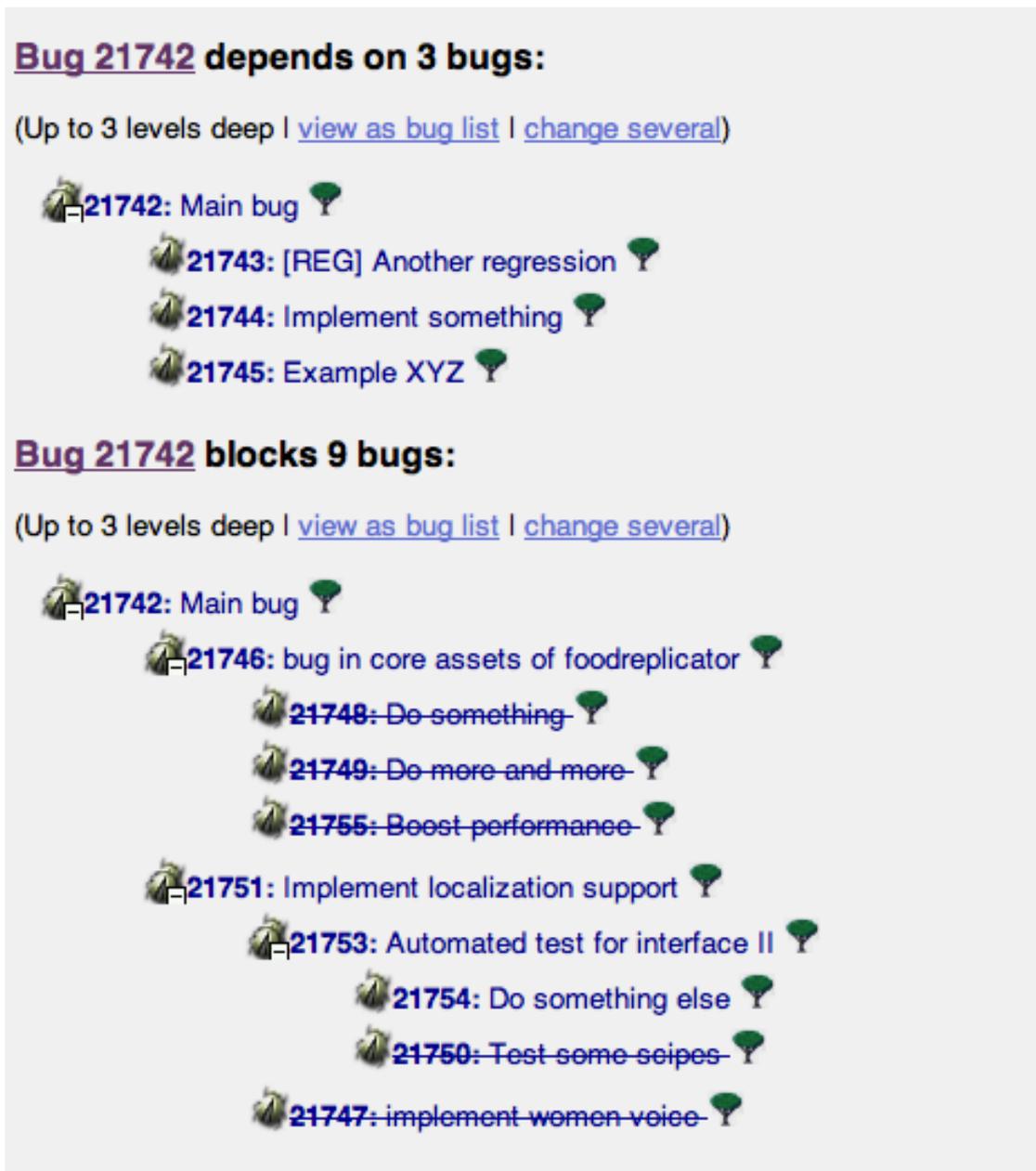


Figure 20: Alternative view of features or bugs dependencies. It provides more straightforward information about what is still missing implementation.

Both pictures are only displaying few items just for exemplification but in practice dependencies link can be scaled to up thousands of connections.

#### 5.4 Configurations

When binary software packages are accepted into pre-release or Staging repositories, it does not mean the Potentially Shippable Product Increment (see Release Planning chapter on section 5.6) is ready or there is a build ready for use. Actually, to enable

such possibility, there are several procedures to follow in order to have the product available on every Continuous Integration Cycle or even by demand.

The repository is a single location only where packages remain before being part of the product. On the other hand, the repository holds several thousands of packages among different versions of the same package, development and supporting tools, test packages, profiling, tracing, debugging and emulation tools. All of these packages are not part of the product, but needful instruments to support building the whole product.

Therefore, there is a need to have a kind of receipt to guide the build system of which packages to select when generating the product. This receipt is called configuration and more specifically product configuration, which, in turn, it's very similar to the definition of Potentially Shippable Product.

A product configuration is, consequently, no more than a set of binary packages put together into an executable or flashable image to mobile phones, conveying the commercial product.

In practice, a configuration is no more than a package that is likewise built into integration system, however, with different result. It is fundamentally a meta-package that by the end of the build process does not produce binary packages but, in turn, the releasable product. Hence, a package configuration differs from usual software packages by containing the following extra pieces of information:

- A list of core packages to install;
- Rules for building and creating the executable image;
- Mandatorily to build;
- Dependency to other configurations by constructing a hierarchy between configurations

Figure 21 shows the configuration package structure as a standard *Debian* Package. It is organized in a directory hierarchy including important files to support the image creation.

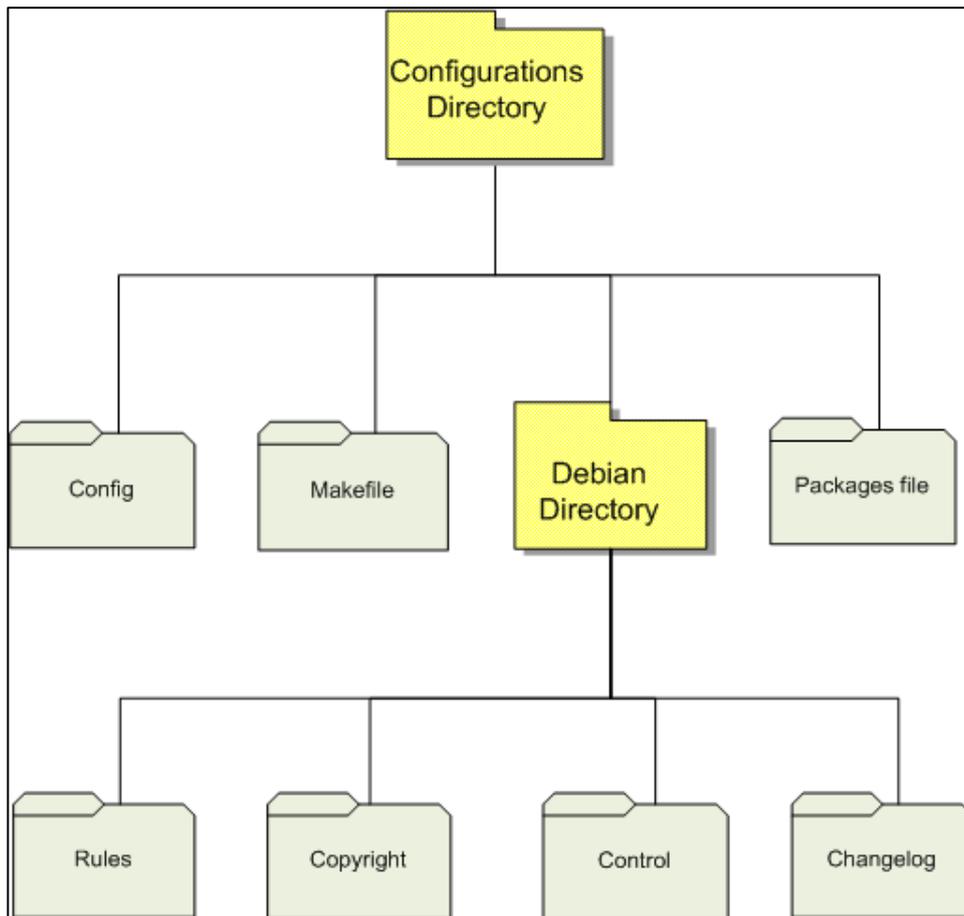


Figure 21: Configurations *Debian* package hierarchy.

**Packages File** is a simple text file listing all package names that comprise the core assets for building the whole product. Core asset packages do not rely on any other package to be installed representing the essential frameworks, Operational System and Middleware capabilities that every package in the product, either directly or indirectly, relies to be installed and configured.

**Makefile** is a standard Linux makefile and contains package build options. It provides the receipt to set up the environment for build and calls underlying scripts to execute all steps to generate the image by installing and configuring all packages under the configured environment.

**Config** contains configuration options and details about the configuration. It contains all the information of how to create the binary image by adding extra content and making it installable to the target devices. It provides instructions to the build system to enclosure the set of installed packages into a binary flashable instance.

The *Debian* directory, which always sits under the top level, contains the files required to build the package by producing the configurations. This include:

- **rules:** the installer script for the configuration. It calls the Makefile when executed by build system.
- **copyright:** no changes are required in this file concerning configuration packaging.
- **changelog:** holds all changes made to the package including details of any backlog item. As mentioned in the Error Management section, on chapter 5.3, backlog items must follow a format in order to be recognized by build system hooks to properly link backlog system to Continuous Integration.
- **control:** describes the package details such as in Figure 22:

```
Source: <configuration-name (source package)>
Section: misc
Build-Depends: <dependencies to create configuration build
environment>
Priority: optional
Maintainer: Configuration Owner
Standards-Version: 3.7.2

Package: <binary configuration package name>
Architecture: all
XB-Buildable: yes
XB-Mandatory: yes
Description: Configuration package for the product
  This package defines package list of configuration.
```

Figure 22: Configuration control file details. Highlighted fields show the key differences between a standard control files.

The important fields in this file are:

- **Build-Depends** lists what packages the configuration package depends on to be built. Other configuration packages can also be added to it by creating a hierarchy structure between configurations improving reusability among them.

The configuration inheritance will be covered later in this chapter on section 5.4.2.

