



**Bruno Tiago
Carneiro Palos**

**Melhoria das práticas de construção de software:
um caso de estudo
Software construction practices redesign: a case
study**



**Bruno Tiago
Carneiro Palos**

**Melhoria das práticas de construção de software:
um caso de estudo**
**Software construction practices redesign: a case
study**

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática (MIECT), realizada sob a orientação científica do professor Doutor José Maria Amaral Fernandes, professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Mestre Ilídio Castro Oliveira, Assistente Convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho ao meu pai.

o júri

presidente

Professor Doutor José Luis Guimarães Oliveira

professor associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Professor Doutor João Miguel Lobo Fernandes

professor catedrático do Departamento de Informática da Escola de Engenharia da Universidade do Minho

Professor Doutor José Maria Amaral Fernandes

professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Mestre Ilídio Fernando de Castro Oliveira

assistente convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos

Agradeço à minha Mãe pelo amor, pelos sacrifícios que fez e pelo apoio que sempre me deu de forma incondicional desde que nasci e não apenas durante a realização deste trabalho. Agradeço à minha irmã Carla Palos por ser das pessoas que mais torceu por mim. Agradeço à minha namorada Andreia Rodrigues pela sua paciência e apoio, que foram imprescindíveis para terminar este trabalho. Agradeço aos meus amigos, especialmente ao Tiago Pereira, pela amizade durante os momentos mais difíceis do curso. Um abraço aos meus colegas de trabalho que sempre me apoiaram. Agradeço ao meu orientador Prof. Doutor José Maria Fernandes e ao meu co-orientador Prof. Ilídio Oliveira pela paciência, atenção e acompanhamento constantes, sem os quais não seria possível terminar este documento.

palavras-chave

processos de qualidade, práticas, software, integração contínua, automatização, desenvolvimento evolutivo

resumo

Em muitos projetos de desenvolvimento de software não são utilizados processos e práticas explícitos com o intuito de garantir a qualidade do produto final. Nesses casos, a organização do ambiente de construção nasce das acções imediatas do dia-a-dia da equipa de desenvolvimento de forma não estruturada e não escalável.

No contexto dos projetos de investigação com desenvolvimento de software, em que as equipas são marcadamente mutáveis, a definição de estratégias para o processo de construção de software é essencial para agilizar o desenvolvimento, aumentar a produtividade e controlar a evolução do produto.

Este trabalho visa a análise e definição de estratégias para a construção de software usando como caso de estudo o projeto Rede Telemática Saúde (RTS) do Instituto de Engenharia Electrónica e Telemática de Aveiro, e a sua implementação, através da introdução de boas práticas e ferramentas que permitem melhorar a evolução do sistema.

A implementação dessas estratégias inclui disciplinas de gestão de configurações, que asseguram a consistência das versões do projeto e respetivas dependências, e um ambiente de integração contínua que controla todo o código-fonte produzido pela equipa de programadores usando testes automatizados. Cada versão é composta por um conjunto de tarefas ou tópicos atribuídos a cada colaborador que são geridos por critérios de prioridade, alavancando a agilidade do processo de desenvolvimento. Todo o ciclo é representável numa plataforma de gestão dessas tarefas, essencial à gestão de alto nível.

Complementarmente, realizou-se um estudo para caracterizar as práticas correntes no processo de construção de software, através de um inquérito à indústria de software portuguesa.

As estratégias propostas e implementadas permitiram redefinir o processo de construção no projeto RTS, introduzindo um maior controlo sobre a linha de produção, especialmente na identificação antecipada de defeitos e controlo de versões. Estes resultados estão alinhados com as necessidades prioritárias identificadas no inquérito à indústria.

keywords

quality processes, practices, software, continuous integration, automation, evolutionary development

abstract

Software projects often neglect the use explicit processes and practices to ensure final product quality are. On those cases, the arrangement of the construction environment arises from pressing needs of the development team daily routine in a non-structured and non-scalable way.

In the context of research projects that need software development, in which teams are strongly mutable, the definition of strategies for software construction practices is essential to streamline development, increase productivity and to control the product evolution.

This study aims at analyzing and define software construction strategies using as a case study the Rede Telemática Saúde project (RTS) of the Institute of Electronics and Telematics Engineering of Aveiro (IEETA), and their implementation, by introducing best practices and tools that help improving the system evolution.

Such strategies include particular topics of configuration management, which ensure consistency of versions and their dependencies, and a continuous integration environment by validating the source-code produced by developers using automated testing. Every version is composed of a set of tasks or topics particularly assigned to each team member and managed by priority criteria to leverage the agility of the development process. Such tasks and the whole development cycle are mapped on a management platform, which is essential to high-level management.

Additionally, an industry study was carried out to characterize current practices on software construction process, through a survey to the Portuguese software industry.

The proposed and implemented strategies allowed redefining the construction process on the RTS project, introducing more control over the production line, especially on version control and early identification of defects. Those results are aligned with identified priority needs in the industry survey.

Contents

List of figures.....	iv
List of tables	vi
List of tools.....	vii
Acronyms	ix
1 Introduction.....	1
1.1 Context and motivation.....	1
1.2 Objectives.....	2
1.3 Document structure.....	3
2 Background concepts and state of the art	5
2.1 Software Quality	5
2.1.1 Overview.....	5
2.1.2 Detection and prevention	6
2.1.3 Verification and validation	6
2.1.4 Configuration management.....	7
2.1.5 Quality standards.....	8
2.2 Agile development.....	9
2.2.1 What is agile development?	9
2.2.2 Agile methodologies.....	9
2.2.3 Agile manifesto and principles	11
2.2.4 Communication and collaboration.....	11
2.2.5 Development lifecycle	12
2.3 Testing	13
2.3.1 Regression tests	14
2.3.2 Performance and load tests	14
2.3.3 Unit tests	15
2.3.4 Integration tests	16
2.3.5 Functional tests.....	17
2.4 Software configuration management.....	17
2.4.1 Version control.....	17
2.4.2 Software dependencies management.....	19
2.4.3 Tasks management	20

2.4.4	Project elements traceability	21
2.5	Coding improvement.....	22
2.5.1	Code reviews and metrics	22
2.5.2	Coding rules and conventions	22
2.6	Continuous integration	23
2.6.1	Overview.....	23
2.6.2	Continuous feedback.....	24
2.6.3	The build process	25
2.6.4	Continuous testing.....	27
3	Industry survey	29
3.1	Overview	29
3.2	Characterization of the sample	29
3.3	Analysis of results.....	33
3.3.1	Version control.....	33
3.3.2	Tests	36
3.3.3	Continuous integration.....	39
3.3.4	Quality control	40
3.3.5	Task management.....	43
3.4	Conclusion	43
4	A proposal for construction process improvement in the RTS project	47
4.1	The previous construction process.....	47
4.1.1	Construction processes, practices and tools.....	47
4.1.2	Identified problems.....	50
4.2	A new construction process.....	51
4.2.1	The development pipeline	51
4.2.2	Releases maintenance and consistency.....	55
4.2.3	Project progress and rapid delivery.....	58
4.2.4	Assuring software quality	61
4.2.5	Team communication and interaction	64
4.2.6	Summary of the new development process.....	65
4.3	Applying the construction process to other projects	65
5	Implementation of the proposed construction process	69
5.1	Selected tools for the construction process.....	69
5.2	The construction environment pipeline.....	72

5.2.1	The project's files repository.....	74
5.2.2	Managing the software artifacts repository.....	76
5.2.3	The continuous integration server	79
5.3	Software project changes management	83
5.3.1	The versioning strategy.....	83
5.3.2	Applying database changes.....	84
5.3.3	Releasing versions of the project	87
5.4	Automated tests on every build.....	92
5.4.1	Automated integration tests	92
5.4.2	Automated unit tests.....	94
5.5	Automated code verifications.....	96
6	Results	101
7	Conclusions and future work.....	103
8	References	105
9	Annexes	109
9.1	Annex A - Questionnaire form	109
9.2	Annex B - Public appraisal of contributing respondents.....	121
9.3	Annex C - Best practices document.....	123