- **XB-Buildable** denotes that the configuration has to be built on every nightly build pulling new or upgraded packages from pre-release repository. This is a strategy to assure all configurations are sanely built on daily basis after daily development cycle. Few supporting configurations are not required to build on daily basis, either because they represent unused configurations or not part of product creation life cycle. In order to save system resources, the buildable capability is “turned off” for those.
- **XB-Mandatory** is used to define a list of configurations mandatory to be built on every stage in Continuous Integration System by making sure, the product is leveraged along with all CI gates of development cycle. The same configuration is sent to test automation system when reaching every gate step in Continuous Integration. Mandatory configurations are created several times on each and every integration request submitted by developers throughout CI cycle. Although it might bring slowness to integration flow forcing developers to wait for product build generation in all gates, it guaranties product sanity while product complexity increases. It fosters product visibility all over the integration flow by creating evidence of the evolving product as well as to packages maturity throughout CI gates.

#### 5.4.1 Types of configuration packages

Besides product configuration, developers require several other configurations to assist their work of testing, profiling, measuring, debugging and emulating when implementing their components. For this purpose, not only the product configuration is maintained across product creation life cycle. A few others are created and carried out for this purpose. They include:

- **Acceptance and Regression Testing (ART):** As the name implies, CI uses the packages for the automatic component of acceptance testing. It's mainly created during the packages acceptance procedure from Staging phase to Pre-release repository and it is also used as part of the validation process after packages acceptance.

- **Basic Regression Testing (BRT):** Used during Staging phase for automated regression testing
- **Research and Development (RD):** Used during the research and development processes, which includes specific package to support developers with their development tasks. Developers mainly install RD configurations on target devices or emulators for their daily activities.
- **Tracing (TR):** Used to produce images with enhanced reporting and logging data output as part of the development activities.
- **Field Testing (FT):** Similar to TR but also includes cellular tracing and debugging modules turned on along with few extra applications to support field testing and respective certifications.

#### 5.4.2 Configurations Inheritance

There are few other configurations used as supporting structure for the main configurations. They are called meta-configurations and are not part of integration process, hence not buildable configurations. In order to promote reusability among configurations, meta-configurations are used as top-level containers holding common packages references to all other inherited configurations. By changing the packages under packages file in meta-configurations affects all other inherited configurations.

This hierarchy is shown in Figure 23:

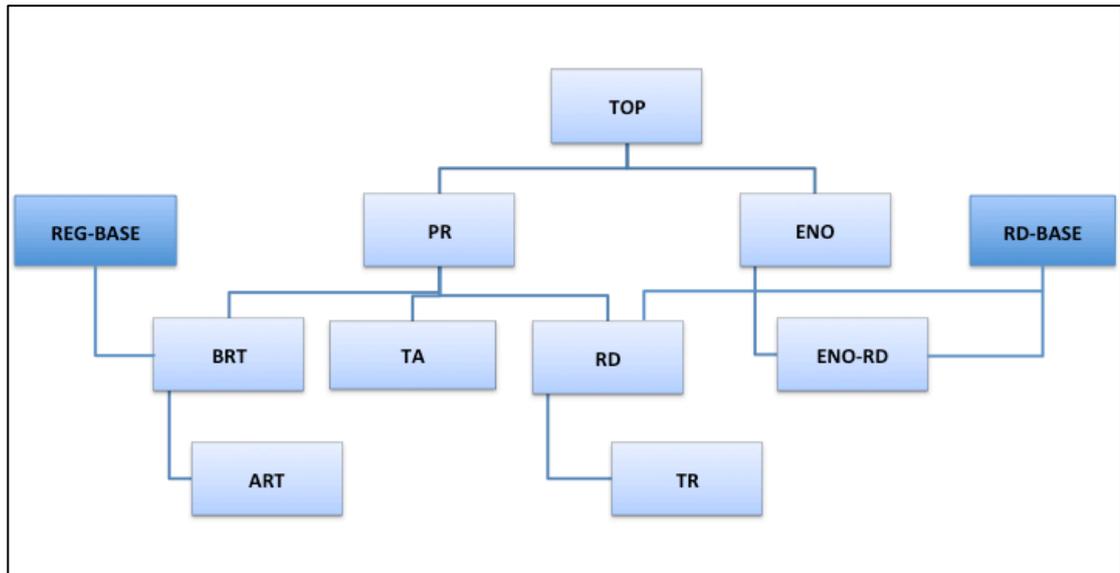


Figure 23: Configurations inheritance tree stressing dependencies between each other and reusability advantages.

TOP and BASE configurations are pure meta-configurations assisting reuse of packages in the configurations hierarchy. REG-BASE meta-configuration for example, includes all specific test packages for regression and acceptance testing scope commonly used by BRT and ART configurations. In the same way, TR configuration inherits all specific packages from RD configuration, which by instance receives specific RD level packages from RD-BASE and also from product packages.

Such a hierarchy is flexible to increase by the team's demand in case it is needed to maintain a specific set of packages, either not part of the product or under the maturity stages of implementation. Later on, the team can decide to include the package only when they are sure it will not negatively affect the rest of other packages. This approach is usually observed on low-level packages, which have a tangled web of dependencies such as application framework or product kernel.

When the build system starts processing one of the inherited configurations, it looks into build-depends field in control file in order to calculate dependency tree and create the complete packages list from all packages files belonging to hierarchy structure as the baseline for start installing the packages into the target environment. As a result, the maintenance of such configurations is scalable and simple allowing the product to include quickly new changes or additions promoting agility in its visibility and progress follow up.

### 5.4.3 Maintenance in the Packages Configurations

Packages end up included into product configuration either by dependency or explicitly adding package name to a package file of the corresponding target configuration. This package is flagged as mandatory by the team and part of the automated test while under Continuous Integration process. In this sense, it is artificially added on top of the product and regression configurations to make sure it will not break the product functionalities.

In the case of direct dependency, once the package reaches the pre-release repository, it automatically becomes part of the product. Even on version upgrades or newly included packages, the newer version or package ends up included in the product image after dependency calculation when the product is being built. However, if there is no dependency to pull in the package from the repository during the product build, it is required to have it manually added into packages file of underlying configuration. This task is performed by demand by teams' Product Manager or architect request when package maturity reaches a maturity stage it can be added into final product. Moreover, the same approach happens to have packages added into other supporting configurations. The Configuration Owner is the one responsible for managing such a scenario. This assignment also makes sure the configurations are always buildable and functional.

Basically, the Configuration Owner will handle the operational duties to include, remove, rename packages into package files as well as properly integrate configuration packages executing the changes required. Likewise any other requirement or bug, the packages handling tasks towards configurations are also submitted into backlog under corresponding, Configurations, component. Thus, it makes it easier to project managers to plan the whole development effort by also including the backlog dependency to package handling against configurations in order to have the team's work fully covered towards the final product.

In order to assist Configuration Owners to properly perform their tasks, packages are registered into a tool that stores all packages metadata. Among other purposes, the package database tool mainly supports Configuration Owners by keeping the packages information correctly maintained during the product development.

Only registered packages can be added to configurations, which are tracked by the package database tool. So, whenever a new package is introduced into the Continuous Integration System or has its name changed, it must to be first recorded into the package database in order for teams and mainly architects to keep track of these modifications under each domain and component of the product. By adding a package to such database, in practice, means choosing the corresponding domain and component which the package belongs and create first source package information and next the respective binary names. Another important information to include is the copyright holder for each package. It actually dictates where the package is going to be stored in the repository. If there are copyright restrictions for some of the packages, they must be indicated as part of the registration into package database. This information limits the access to selected packages in the repository. As a result, only the build system is able to reach these packages, though.

## 5.5 Mass Customization

Software customizations are ways in which customers modify the product user interface, connectivity settings and user experience to better reflect brand and to fit the network and market in which the product will ship. This can include applications, modifying icons and layouts, change defaults and add brand-specific art such as sounds and pictures.

Customization, therefore, is the key enabler towards commercially viable products and locally relevant solutions. It strengthens product diffusion in a wide diversity of markets covering different and cultural consumer needs as well as technical specificities discriminative of different regions the product is sold. It also provides ways to magnify customer brand enabling differentiation by adding value to customer's brand proposition.

As important as the commercial features, customization also has to provide scalability, flexibility and dynamic when building the product. The strategy in this case, is to mass customize in order to create customization-enabled products along with its development cycle. For such, it requires high-level of automation as well as gathering customization requirements along with partners for quick responsiveness.

Mass customization goes around the concept “build to order”. “Build to order” is certainly the solution for mass customization when the company does not know demands until having orders in hand. Hence, production just happens after that. In the past, with the concept “build to forecast” or “build to stock”, production was based on demand forecast. The drawback of demand forecast is the forecasted information is not exact since it relies on past numbers and many assumptions, thus the outputs of production cannot match the highly different demands of mass customization. (Gilmore & Pine, February 2000, 77-79.)

As a result, software mass customization focuses on the means of efficiently producing and maintaining multiple similar software products, exploiting what they have in common and managing what varies among them, known as Variants. This is analogous to what is practiced in the automotive industry, where the focus is on creating a single production line, out of which many customized but similar variations of a car model are produced.

All in all, some key benefits are identified when adopting mass customization against standardization of products. They are:

- Higher profits for providing tailored needs supplying marketing differentiation;
- Lower costs bringing exceptional value for money in a competitive price;
- Reduced lead time in comparison to build different standard product to fulfill consumer needs;
- Scalable and flexible strategy enabling the just-in-time lean thinking and “build to order” strategy.

The larger the customer base of a company is, the more it has to offer in terms of customization asset. The collaborative customization approach (Gilmore & Pine, February 2000, 78), where companies work in partnership with individual customers to develop precise product offerings to best suit each customer’s needs, precisely fits into the mobile devices market due to an ample range of carriers spread around the globe as much as retail channels with a strong presence in some local markets.

In software, the collaborative customization takes place by identifying the sorts of customization requirements that product needs to be commercialized under each local market. It goes from legal standard requirements enforcing the basic obligations to have the features implemented in a way to obey the govern laws as well as regulatory agencies responsible for product certifications and authorizations. These requirements differ from region to region and country to country demanding the software assets to comply with all these requirements in order to enable its availability in such markets.

As example, the Chinese government prohibits any of the most worldwide known social and media services to be shipped along with any product. Many products include in their current standard portfolio, Facebook® and Twitter® currently forbidden in China. For these products to be sold in China, one of the strongest markets with a high demand for consumer electronics, these services must be taken out from them. Another case is the lack of regulation for NFC use in Argentina. Products or services that make use of this technology must have this functionality disabled; otherwise the Argentinian regulatory agencies will not certify the products under these circumstances.

A second group of requirements lay down over intrinsic clients' capabilities to operate the product under their respective environment. For mobile phones, it means all the carriers specialized and fine tuned configurations of the installed cellular network base. These requirements must implement flexibility to mobile products to suit to these specifications. For instance, the signal strength bars represent the strength of the cellular connection and are determined by a mapping table defined for the each network the device is operating. Each network, such as GSM, UMTS, CDMA and LTE has different tuning for the thresholds for measuring the signal strength received from the modem. So, the correct implementation of these mapping tables for each network is important to assert the accuracy of signal bars in device's user interface.

The third group of requirements concerns differentiation in user experience by bringing value to brand equity. They also split in two types: user interface requirements and pre-loaded content requirements. An user Interface refer to changes like wallpaper definition, menu layouts to increase visibility of customer specific content, additional e-mail accounts, themes, languages, additional default keyboards installed and all kinds of pre-set configurations that provide appealing out-of-the-box experience. The second type concerns to pre-loaded content such as customer specific applications including also locally relevant software as well as digital content such as pictures, ringtones and music. Along with software customization, there are a few other assets such as hard-

ware and cover customization, sales packages, user guidance and accessories (headphones, memory cards and handsfree kits) that comprise a ample scope of mass customization in the enterprise which software customization is just another component, the more complex and the one that brings highest value, though.

The scope of this thesis focuses on how to create the software customization enablers in an agile and lean environment leveraging different variations along with product development. Improve product customizations throughout key customer validation and local sales offices. Additionally reduce lead-time that customization phase would possibly introduce during sales release candidate creation.

### 5.5.1 Creating Customization Enablers

Features and components are implemented according to requirements available in backlog. Part of these requirements refers to user stories that request the feature to have its functionality changed to support different values defined at build time. In fact, features that require customization changes are implemented in a way that feature configuration values are disassembled from the main component and defined as part of another component called customization enabler. Both rely on each other, not directly dependent, once enablers are regarded as core asset packages always included in software configurations or via virtual packaging dependency.

One common example to illustrate this technique is to consider the feeds feature. The feeds display frequently updated content selected by user or pre-configured by operator before sold to consumers for their news or merchandizing. The middleware component of the feeds feature that interprets the values each feed expects to have a list of feed title, feed icon name as well as the icon path location to file system and feed URL. Besides, feeds can be also configured on how they are displayed and organized to users such as showing all items or only unread ones, update feeds entries manually, through WI-FI only or always update regardless the medium and how many items to display in the list visible to user. In this sense, the feature implementation then exposes these settings, where the values are set, into another component, the customization enabler. It can be simply a text or xml file containing the values that will initially configure feeds settings in the device. The component expects this file is located somewhere in the file system and when reading the values they are used for feeds functionality to be initialized in runtime.

Another alternative to expose customizable values is to maintain a database storing all the values that may change during customer configuration so that, if features require large amount settings to be defined, it is easier to keep them under a database structure. This Database functions in the same idea as the Windows registry stores the application or features settings to be used and shared by the Operating System. In the same way as in file-based approach the settings are exposed in the main component while values are defined in another, at the customization enabler. However, instead of creating individual files, the enabler contains scripts to store values in a common database also part of the Operating System.

On Linux Systems, the most common database to manage settings and preferences is *GConf*. Essentially, *GConf* provides a preferences database, which is like a simple file system. The file system contains keys organized into a hierarchy. Each key is either a directory containing more keys, or has a value. For example, the key `/apps/feeds/general/show_all` contains an integer value indicating whether to display all feeds items or only unread ones. It also provides standard APIs for accessing and modifying configuration information. In this work, *GConf* is the preferred way to store settings of a software or application, mainly because of the following advantages over custom configuration files:

- It has built-in support for default values (using schemas);
- “Backup & Restore” and “Restore Factory Settings” are automatically handled by the system. If a consumer wishes to revert the settings to factory defaults, meaning to the values set when the device was turned on for the first time, *GConf* is capable of reverting to defaults stored in its database;
- Merging and overwriting data is significantly easier than with custom configuration files (through schemas).

## 5.5.2 Package Based Customization

The most important point to note is not how customizable values are stored and retrieved under OS, but the process of how customization enabler gets instantiated as-

suming either default or customer defined values. The package-based customization method entirely fulfills this constraint and is based on the principle of splitting the application or software and its settings into separate packages. A package that provides default values for settings and other customization settings is called a variant package. Packages that are not variant packages are called functional packages. Comparatively to section 5.5.1, functional package is the feature and variant package the enabler.

A variant package can cover all of the settings of a functional package, or only a subset of it. It can also cover the settings of more than one functional package. The functional packages define how their variant packages should be structured. Covering one or more functional packages with a single variant package is preferred over having multiple variant packages for a single functional package.

The definition of the variant package structure is provided in a formal way by defining template packages for the variant packages that are automatically instantiated by the variant creation tools under Continuous Integration System. The relationships of these concepts are shown in Figure 24:

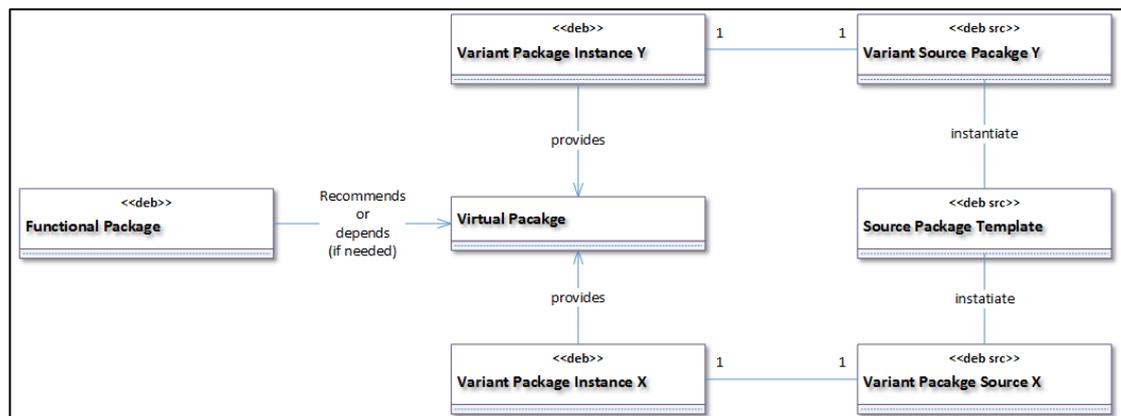


Figure 24: Functional, Variant and Template packages relationship

Source Template packages have a standard name and format that supports such instantiation process. These packages are in the strict format <feature name>-template.deb (feeds-template.deb in the example above) and package structure is defined as a standard Debian Package.

Figure 25 depicts the source package template as a special kind of source package that can be transformed into different, variant specific source packages. Source pack-

age templates must be valid and buildable source packages; it should be possible to upload to a repository and build them. However, the binary packages built from a source package template are not required to provide any useful functionality. The useless binary packages are needed because Continuous Integration tools expect *Debian* source packages to build at least one binary package. The template binary packages should never be part of any configuration.

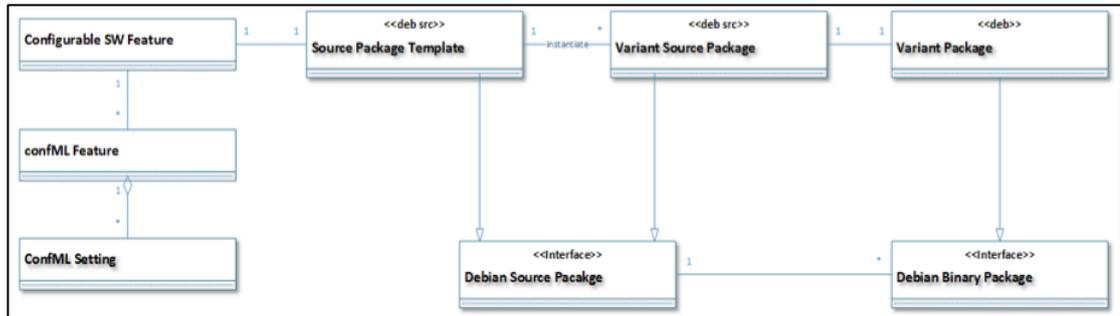


Figure 25: Packaging concept and relationship between confML, source package templates and variant packages.

A source package template is, therefore, organized in a *Debian* directory hierarchy including important files to support package instantiation as depicted on Figure 26.