List of figures

Figure 1 - Quality assurance components [6].	5
Figure 2 - Software configuration management components [6].	7
Figure 3 - Agile delivery framework [1].	13
Figure 4 - The repository and working copies [70].	18
Figure 5 - A traceability information model for a basic agile project [81].	21
Figure 6 - Components of a continuous integration system [38].	24
Figure 7 - Number of employees.	30
Figure 8 - Weight of the software projects.	30
Figure 9 - Project team disposition.	31
Figure 10 - Project team size.	31
Figure 11 - Project age, since the beginning of the activity.	32
Figure 12 - Time in production of the project.	32
Figure 13 - Programming languages used in the projects.	33
Figure 14 - Project categories.	33
Figure 15 - Dependency control.	34
Figure 16 - Capacity to reproduce existent versions.	35
Figure 17 - Use version control servers.	36
Figure 18 - Used version control software.	36
Figure 19 - Use dedicated functional tests team.	37
Figure 20 - Performed test types.	38
Figure 21 - Construction practices versus tests team.	38
Figure 22 - Uses automated continuous integration.	39
Figure 23 - Automated continuous integration frequency.	39
Figure 24 - Phased affected by continuous integration.	40
Figure 25 - Manual integration frequency.	40
Figure 26 - Quality parameters versus continuous integration.	40
Figure 27 – Use of rules, conventions or good practices document.	41
Figure 28 - Development process and methodologies adoption.	41
Figure 29 - Most used development processes and methodologies.	42
Figure 30 - Regular code revisions adoption.	42
Figure 31 - Code revision types used.	42
Figure 32 - Used issue tracking systems.	43
Figure 33 - Impact of the different aspects in software construction.	44
Figure 34 - Task development activity.	47
Figure 35 - Old versioning strategy.	48
Figure 36 - Continuous integration base.	52

Figure 37 - Continuous integration cycle.	53
Figure 38 - Integration activities.	54
Figure 39 - Proposed versioning strategy.	56
Figure 40 - Using CI on the main dev. line versus "integration hell".	57
Figure 41 - Continuous version evaluation.	58
Figure 42 - Issues management.	59
Figure 43 - Using a distribution repository.	60
Figure 44 - New task development activity.	62
Figure 45 - Automatic integration tests.	63
Figure 46 - Simplified RTS' development pipeline.	73
Figure 47 - Subversion's directory structure.	75
Figure 48 - Repository Management scenario on RTS project.	77
Figure 49 - Snapshot dependencies download.	78
Figure 50 - Snapshots repository maintenance.	79
Figure 51 - Hudson integration jobs.	81
Figure 52 - Configured jobs on Hudson.	82
Figure 53 - Using Liquibase for database migrations.	86
Figure 54 - Typical issue statuses.	87
Figure 55 - Shifting issue to another version on Redmine.	88
Figure 56 - Creating a RC branch.	89
Figure 57 - Releasing process steps.	90
Figure 58 - Maven Release Plugin mapping with RTS' releases strategy.	91
Figure 59 - Closing a release on Redmine.	91
Figure 60 - Integration testing.	92
Figure 61 - Using soapUI to create a test suite.	93
Figure 62 - Database automatic migration on RTS dev. environment.	93
Figure 63 - Running integration tests with Hudson.	94
Figure 64 - Unit tests directory structure.	95
Figure 65 - Test result with fails.	96
Figure 66 - FindBugs summary on Hudson.	97
Figure 67 - Problem detail on Hudson found by FindBugs.	98
Figure 68 - Bug found by FindBugs on Eclipse.	98
Figure 69 - Duplicate code detection example.	99
Figure 70 - Test coverage of CitizenCardParser class.	100
Figure 71 - Best practices document's structure.	101

List of tables

Table 1 - Load and performance tests frameworks.	15
Table 2 - Unit testing frameworks.	16
Table 3 - Functional testing frameworks for Web applications.	17
Table 4 – Number of projects using dependencies management software.	34
Table 5 - Projects that can reproduce versions.	35
Table 6 - Quality parameters in function of time in production.	45
Table 7 - Quality parameters in function of number of participants.	45
Table 8 - Diagnosis of dev. practices implementation in previous development process.	49
Table 9 – Problematic practices and suggested actions.	65
Table 10 - Version segments.	83

List of tools

Build	Description
Apache Ant [99]	Automated build tool for Java.
Apache Maven [72]	A build manager for Java projects. Supports automation.
Code static analysis	Description
Checkstyle [86]	Analyses coding standards and conventions.
Cobertura [87]	A Java code coverage analysis tool.
FindBugs [84]	Analyses Java bytecode for potential bugs.
PMD [85]	Scans source-code and looks for potential problems.
Continuous integration	Description
Apache Continuum [92]	A continuous integration server for Java projects.
CruiseControl [90]	A wrapper for Ant for automating builds on Java projects.
Hudson [88]	A flexible continuous integration server.
Jenkins [89]	A fork from Hudson project.
Luntnbuild [91]	Build automation and management tool.
Database migrations	Description
autopatch [114]	An automated database-patching framework for Java.
DbDeploy [112]	A database change management tool.
Liquibase [115]	An open-source database change management tool.
migrate4j [113]	A database migration tool for Java.
Dependency Management	Description
Apache Archiva [110]	Extensible build artifact repository manager.
Apache Maven [72]	A build manager for Java projects. Supports automation.
Artifactory [108]	Artifacts repository manager.
Ivy [74]	Dependency manager for Ant.
Nexus [109]	Lightweight artifacts repository manager.
Functional testing	Description
actiWATE [62]	Java-based tool for automating tests on Web applications.
IeUnit [63]	Testing framework for Web browsers
Selenium [45]	Framework for automating Web browsers.
Watir [61]	Automates Web browsers.
Integration testing	Description
Arquillian [57]	Integration testing framework to run on the JVM.
Citrus [59]	Testing framework for SOA applications.
Gint [58]	Groovy based integration testing framework.
soapUI [46]	Multi-purposed testing framework for Web technologies.