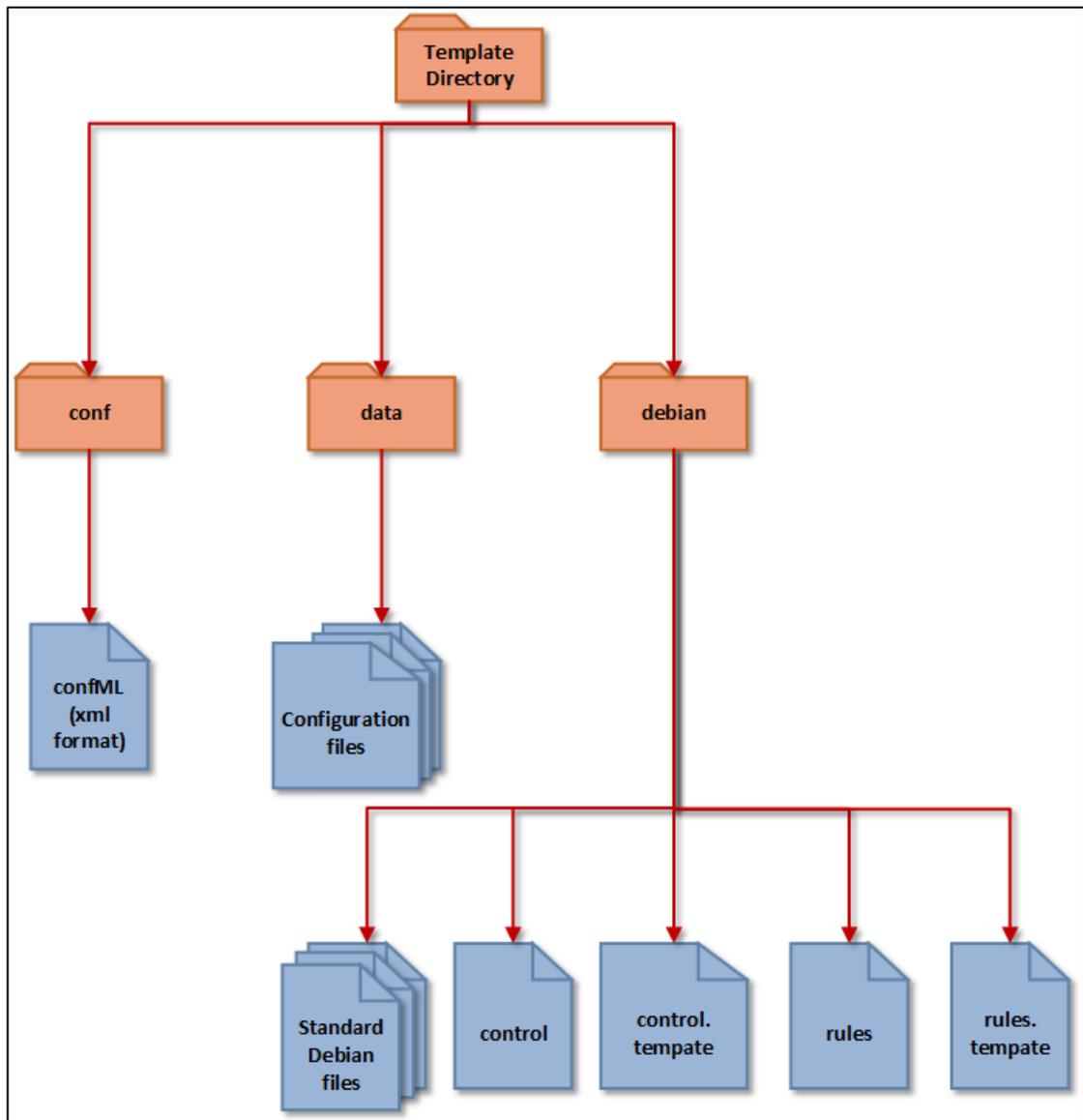


Figure 26: Template package *Debian* package structure

As Figure 26 shows, the detail of a template package is followed by:

**confML file** (Configuration ML) is a file in XML-based markup language (that is, an application of XML) for defining software Configuration Elements in order to express configuration data of a software product. A ConfML file can contain:

- A declaration of a set of Features, providing an interface definition;
- Data values for those Features

A collection of ConfML files can make up an interface definition whose purpose is to provide operators with the ability to customize certain Features. ConfML descriptions allow configurator tool (see section 5.5.5) to visualize, validate and restrict data when added by customization representatives while customizing the product.

There are three main concepts in the language:

1. **Feature**, whose role is to group related settings.
2. **Setting**, whose role is to describe a capability implemented in software.
3. **Data**, whose role is to define the values for *Settings*. Generally, the values in *<data>* section are the default values for the feature itself.

Figure 27 provides a code snippet of a confML file from the feeds example previously mentioned:

```

<feature name="Feeds Settings" ref="feeds-settings">
  <desc>Default subscriptions for feeds application.</desc>

  <setting name="Feeds" type="sequence" ref="DefaultFeeds" required="false">
    <desc>Default subscriptions for feeds application</desc>
    <setting name="Feed Title" type="string" ref="FeedTitle" required="true">
      <xs:maxLength value="50"/>
      <desc>Title of the feed.</desc>
    </setting>
    <setting name="Feed Url" type="string" ref="FeedUrl" required="true">
      <desc>URL pointing to the feed. This needs to be a web URL where the feed is published.</desc>
    </setting>
    <setting name="Favicon" type="file" ref="FeedFavicon" required="false">
      <property name="type" value="image/jpeg image/png"/>
      <localPath/>
      <targetPath readOnly="true">BUILD:///data/favicons/</targetPath>
      <desc>Favicons for the feed. (upload only jpeg/png images)</desc>
    </setting>
  </setting>
</feature>

<data>
  <feeds-settings>
    <DefaultFeeds>
      <FeedTitle>Linux</FeedTitle>
      <FeedUrl>http://linux.org/rss.xml</FeedUrl>
      <FeedFavicon>
        <localPath>penguin_picture.jpg</localPath>
      </FeedFavicon>
    </DefaultFeeds>
  </feeds-settings>
</data>

```

Figure 27: ConfML/XML structured in *<features>*, *<settings>* and *<data>* elements.

**Data directory** is the location where configuration files, *GConf* schemas and other content such as pictures and sounds is located. The values are replaced into configuration files during the instantiation process covered on section 5.5.3. In the feed example, the `penguin_picture.jpg` is physically located at `data/favicons` and the

feeds.opml.template file under data directory and has the xml content as displayed in Figure 28:

```
<opml version="1.0">
  <head>
    <title>Defaults imports for feeds</title>
  </head>
  <body>
    <outline title="@FeedTitle@" feedUrl="@FeedUrl@"
      favicon="/etc/xdg/feedreader/favicons/@FeedFavicon@/" />
  </body>
</opml>
```

Figure 28: Configuration file template snippet including the placeholders (“@Something@”) for instantiation.

The items between “@@” are replaced by the values under `<data>` during the instantiation as well as the leading “.template” of the file name.

**Debian Directory** contains the files required to build the package by producing the so-called variant package. The control and rules file are used to build the enabler package, or the template package. The `rules.template` and `control.template` are the respective Debian files created for building the variant package after the template package instantiation.

The `control.template` file is a common Debian control file with few peculiarities as seen in Figure 29:

```
Source: feeds-settings-@VARIANT@
Section: misc
Priority: optional
Maintainer: maintainer@maintainer.org
Build-Depends: debhelper (>= 5)
Standards-Version: 3.8.0

Package: feeds-settings-@VARIANT@
Architecture: all
Provides: feeds-settings
Description: Customization package for feeds application
```

Figure 29: Template control file, including the environment variable @VARIANT@, to be replaced by VARIANT ID during instantiation.

The `@VARIANT@` placeholder is the variant configuration name which replaces the package name after the instantiation from a template package to variant package. The `@VARIANT@` placeholder is part of an environment variable that designates the variant identification for the instantiating variant. Therefore, `@VARIANT@` is replaced by the variant name after instantiation.

The Provides field, as previously seen on the *Debian* packaging virtual package notes, basically provides the package name for dependency purposes from component (functional) package to the actual variant package throughout the virtual package name. The same happens to the `rules.template` file, which after instantiation it is only `rules` file and all placeholders under it have been also replaced during instantiation. For details about instantiation, see section 5.5.3 in this chapter.

### 5.5.3 Instantiation Process

The process of modifying an enabler known in practice as a template package into a variant package is called instantiation, depicted in Figure 30. The package templates are instantiated with a script that basically performing the following:

- Process all files ending with `.template` in the source package template (also under subdirectories). For each file named like `FILE.template`, write a new file `FILE`. Each occurrence of `@SomeSetting@` (case sensitive) in `FILE.template` is replaced with the value of `FEATURE/SomeSetting` in `confML`, `<data>` section.
- For the `@VARIANT@` tag, replace it with the `VARIANT_ID` which is the identifier of the variant. The same happens with package name from `<component name>-template.deb` to `<component name>-VARIANT_ID.deb`. Variants definition will be covered in details in the next topic, 5.5.4.

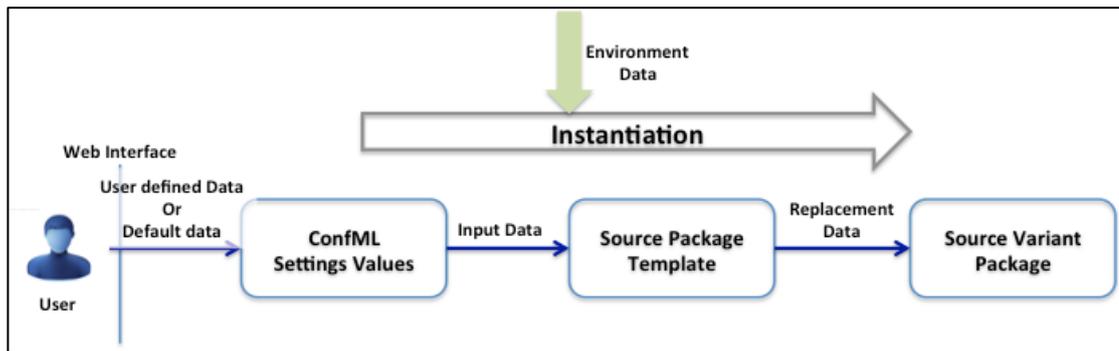


Figure 30: Instantiation process

Figure 30 defines the data flow diagram when an instantiation occurs over a source template package. User (operator representative, for instance) defines the data through a web interface of what must be the configuration preferences for a certain feature. This data is automatically placed under `<data>` tag into confML, which during instantiation, along with environment data such as VARIANT ID, is used to replace data on the template package by creating a new package, the variant package.

#### 5.5.4 End-to-End Customization Data Flow

In order to perceive how the customized data is saved into final product variants, it is important to understand the key role Continuous Integration has as the backend enabler of the customization end-to-end process.

Initially, Variants are defined as different, possibly customized versions of the same product. When considering mass-customization production, there is no "unvaried" version of a product but instead; all shipped versions of a given product are called variants. The difference between variants may not necessarily be in the SW, but it could also be in hardware, mechanics and mass-memory content.

Vanilla Variants or Regional Variants are variants that are under product program responsibility and are not customized for any specific customer but usually intended as a baseline variant for a geographical region. The settings and content in Vanilla Variants can differ because of legislation, marketing campaigns or cultural differences as mentioned early in the chapter 5.5.

Actually, in practice, aside regional variants, a Vanilla Variant can be also targeted to a key customer that requires a many special customizations and specific, customer ori-

ented, applications. Hence, along with product development cycle, the customer can track and accept special customizations while product evolves as well.

Customer Variants are the variants created and maintained under customers' responsibility along with product account representatives. The settings under customer variants are defined according to customer brand and are implemented on top of Vanilla Variant settings defined as an inheritance hierarchy between vanilla and customer variants. Usually customers are allowed to change all settings that are offered by customization policy, while some of them are part of regulatory or low level customizations that only apply to product program to define and tune them during incremental delivery of Vanilla Variants. Hence, the utmost software product, which concerns customers, is the Vanilla Variants, from where it will build the customizations for later, after entire acceptance cycle, to be shipped along with the product, specific for that customer. This method enforces Vanilla Variants to be the main artifact from frequent releases product program conduces on a weekly basis.

In essence, a variant is just another software configuration. The difference regards to the different settings each configuration holds. For example, the Vanilla Variant defined for North America region has English US defined as the startup language, it contains specific applications in more evidence in this region (along with other global applications), the map content related to countries under this geographic region and specific government alerts enabled. Likewise to North American vanilla, the Chinese variant implements all restrictions imposed by govern as well as other basic China, specific configurations as Chinese input methods setup by default as the main keyboard for typing messages.

In order to leverage such flexibility, a process is required to be followed in the following order:

1. Teams identify, along with product managers, the customizable features.
2. Teams implement customization features by splitting the feature development in functional package and template package.
3. During implementation, teams define default values for each setting under confML file.

4. The team delivers both a functional and template package through CI. During CI execution, few steps are executed towards accurate integration of both packages:
  - a. First, CI normally builds both, functional and template package. On template package, it executes few extra steps to make sure confML is free of errors and the package follows all integration policies concerning template packages
  - b. After the creation of both binary packages, a variant build process is triggered through following steps:
    - 4.b.1. It first instantiates the template package replacing the customized data from confML by generating the actual variant source package.
    - 4.b.2. CI System, then triggers the actual build of variant source package by generating the respective binary package from it
    - 4.b.3. Lastly, CI invokes a variant build by adding functional binary and variant binary packages into a temporary configuration. The resulting product build contains the customized data into it.
  - c. The image build is then sent to automated test system where corresponding customization tests that have been also either delivered along with functional and templates or previously through another integration process. It is important to keep in mind test packages are not mandatory packages and are accepted by CI gates automatically.
  - d. Finally, both packages are accepted under CI Staging Gate, ready to pre-release promotion. Template packages as well as customization test packages are not mandatory, therefore automatically promoted by ci-promoter due to the fact a test variant build, during Staging Gate promotion, proved its respective variant package is qualified to be part of pre-release repository.
5. Once in pre-release, the variants maintained by the product program automatically include instantiated variant packages into it. The intent is to build a new repository which sole purpose is to store variant packages instantiated for each of the Vanilla Variants. Therefore, when the Vanillas are updated to build up new customizations, each of the accepted template packages is rebuilt and stored into customization re-

pository. This procedure happens automatically several times a day whenever a new template reaches pre-release.

In order to illustrate the steps from 1 to 4 in the process, Figure 31 depicts the integration flow of packages up to reach pre-release repository. Step 5 is detailed in Figure 32.

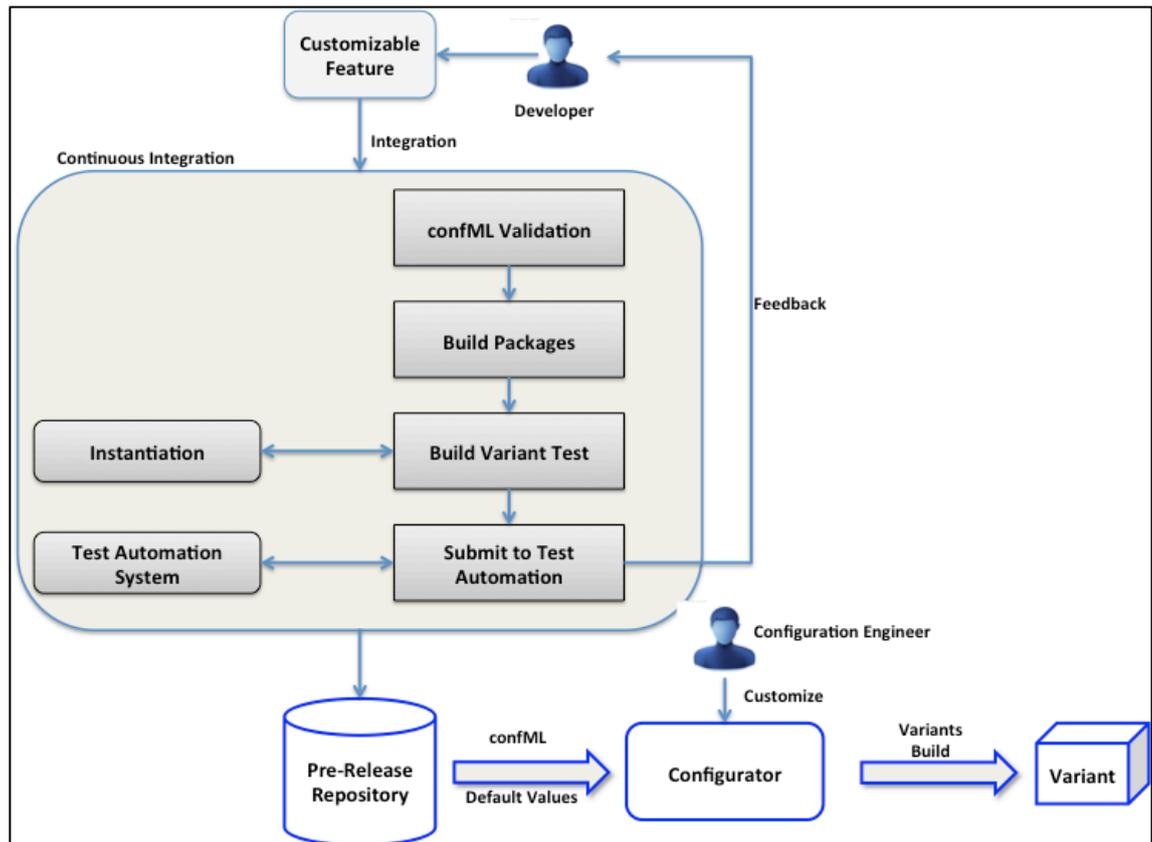


Figure 31: Customization Integration process

Up to this point it was covered how the default values, by the development teams and defined by product managers are instantiated into variant packages under Continuous Integration. However if no changes are performed, all variants end up with the same data when regional variants are created. Actually, this is the initial arrangement just after template packages get into the pre-release repository. All variants have the same default value for the corresponding integrated template. Thus, product managers have to define and request the equivalent setting values pertinent for each of the variants otherwise, the default will take place. Figure 32 along with section 5.5.5 explains how a Variant is changed from its default and rebuilt throughout the Continuous Integration

System. Using Figure 32 as a background, imagine the Configuration Engineer has worked on North America (NA) Variant under configurator.

All builds triggered from Configurator for NA Variant, will:

1. Export NA customized data from Configurator to Continuous Integration to have the build prepared.
2. All template packages are retrieved from repository and instantiated with data from the configurator. In this case, VARIANT ID is NA and therefore, all variant binary packages after instantiation and respective build will have `<feature>-NA.deb`.
3. These packages are stored into Variant Repository and along with the pre-release baseline repository, where functional packages reside, is used to build the NA Variant.

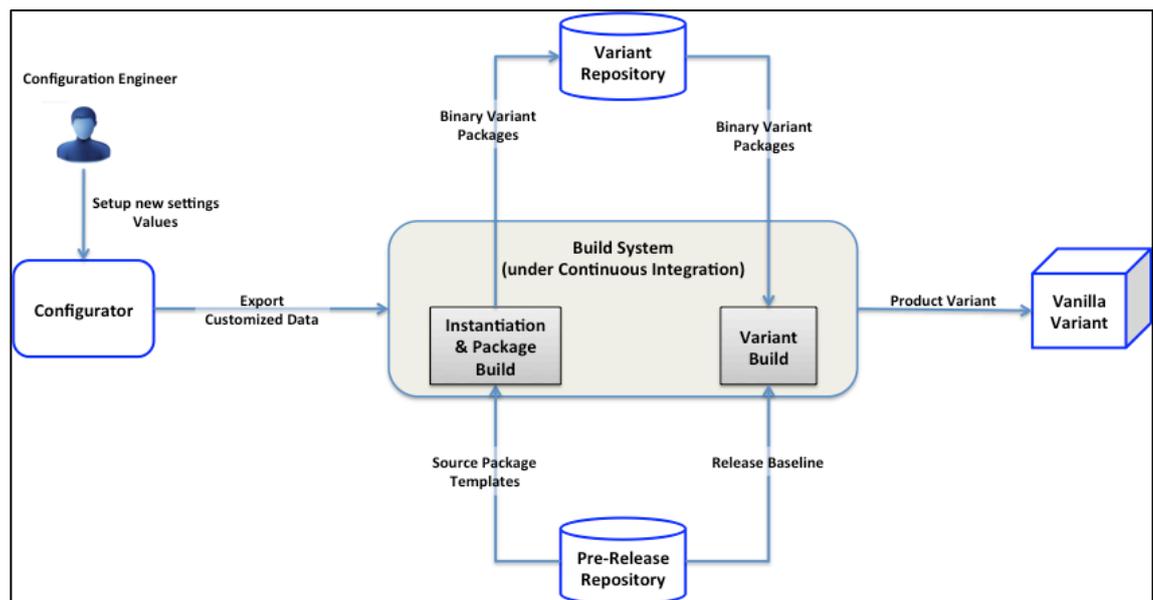


Figure 32: Variant Build in detail. It covers the process from Variant changes in Configurator up to Variant Build throughout Continuous Integration.

### 5.5.5 Web Configurator

As previously mentioned in section 5.5.4, for the activity to have default values changed from the original, default implemented values to realistic customized values, product managers submit an operational task into backlog. The purpose is to configu-

ration engineers to align the changes into respective variants. A web configurator tool assists these engineers replacing defaults with similar, regionalized, data. This web configurator has several attributes.