Issue Tracking System	Description
Bugzilla [75]	Bug tracking framework.
Launchpad [76]	Software collaboration platform. Includes bug tracking.
Redmine [78]	A flexible project management web application.
Trac [77]	Project management and bug/issue tracking system.
Load/performance testing	Description
Apache JMeter [32]	Server performance testing tool.
FWPTT [30]	A Web load-testing framework.
Grinder [29]	A Java load-testing framework.
Multi-Mechanize [31]	A Web performance and scalability-testing framework.
Pylot [33]	Open-source tool for testing performance of Web Services.
Siege [34]	
Version Control System	Description
Bazaar [68]	A scalable version control system.
CVS [67]	An old open-source VCS, which is widely used.
Git [66]	A popular distributed VCS designed to be scalable.
Mercurial [69]	A distributes VCS, dedicated to speed and efficiency.
Subversion [65]	A popular open-source VCS.
Unit testing	Description
Jasmine [52]	Behavior-driven framework for testing JavaScript code.
JSUnit [50]	Unit testing framework for client-side JavaScript.
JUnit [47]	A popular Java unit-testing framework.
Mockito [49]	A mocking framework for Java.
QUnit [51]	A powerful JavaScript test suit.
TestNG [48]	A testing framework inspired from JUnit and NUnit.
Other	Description
Cargo [118]	A thin wrapper that allows manipulating J2EE containers.

Acronyms

API	Application Programming Interface
AS	Application Server
CI	Continuous Integration
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
DB	Database
DETI	Departamento de Electrónica, Telecomunicações e Informática
IEETA	Instituto de Engenharia Electrónica e Telemática de Aveiro
ISO	International Organization for Standardization
ITS	Issue Tracking System
J2EE	Java Platform, Enterprise Edition
JAR	Java Archive
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OS	Operating System
POM	Project Object Model
QA	Quality Assurance
RC	Release Candidate
RTS	Rede Telemática Saúde
RUP	Rational Unified Process
SCM	Software Configuration Management
SME	Small Medium Enterprises
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SVN	Subversion
TDD	Test Driven Development
UA	Universidade de Aveiro
UP	Unified Procees
VCS	Version Control System
WAR	Web Application Archive
WS	Web Service
XML	Extensible Markup Language
XP	Extreme Programming

1 Introduction

1.1 Context and motivation

In software industry, costumers assume development teams to promptly meet their expectations. As software applications are products of great complexity, which can attain countless different states, the need to immediately solve unforeseen problems is critical to ensure the trust of stakeholders. This unpredictability must be embraced by development teams as something natural[1], compelling them to equip themselves with rapid response skills because it is impossible to predict when a bug will be detected on applications (which may occur days, weeks or even months after installation on the client). For such reason, the ability to reproduce and fix production releases must be part of teams' skills, requiring agility in execution, from detecting the problem, through its resolution to the deployment on runtime environments.

Beyond the ability on maintaining software releases, development teams must ensure minimum defects to be introduced into the solution and new features to be developed in tune with costumers expectations, ensuring constant production quality. Yet, orchestrating all these guarantees, given the software unpredictability and regarding all deadlines, may ascertain an extremely complex task to achieve. Therefore, during development periods of new releases, progress must be frequently analyzed and priorities must be continuously assessed and changed [2]. This flexible approach must be a daily basis process and the development staff should grasp it.

This work is carried out under the Rede Telemática Saúde (RTS) [3], [4] project of the Institute of Electronics and Telematics Engineering of Aveiro (IEETA) - a solution to share medical information about patients among health institutions. The project has the peculiar trait of highly transient development staff, keeping developers for short periods of time, at most a year with minimal number of resident developers. The project has periods of inactivity, depending on the focus of research by IEETA. Usually, developers are students of the Department of Electronics, Telecommunications and Informatics (DETI) at University of Aveiro (UA) working at part-time, leveraging the lack of communication among individuals. People do not regularly interact due to the academic environment characteristics, where software development processes are not subject to strong investments and development teams are usually uncoordinated. Thus, two needs arise to mitigate the problem: a fast integration of new developers and simplifying the development team's common language.

An extra characteristic of the project is the lack of an independent tests team to carry out project's releases validations throughout the iterations. Therefore, the development team itself must ensure the product implementation and product validations prior delivering it to customers.

The key is to streamline validations duration and effort by adopting automated tests (unit testing, integration testing, regression testing, etc.). Since RTS handles sensitive health information, such checks must obtain proper care.

The proposals submitted on the present dissertation are expected to improve the entire construction practices of the RTS project by speeding and to ease the application availability to customers, ensuring stakeholders' expectations to be fulfilled. It is important to note that this document centers on merely a few processes of the Software Quality Assurance (SQA) and does not discuss the entire spectrum of each. The processes and practices covered relate to version control, code construction, releases management, continuous integration, code reviews and certain kinds of tests.

1.2 Objectives

This work proposes enhancements on software construction practices, using the context of the Rede Telemática Saúde (RTS) development environment - a research project with a highly rotational team of developers - as a case study. Despite focusing on a particular project, practices and processes defined in this work are also pertinent for use by other software projects in general but the selection of technologies just applies to systems developed in Java programming language.

The main objective is to improve the project building process by designing and to implement a developmental pipeline using appropriate tools.

The work will start by establishing a baseline of the current development practices and to identify potential problems. This assessment will be performed in detail on the RTS case study but a wider observation will also be conducted on the Portuguese software industry through a survey (see Annex A).

Some of the agile principles will be applied to propose improvements that favor the overall production performance in view of the deliverables' quality and endowing the project with constant and rapid delivery capabilities, either during construction phase or during maintenance situations of production releases, deployed on costumers machines without ever losing control, nor losing the coveted quality.

We also intend to produce a functional best practices document towards aligning the entire team to adopt the suggested construction processes and practices (see Annex C). This document also specifies some rules and conventions for improving communication between individuals, increasing the quality of the source code and improving the project maintainability.

1.3 Document structure

The following describes the summary of the next chapters:

- Chapter 2 – Background concepts and state of the art:

An overview on software development concepts and quality assurance processes applied to software production is introduced. Also, a review on software engineering tools, techniques and methodologies is presented.

- Chapter 3 – Industry survey:

Presents the results and conclusions of an industry survey carried out on the Portuguese small and medium-sized enterprises that produces software systems.

- Chapter 4 – A proposal for construction process improvement in the RTS project:

On this chapter, the old construction process used on the RTS project is analyzed and a proposal of a new one is presented, with the intent of improving the overall developmental practices and to assure the quality of the final product.

- Chapter 5 - Implementation of the proposed process:

The implementation of the proposed construction process on Chapter 4 is presented, where tools and techniques are materialized and the specific development environment is defined. On this chapter, particular implementation use-cases on the different aspects of the new development pipeline are also described.

- Chapter 6 – Results:

The industry survey results are discussed and the success of the construction process implementation is analyzed.

- Chapter 7 – Conclusion:

This chapter discusses a comparison on the outcome of the proposed construction process and the ones used by Portuguese industry. Mappings between the agile methodologies and the proposed practices are described. Finally, the chapter discusses possible future improvements on this work.

- Annex A – Questionnaire form:

Questionnaire sent to Portuguese companies to conduct the industry survey.

- Annex B – Public appraisal of contributing respondents:

Acknowledgments to all the cooperating companies. Includes a list with the companies that agreed to incorporate their name in the public appraisal of their collaboration.

- Annex C – Best practices document:

Document for use by the RTS development team in order to improve the development practices, developed in the scope of this work.

2 Background concepts and state of the art

2.1 Software Quality

2.1.1 Overview

Prior initiating some quality concepts definition, the quality concept itself will be featured. Quality, according to Oxford's Dictionary, is "the standard of something as measured against other things of a similar kind; the degree of excellence of something" [5]. Different people in different circumstances regard the concept of quality differently. Broadly, quality is seen as a product or service characteristic that customers expect [6], which in this case applies to software industry. Therefore, customers define quality with their satisfaction, according to previously established needs [7]. In quality processes of software systems, there are two concepts, often confused on distinctness: quality assurance and quality control.