- The configurator engine retrieves and renders all confML files from template packages available in pre-release repo. The rendering process transforms xml information present in confML file into web forms to support users to visually interpret settings definition in a web interface. Whenever a new template package or an upgraded version of it is accepted into pre-release, the rendering process is triggered updating the configurator web user interface. As a simple example, Figure 33 displays the Feed customization form present in the feed description confML after proper renderization.

The image shows a web interface for customizing feeds. On the left, there is a sidebar with three menu items: 'Feeds', 'Bluetooth', and 'Wallpaper'. The 'Feeds' item is currently selected. The main content area is titled 'Customizations for Feeds'. It contains three input fields: 'Feed Title', 'Feed URL', and 'Favicon'. The 'Favicon' field has a placeholder image of a square with an 'X' inside. A 'Save' button is located at the bottom right of the main content area.

Figure 33: Web Interface for feeds customization by rendering the confML <setting> elements and producing a web form alike depicted in the picture.

- The Web Configurator assists configuration engineers to change settings according to requests submitted by product managers in order to conform the regional variants to required customizations. Thus, the web interface is used to override default settings implemented during feature development. Once a default value is set on configurator web interface, it will no longer assume defaults, even if new confML versions are imported containing different default values for manually changed settings in configurator.
- The Configurator enables the creation and maintenance of variant configurations organized in a hierarchy structure as seen in Figure 34. This structure is

flexible to assume any type of configuration and request. For instance, if a country has some specific settings but majority can be inherited from its regional variant, the configurator holds the dependency between parent (regional) and child country Vanilla Variants. All settings changed in the parent are consequently automatically inherited to its children. This approach speeds up the vanillas setup process by reusing several common settings. It also helps reporting towards variants values change progress once it also visually provides ways for product managers to compare one or more variants by easily providing a picture of which setting is still required to be part of each compared variant.

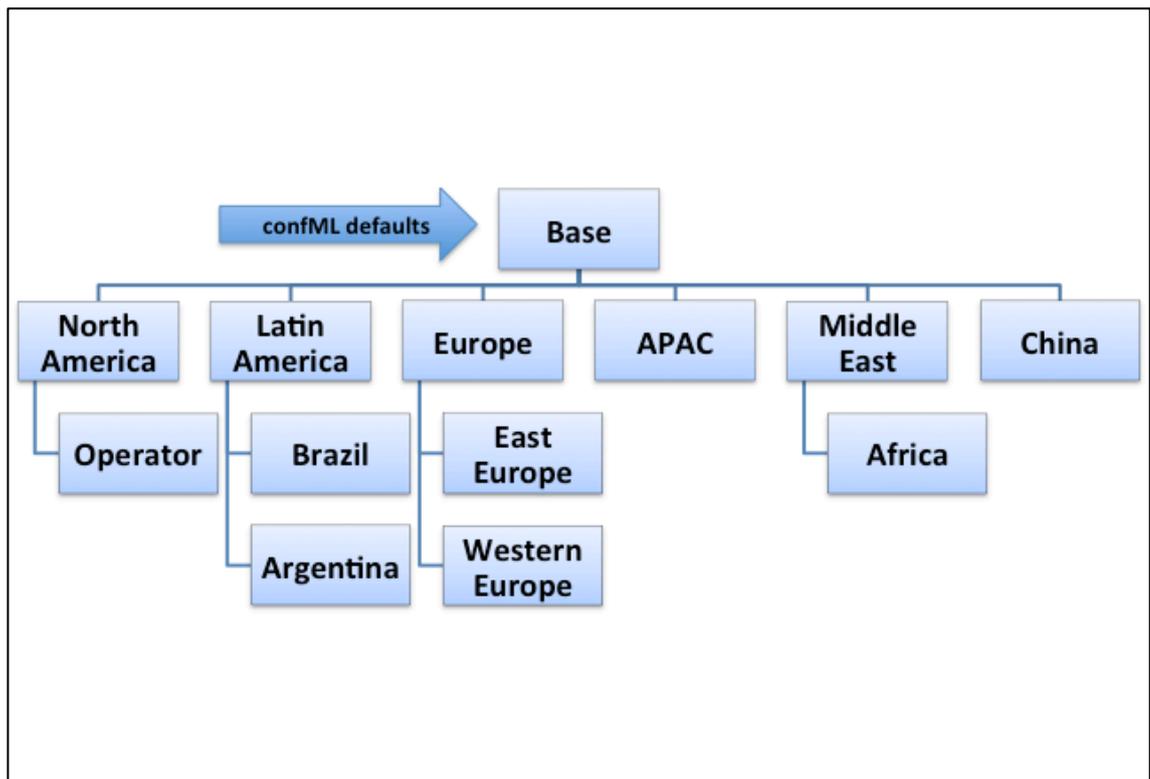


Figure 34: Variants hierarchy structure. ConfML defaults are imported into base Variant which values are automatically inherited by descendant variants in the tree.

- The Configurator also is the tool, which customer account representatives create, maintain and build the Customer Variants. Thereby, these variants are created using as baseline one of the Vanilla Variants where all basic and mandatory settings have already been applied and tested by the product program. Whenever a new change, whether functional or customization related, they are automatically replicated to Customer Variants. The customization changes (new or changed settings) appear straight on Configurator, while functional changes are available in the pre-release repository from where the customer variant will

build against when a variant build is requested from the Configurator (See Figure 32). While customization changes are updated on Vanilla Variants very often during development phase, always when a new template package is integrated, these changes are more controlled when they are published to customers. Only on weekly releases, or even only on Sales Candidate Releases (it will depend about the product development phase), the changes are replicated to users and therefore, they can generate their build against stable release baselines.

## 5.6 Release Planning and Validation

The product releases in the case organization are based on epics selection and take two months. The epics are usually themes that cover a wide range of functionalities on a very broad scope used as target for all teams involved in delivering towards the final product. For example, the epic “Phone calls” defines the features that are required to comprise such target. In this sense, at the end of the release “Phone Calls” the product created from this release must be capable of performing all sorts of phone calls along with contacts management, emergency calls and underlying customizations, display and group call history, properly divert the call to a voicemail number or forward it to a previously registered phone number among several other depending functionalities. Usually more than one epic is included in release planning and it is based upon the product management team decision on what has to be prioritized towards the main customers read up, carriers and value-driven approach.

Before the beginning of the release cycle, the teams meet up in an Open Space Coordination fashion, as indicated by Larman and Vodde (2010, 204). The Open space technique encourages teams to discuss issues and dependencies openly and face-to-face in order to organize their duties for the forthcoming release cycle. This meeting is important to mainly prioritize features taking into account dependencies cross-teams might have. The target is to have a shared release backlog ranked by taking into account or cross-functional issues discussed during the open space meeting. The features and stories are vertically selected by each team and also prioritized into their internal backlogs, which will lead more planning meetings under each of the teams.

Each story in the backlog that should be assigned to the current release is flagged with a corresponding release assignment in order to request a consensus towards the addi-

tion to such story or task to the current release. Anyone in the organization is free to request the flag, which release and product management teams work together in order to accept the request along with teams' collaboration. The flagging approach is useful mainly for two reasons:

1. To properly adapt new stories due to feedback from stakeholders and teams;
2. To support follow up from release management team. In this case, the traceability of delivered items is improved once the backlog tool is also connected to the CI System. When a story is finally accepted into pre-release repository, the item with such flag in the backlog tool is marked as VERIFIED. Reports from such tool indicate the progress of flagged items across the release cycle.

Each release cycle is made of two increments or checkpoints for inspection and adaptation. On each increment teams make use of open space to demo their accomplishments from the product release created at the end of the increment. This meeting provides a great opportunity for teams to receive feedback from other teams and all types of stakeholders. They have a chance to adapt to the requested changes and reorganize the forthcoming deliverables and scope for the next increment. Under each increment, teams are free to organize their internal sprint duration. Usually, what has been observed, teams use either a two-week sprint or weekly basis sprint cadence, following also weekly releases, planned by the release team.

Thus, two types of releases are identified. One practiced internally by teams and another exercised by the release team. Yet, both are in synchrony with the product release which aims to keep the outside organization stakeholders aware of the product progress as well as to maintain the whole organization focused on the same targets and intent, by testing, validating and accepting the same release version.

In fact, the program, followed by the release team, has established three releases:

1. The Daily release, which deliveries are possible up to 17:00 on the same working day. After this time integration requests are submitted to next day release. Therefore, when the daily release round is closed, the remaining packages under Staging repository all grouped towards Gate 2 promotion using ci-promoter. After this step, which was already covered on section 5.1, the daily release is

generated by tagging the release repository and starting the validation process, to be covered later in this chapter.

2. Weekly release, generated by the end of working week, on Fridays. On this day, the daily release is replaced by the weekly release. In a weekly release, release management team announces backlog items implemented and errors fixed during the week along with release notes encompassing release information collected from all teams, which means documenting what teams have achieved on a weekly basis. During weekly release creation the pre-release repository is also tagged the same way it was performed during daily release tagging process. The Release team also announces known issues as well as delayed items that were supposed to be implemented during the past week.
  
3. On Increment release, the product reaches an important baseline. In practice, it follows the same procedures as a weekly release but it is regarded as a deadline release for the entire organization. The release announcement covers previous weekly releases announcements and major epics implementation as well as whether main themes were successfully carried out. Concomitantly, release plans are reviewed validating what has been achieved during the increment release.

Figure 35 depicts how product releases and teams release are synchronized together by enforcing the same pace to all teams in the organization:

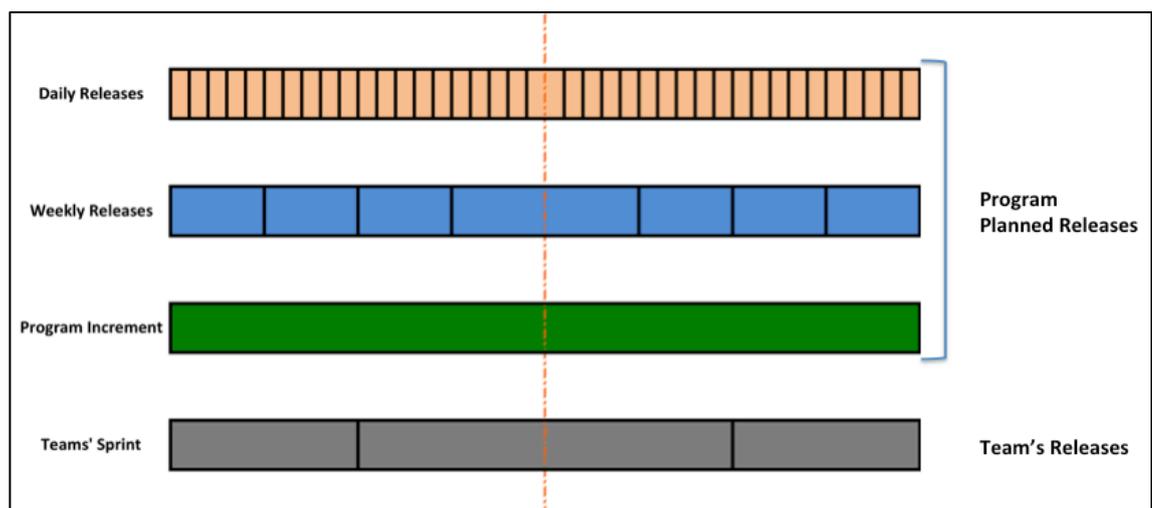


Figure 35: Product Program Releases. Based on: daily, weekly and bi-monthly releases and team Sprints usually set as fortnightly releases.

In fact, there is also a fourth release established when a shippable version of the product is ready to be commercialized (or even to be finally customized by carriers, see Mass Customization chapter). Reaching this level of maturity of the product, means it is ready to be delivered to next stages of product delivery chain. In this matter concerning the case company, the software product is ready to be part of factory assembly line where final software product and hardware are assembled together in the product line. During this stage, called Release Candidate, the product reaches version 1.0 acceptance phase. The special process at this point is towards branching the repository in order to “protect” the release candidate against forthcoming release implementation, i.e., version 1.1 or 2.0, for example. The release candidate is therefore finally verified which corrections and adjustments are performed towards the Release Candidate branch. The mainline branch follows the implementation sequence by delivering features assigned for newer product versions. Figure 36 illustrates the release candidate branch.

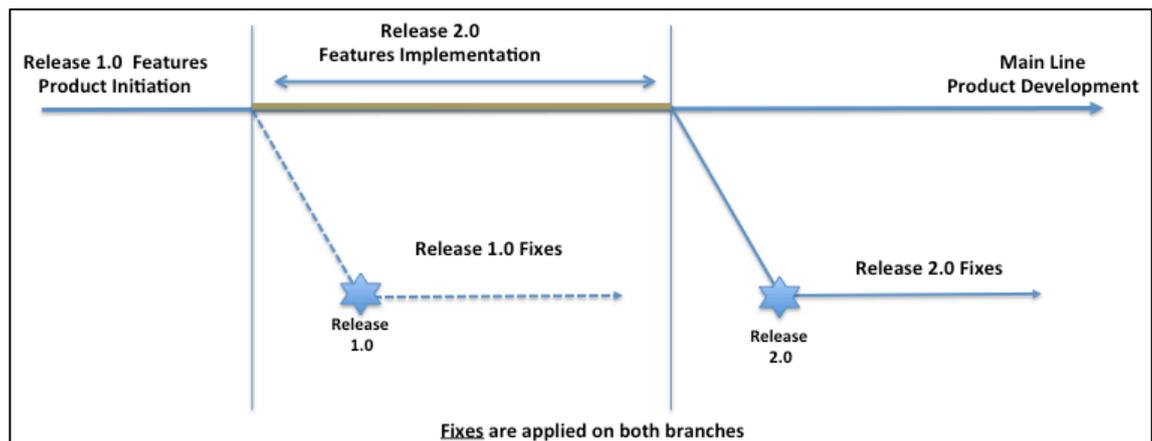


Figure 36: Release Candidates for Product version 1.0 and 2.0 as an example of how product releases versions are managed against product mainline.

The validation process, as part of release process, occurs against daily and weekly releases and aims to provide a verdict towards the release assessing whether it is RELEASABLE or not. This assessment is important for teams to have a common understanding whether the release can be used as baseline for their continuous development or a previous release has to be used instead. It has, therefore, the following goals:

- To make the release better quality since everyone is looking at the same, and the latest, release;
- To achieve fast turnaround by finding bugs during the *validation* process;
- To achieve feature and user story acceptance during the *validation* testing;
- To include regression testing for functionality, which includes re-testing regression bug fixes.

In this sense, *Validation* includes both automatic and manual testing. The validation test set is actually a subset of the overall test set to determine if the software is suitable for release. Each team defines their own validation test set by following the criteria:

- Tests should include features in the current increment and the regression test cases;
- Test sets should be cumulative;
- Test cases should only be dropped in exceptional circumstances.

The validation process is detailed in Figure 37:

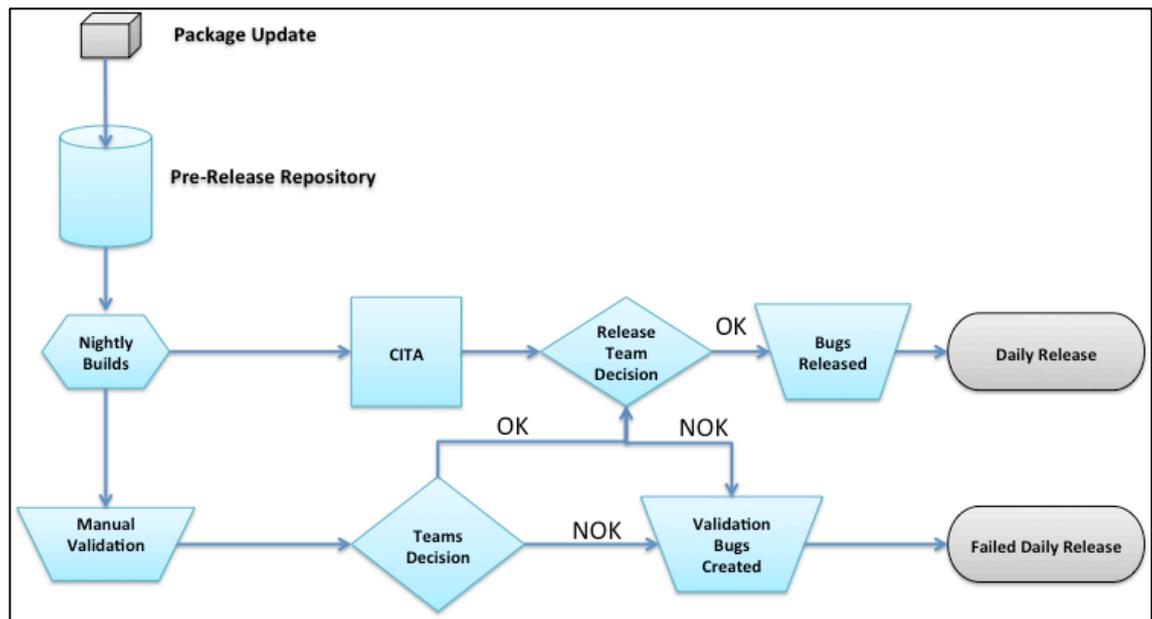


Figure 37: Validation Process including Automated testing and manual validation.

When the Validation process starts, the pre-release repository is snapshotted figuring a candidate for the release on which three product images are created on top of such snapshot. One is based on BRT configuration, the other on the ART configuration (see Types of configuration packages, chapter 5.4.1) and the third based on the product configuration. Both BRT e ART are automatically executed through the Test Automation tool (CITA) and the product image is used for manual validation using exploratory tests.

Validation process aims testing the feature of which the original package is part. If the feature passes the test, it will become part of the pre-release repository and the daily release is therefore approved. However, if the feature tests do not pass, the release will be rejected on that day. Thus, the same approach applies to a weekly release as well.

## 6 Discussion

It is not an easy move to create by changing the status quo of traditional management to agile and lean thinking. On the enterprise scale, the challenge of achieving the full

benefits of agile is significant and mainly relies on the recognition that serious changes in the organization must be addressed. As the enterprise grows, organizational patterns such as policies, procedures, managerial roles and bureaucracy grow with it, running directly into the opposite way from the philosophies that characterize agile and lean. Worse than that, many of these patterns resist changing, either by cultural intransigence or by political games commonly played in large organizations.

Moreover, when scaling agile methodologies to the enterprise level there are also challenges inherent in agile that limit the methodology itself to scale out. However, many authors have addressed such topic and many ideas are now covered and practiced by several organizations. Along with literature, the case company has achieved significant productivity and efficiency improvements when compared to traditional ways of developing and managing software mainly concerning teams motivation and self-management, innovation and fast adaptation through product feedback and throughput.

The investment and adoption of continuous integration, later scaled to multi-stage continuous integration, has paid off when considering the benefits of highly and frequent product progress visibility. It also enabled a higher level of integration and collaboration between teams and cross-functional areas. In this account, continuous integration along with test automation tools were capable of interfacing with requirements and releases areas by providing a thorough automated mechanism that connects the software production line end-to-end.

This approach fosters the creation of a lean and interconnected environment where improvements are easily achieved throughout the perception of wastes and queues in a value stream map towards the integration process by building in quality and continuous optimization of the whole, avoiding “silos” and therefore improving communication between all areas of the organization.

Mass-customization, on the same way, showed a complete shift from previously observed way of execution (legacy from case organization) by breaking several paradigms formerly applied by the case company on other non-agile products development. The main focus of mass-customization was promptly delivering customizable software along with the vanilla product consisting in quickly adapting on customer feedback and competitors introductions.

In this perspective, it is not enough to adopt agile practices, but the tools should be implemented in a way that support such practices and moreover to ease the deployment the process of agile and lean. The Multi-stage continuous integration tools are therefore regarded as the backbone of product development, interoperating with all other practices between teams and sustaining leadership team decisions by providing instant feedback of product issues, impediments as well as progress.

Although the key benefits of an agile and lean organization are significant, there are several impediments as mentioned above. The most important drawback concerns the fact that lean radically impacts every person in every function of an organization by literally changing the organizational culture. Many teams are not able to cope with this magnitude of change and cause delays against the full implementation of lean in the organization. Nevertheless, software organizations have long observed that being lean and agile is a matter of survival in the current marketplace and the recognition of this change is the first step given.