Software Quality Assurance (SQA) activities are held on a particular methodology (procedures), which provide evidence about how the whole software system is suitable for end use and about compliance with specifications (Figure 1). Those activities are used to control and monitor software development processes to attain the projects' specific goals with a certain level of confidence in quality terms [6]. SQA also enables checking and assessing how processes are running and if they are being well accepted by development teams, using process reviews. This enables organizations to check if products follow internal standards, templates and conventions [8].

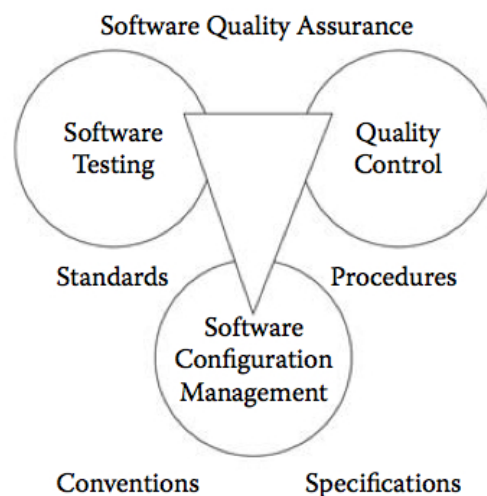


Figure 1 - Quality assurance components [6].

Software Quality Control is about processes and methods definition used to monitor software development and to verify whether requirements are being fulfilled, focusing on removing defects from products before they are delivered to customers [6]. That is, it evaluates if software products are within the defined quality standards resorting to formal inspections, walkthroughs and different kinds of tests [8]. Quality control aims to detect and fix defects, whereas quality assurance enables to prevent them.

2.1.2 Detection and prevention

The commonest approach to improve software quality is detection and consists in thoroughly test software systems until all defects are found and fixed. That is the purpose of detection and allows finding large amount of problems but it is not possible to test entire software systems. There are plentiful ways of detecting defects in software applications: static analysis tools, automated testing, system testing and functional testing [9]. Those detection techniques are used on many software projects but although several faults are found is nowhere to be the most effective way to enhance the quality of software systems.

Applying just fault detection techniques, quality cannot be reached in an already advanced development stage of software applications. It is therefore necessary to prevent defects to be inserted into the solution during development progress [6]. This can be achieved by defining development processes using methods, techniques and tools. Prevention allows reducing costs of quality management, maintenance and production by decreasing defects count, which might be found on customer environments. This quality improvement degree is the most effective and can be optimized with gained experience on detecting defects. With that experience, development processes and practices can be improved and defects root causes can be eliminated in order to prevent re-occurrence. Prevention helps improving the overall product quality and end-user satisfaction, since defects have detrimental effects on the final solution's image. Prevention also helps to decrease development and maintenance expenses, since defects increases costs with time they pass unnoticed and shifts problems correction to the initial stages of development. The identification of defects root causes and prevention techniques appliance enable foresee ability and improves production processes, since defects are unplanned events that can slow down iterations completion and releases deployment on costumers' machines [10].

2.1.3 Verification and validation

As means of quality control, verification and validation guarantees agreement with client requirements definition when software development process is conducted. However, both processes have different roles during software development lifecycle. Verification is the process to guarantee the product is being developed in line with requirements at the end of each phase during

the development cycle. Validation guarantees consistency between software product result and costumers' expectations throughout development cycle, and if the application's behavior conforms with specifications [6].

Typically, tests are performed during software validation process, namely, at the end of the development cycle, right after programming stage, to check if the system is compliant with the result that should be delivered to customers at functional and operational levels [6], [8]. Verification process occurs during development cycles, not just at the end of each stage. Verification proves that each phase is being developed properly, from requirements specification to construction [8]. Verification practices include: reviews, analysis and testing of project items, such as requirements, design, architecture and code.

2.1.4 Configuration management

In general, Software Configuration Management (SCM) is a suite of management practices that enables to reproduce future product versions, allowing teams to remove defects from those versions [9]. Enables distribution of software applications by using version control mechanisms in order to implement all its objectives [11]. SCM relates to change control of software applications, and its components relationship through time. SCM is a discipline that can be applied to software development, documentation, problem tracking, change control, test cases, external dependencies and versions maintenance. Allows software reuse because each component and their mutual relationships are defined, thus reducing production costs. Helps organize software development and prevent inadvertent introduction of unwanted changes [6]. Its stabilizing effect provides consistency among different software components, especially between source code, documentation, requirements and product validation actions, while maintaining versions integrity and sustaining configurations traceability throughout the implementation process. It also allows managing how individuals collaborate during software production [12].

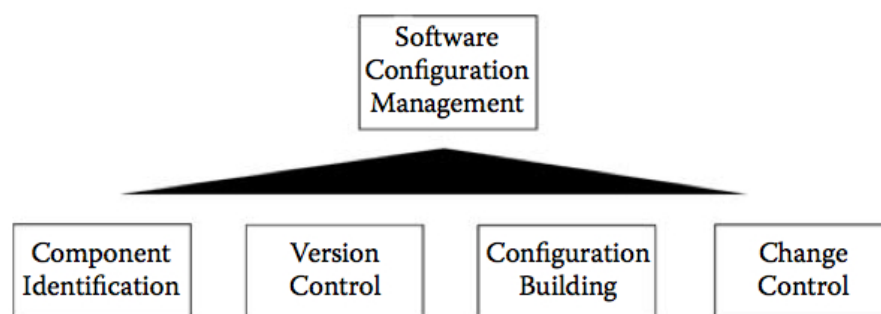


Figure 2 - Software configuration management components [6].

There are four major groups on SCM: version control, configuration building, change control and component identification [6], [12], [13] (Figure 2). Version control is the most used, which allows managing different software versions consistently through project evolution.

Configuration building concerns linking each source component to obtain complete software derivations. This SCM element defines the way software components are put together. An example is external dependencies management, which consists to organize external libraries and external binary files used by software applications. SCM must ensure consistency between software versions with change management of such dependencies. Change control is the process where all software changes made are proposed, evaluated, approved, rejected, scheduled and tracked [6]. Component identification consists on identifying software components at each point of its development, such as the software baselines, components and configurations. Software suffers a series of changes and methods must be established to uniquely identify each revision [6]. An example is the version schema, depicted by literals, which usually use incrementing numbers for each release evolution.

2.1.5 Quality standards

The following quality standards shall apply to development projects and software systems and are references towards improving production processes and enhance the ultimate quality of what is delivered to customers.

- Capability Maturity Model (CMM):

It is one of the oldest models for software and was described by Watts Humphrey (1989) [14]. The main purpose of this model is to evaluate the capabilities of organizations to accomplish a defined set of processes, but is also a continuous improvement driver. As the name suggests, outlines a series of maturities to organizations exercise over specific elements of the model [9]. There are five maturity levels, each one with an aim, based on milestones achieved on previous levels, allowing stepwise improvement procedures. Each level is incremented whenever a number of key practices are reached by completing the former level maturation [6].

- Capability Maturity Model Integration (CMMI):

The model described previously applies to organizations that produce software. CMMI is a more complex model that stretches CMM to be appropriate to organizations that develop software and systems engineering [9]. Provides a collection of practices for developing products and services, ensuring their maintenance [6], [15]. This model adds more information to the previous one, which is reflected in the structure of its maturation levels. Organizations who implement this model claim that is suitable for process improvement [9].

- ISO9000:

Published by the International Organization for Standardization (ISO), ISO standards are widely used by countless organizations that provide products and services [9]. They are more suitable for non-IT organizations but the generic nature of such standards allows them to be

tailored to software industry by using flexible rules, allowing interpretations that apply to software industry. ISO9000 establishes a series of quality standards, including ISO9001, which implement a number of guidelines to normalize quality parameters, namely to ensure that organizations have abilities to design and produce high quality products, retaining control over development activities and providing capacity to detect and manage non-compliances during inspections and tests [16].

- ISO 15504:

Also known as SPICE (Software Process Improvement and Capability Determination), is a similar model to CMM, containing six levels of maturation. Deploys a collection of processes broken down into categories, which define how to manage customer requirements, software products implementation, operations over the organization's solutions, project management and process improvement [17].

2.2 Agile development

2.2.1 What is agile development?

Agile methodologies enable developing software applications in a highly collaborative fashion [18]. Agile development methods use clearly defined timed-iterations, evolutionary development, adaptive planning and evolutionary deliveries. Those practices enable teams to quickly respond to constant changes, encouraging agility. Since there are different agile development methods and practices, it is not possible to define them precisely but they all share common characteristics: short iterations, adaptive plan refinement and development evolution. Additionally, they promote principles and practices of agility and simplicity, communication and focus on programming instead on documentation [2], [19].

The concept originated in the 1990's when several emerging methodologies were calling the software development community's attention. Each of these methodologies, with different combinations of old and new ideas, valued the close collaboration between individuals, close communication over written documentation (less efficient), frequent deliveries and new business values; small and self-organized teams; and new ways to produce code and to mitigate requirements without being troublesome [20].