## **7 Conclusion**

The target of this thesis was to provide a clear picture of how a large-scale organization scales agile and lean practices with the main focus on multi-stage continuous integration, which is a key practice to enable the implementation of such scalability. The main reason is the ability of continuous integration practice and tools to inter-operate and integrate between all focus areas in the software product development throughout the organization.

In this sense, the study was focused on describing how cross-functional, self-organized and distributed teams implemented product requirements centralized into an single tool which provided At the same importance, release management, product configurations and mass-customization variants as well as test automation and validation by means of having all of these processes integrated into continuous integration system which provided a higher level of automation bringing noticeable gains to the product delivery flow, projects progress visibility and fostering communication between development teams.

Thus, the results of this study are strictly aligned with the goal set at the beginning of the research work. The goal was to analyze and observe how the case company boosted innovation and created an environment of self-motivated individuals, which were capable of meeting industry competitive challenges building an adaptable, customer-driven enterprise. Consequently, it also emphasized the agile and lean practices required to set up and scale such an organization, stressing the key practice of continuous integration responsible for fostering such scalability and for supporting team collaboration.

Accordingly, as observed and described in the empirical study, the practices covered in all areas linked through continuous integration are significant and can be easily adopted as practices in another large-scale organization with the attempt to promote lean and agile. In such a way, the results of this thesis can be applied to many other software organizations to accomplish the transition from traditional management to adaptable, value-driven and fast feedback concepts, which highlights self-organized teams with highly motivated professionals looking for productivity and innovation.

This thesis focused on observing and describing the facts that made up a scaled agile and lean software organization with the strong support of a Continuous Integration System. This study, in fact, may serve as a baseline for several other researches in the agile and lean software development field. For instance, by taking into account such an environment, the delivery towards product repository can be faced as a Kanban flow process. The resulting Value Stream Map as well as building the cumulative flow diagrams can be analyzed in order to investigate opportunities of improvements, create measurements, and identify bottlenecks and constraints. The main focus would be solving one of the biggest issues in large software organizations, to reduce the software regressions.

Another opportunity of research is also applying Kanban and Lean practices in order to eliminate wastes in the mass-customization workflow, which covers not just the software organization but also marketing, sales and manufacturing organizations and processes. In this case, the problem to be addressed would be focused on drastically reducing the time to market customization features, prioritizing customers adding more value to the chain.

## References

Agile Alliance, Online. *Agile Manifesto*. [Online] Available at: <http://agilemanifesto.org> [Accessed 14 February 2013].

Ahonen, T., 2011. *Ecosystem 1-on-1 and Stephen Elop's Alternate Universe at Nokia*. [Online] Available at: <http://www.brightsideofnews.com/news/2011/6/8/ecosystem-1-on-1-and-stephen-elops-alternate-universe-at-nokia.aspx> [Accessed 15 March 2013].

Ambler, S.W. & Lines, M., 2012. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. Pearson Plc.

Bryman, A., 1989. *Research Methods and Organization Studies*. Routledge.

Crispin, L. & Gregory, J., 2009. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley.

Cromar, S., 2010. *Smartphones in the U.S.: Market Analysis*. [Online] Available at: <https://www.ideals.illinois.edu/bitstream/handle/2142/18484/Cromar,%20Scott%20-%20U.S.%20Smartphone%20Market%20Report.pdf> [Accessed 15 March 2013].

Debian, 2012. *Debian Policy Manual, Chapter 5*. [Online] Available at: <http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source> [Accessed 16 March 2013].

Denning, S., 2010. *The Leader's Guide to Radical Management: Reinventing the Workplace for the 21st Century*. Jossey-Bass.

Duvall, P.M., 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.

Fowler, M., 2006. *Continuous Integration*. [Online] Available at: <http://www.martinfowler.com/articles/continuousIntegration.html> [Accessed 14 February 2013].

Gilmore, J.H. & Pine, B.J., February 2000. *Markets of One: Creating Customer-Unique Value through Mass Customization*. Harvard Business Review Press.

Griffiths, M., 2012. *PMI-ACP Exam Prep*. RMC Publications.

Gruver, G., Young, M. & Fulghum, P., 2013. *A Practical Approach to Large-Scale Agile Development*. Addison-Wesley.

Jeffries, R.E., n.d. *XProgramming.com - An Agile Software Development Resource*. [Online] Available at: <http://xprogramming.com/images/circles.jpg> [Accessed 19 March 2013].

Klipp, P., 2013. *Getting Started with Kanban*. Kanbanery.

Kniberg, H. & Skarin, M., 2010. *Kanban and Scrum making the most of both*. InfoQ.

Kurri, S., 2012. *The story of Nokia MeeGo*. [Online] Available at: <http://taskumuro.com/artikkelit/the-story-of-nokia-meeGo> [Accessed 15 March 2013].

Larman, C. & Vodde, B., 2010. *Practices for Scaling Lean and Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Addison-Wesley.

Leffingwell, D., 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley.

Nokia, 2013. *The Nokia History*. [Online] Available at: <http://www.nokia.com/global/about-nokia/about-us/the-nokia-story/> [Accessed 14 March 2013].

Patton, M.Q., 1999. *Enhancing the Quality and Credibility of Qualitative Analysis*.

Poppendieck, T. & Poppendieck, M., 2006. *Implementing Lean Software Development From Concept to Cash*. Addison Wesley Professional.

Ramloll, R., 2010. *Deep Semaphore*. [Online] Available at: [http://p.blog.csdn.net/images/p\\_blog\\_csdn\\_net/wisdom521/EntryImages/20080821/scrumProcess.PNG](http://p.blog.csdn.net/images/p_blog_csdn_net/wisdom521/EntryImages/20080821/scrumProcess.PNG) [Accessed 17 March 2013].

Raymond, E.S., 1999. *The Cathedral and the Bazaar*. O'Reilly Media.

Ritchie, J. & Lewis, J., 2003. *Qualitative Research Practice: A Guide for Social Science Students and Researchers*. Sage Publications.

Shore, J. & Warden, S., 2008. *The Art of Agile Development*. O'Reilly.

Wikipedia, 2013. *Debian*. [Online] Available at: <http://en.wikipedia.org/wiki/Debian> [Accessed 11 March 2013].

Wikipedia, 2013. *Nokia*. [Online] Available at: <http://en.wikipedia.org/wiki/Nokia> [Accessed 14 March 2013].

Wikipedia, 2013. *Package Management System*. [Online] Available at: [http://en.wikipedia.org/wiki/Package\\_management\\_system](http://en.wikipedia.org/wiki/Package_management_system) [Accessed 26 February 2013].

Wikipedia, 2013. *System Integration*. [Online] Available at: [http://en.wikipedia.org/wiki/System\\_integration#cite\\_note-SICourse-3](http://en.wikipedia.org/wiki/System_integration#cite_note-SICourse-3) [Accessed 28 February 2013].

Yin, R.K., 1994. *Case Study Research: Design and Methods*. Sage Publications.

**Agile Principles:**

Retrieved from Agile Alliance, Online, accessed Feb. 2<sup>nd</sup> 2013:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

## Debian Source Package Description

Figure 38 depicts Tomboy source package for Ubuntu packaging procedures. Tomboy is a desktop note-taking application for Linux, Unix, Windows, and Mac OS X. Ubuntu maintainers retrieve the source for Tomboy project upstream and “debianize” it by encapsulating the source under *Debian* packages descriptor files like .dsc file. This procedure makes easier to build tomboy source package by generating Tomboy binary package. Both source and binaries are available from Ubuntu repositories to be installed under such system.

```

Format: 3.0 (quilt)
Source: tomboy
Binary: tomboy
Architecture: any
Version: 1.10.1-0ubuntu2
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Uploaders: Sebastian Dröge <slomo@debian.org>, Iain Lane <laney@debian.org>
Homepage: http://www.gnome.org/projects/tomboy/
Standards-Version: 3.9.3
Vcs-Browser: http://git.debian.org/?p=pkg-cli-apps/packages/tomboy.git
Vcs-Git: git://git.debian.org/pkg-cli-apps/packages/tomboy.git
Build-Depends: debhelper (>= 7.0.50), dh-autoreconf, dh-translations, gnome-common, intltool,
pngquant, mono-devel (>= 2.4.3), libgconf2-dev, libgtk2.0-cil-dev (>= 2.10.4), libgconf2.0-cil-dev (>=
2.24), libgtksPELL-dev (>= 2.0.9), cli-common-dev (>= 0.5.7), imagemagick, libgtk2.0-dev (>= 2.14.0),
libatk1.0-dev (>= 1.2.4), libgmime2.6-cil-dev, gnome-doc-utils (>= 0.3.2), scrollkeeper, libdbus1.0-cil-
dev (>= 0.4), libdbus-glib1.0-cil-dev (>= 0.3), libmono-addins-cil-dev (>= 0.3), libmono-addins-gui-cil-
dev (>= 0.3), liblaunchpad-integration-cil-dev, libappindicator0.1-cil-dev (>= 0.4.90), libproxy-dev
Package-List:
tomboy deb gnome optional
Checksums-Sha1:
9414e5568ccc4c3b05db0a8b1b88bb94dea7dd81 6619804 tomboy_1.10.1.orig.tar.xz
b53e71e71e73914f887422d1c2eff7b14c44e59c 24331 tomboy_1.10.1-0ubuntu2.debian.tar.gz
Checksums-Sha256:
85bc277b278fe6aaa38fc0ec8b1777804ea0de647111a5065d43614d1ad10077 6619804
tomboy_1.10.1.orig.tar.xz
e7afeb53967c26ef53a4710d5f532026f28299d4d1e7572c4f71b8e010cd74de 24331
tomboy_1.10.1-0ubuntu2.debian.tar.gz
Files:
d8a1b359fa12d414097e0111f87771b3 6619804 tomboy_1.10.1.orig.tar.xz
c658b9700a50bf2a217b99e1acc638fa 24331 tomboy_1.10.1-0ubuntu2.debian.tar.gz
Original-Maintainer: Debian CLI Applications Team <pkg-cli-apps-team@lists.alioth.debian.org>

```

Figure 38: Tomboy .dsc file describing Tomboy source package