2.2.2 Agile methodologies

A methodology is something to ease software production. Includes the description of how teams work, who is doing what, how they interact and what they produce. Every organization has a methodology and is how they do business. A precise definition of agile methodologies cannot be

found, because specific practices vary. However, they all share the use of short iterations, adaptive evolution and plan refinement. All of them foster practices and principles of simplicity, communication, lightness, self-organized teams, etc. [2], specified above. Three of the most popular methodologies in agile software development will be introduced next.

- Unified Process (UP):

For medium sized projects, this method recommends using iterations enduring two to six weeks. It is characterized by being highly flexible on process formality and documentation. The flexible nature of its optional work items allows it to be adopted by small projects of three people up to large critical projects involving hundreds of developers [21]. It is a broad and ambiguous iterative process that allows being continuously refined, yielding particular methods such as RUP (Rational Unified Process). Requires to be "tailored" for every project because it defines more than 50 optional work items. These items must be selected in order to mitigate risks and goals for specific projects. Despite the numerous optional work items, the recommendation is "less is better" [2].

- Scrum:

Like UP, Scrum is a flexible methodology with respect formality of its work items for use on projects. As a guideline, the choice should be towards the lowest possible use of such artifacts. In Scrum, teams will decide the formality degree for documents, not the manager. Teams consist of few elements (seven or less), however, several teams may work on the same project, with option of scaling to hundreds of developers (scrum of scrums) [22]. Team elements work physically close and held a daily meeting to monitor iterations progress. Its flexibility enables it to be used for informal projects, or be used in large life-critical projects. Scrum can be combined with other iterative methodologies, since its emphasis is on a set of management practices rather than on requirements, or implementation [2].

- Extreme Programming (XP):

Is an incremental and iterative development (IID) method that values customer satisfaction by producing high quality software, sustainable development techniques and flexible response to changes. It is suitable for small sized teams with very short iterations - one to three weeks. Provides quite explicit programming methods to quickly respond to changing requirements without denigrating the code quality. This includes test-driven development, pair programming and continuous integration [2]. The XP distinguishes itself by not require detailed definition of its work items, except for the code and tests. Thus, oral communication to specify requirements and design, rather than detailed paper documentation, is valued. It may seem a messy method but proves to be highly effective in projects with referred characteristics but it requires constant practice and discipline [23].

2.2.3 Agile manifesto and principles

In 2001, 17 representatives of different development methodologies signed the agile manifesto where they reached consensus on four core values (the elements on the left are more valued) [20]:

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

It thus becomes clear the focus on people and teams, as opposed to paper as a way of communication, where the quality of personal interaction is more valuable. Documentation may be critical but its overuse does not prove the value of the developed product: a working application is the sole way to present a tangible result. Interpersonal collaboration is highly valued because the value created by the closeness between teams, customers and managers boosts the ability to quickly adjust unforeseen and unplanned situations, making requirements changes during development period as something to be faced as natural. That is, a rigid plan is useful until lies outdated, rendering it useless [24]. For this reason, on agile methodologies, constant changes on requirements should be accepted.

Additionally, the group of 17 described a set of principles that define more accurately the agile manifesto [20]. These principles may well evolve in the existence of new points of discussion [24]. Some of these principles are implicitly described in the following sections.

2.2.4 Communication and collaboration

A team defines itself by having a purpose, for being composed of two or more persons and for being subject to coordination by these people [25]. Self-organized teams shape the basis of agile project management. They interweave concepts as freedom, responsibility, flexibility and structure. In addition to establishing self-organized teams, each element is also self-disciplined, which allows, in the event of ambiguities or inconsistencies throughout development, teams themselves to decide the path of project's vision complying with the project's constraints [1]. In self-organized teams, work can be switched between individuals, according to needs and strategy for each iteration. Individuals are responsible for managing their own activities, which are not decided by managers, as opposed to "command-control" management system. This approach does not imply lack of a leader, just supplies more leeway to teams, being managed with a different style. Thus promoting cultures of self-organization freeing inventive spirit of individuals, providing more value to customers than traditional management projects [2]. As stated in the agile principles, the best architectures, requirements, and designs arise from self-organized teams [24].

Skills of agile teams are based on collaboration, interaction and cooperation of all members to produce value. This result can be defined as a functional product, a decision or a shared knowledge [1]. Agile teams being self-organizing and self-directional, encourage the adoption of regular meetings to reflect on how to improve development efficiency based on experience gained in previous iterations, adjusting next iteration according to needs. Those intervals enable settlement of conflicts and dependencies in real time [2], allowing constantly measure progress.

Other encouraged approach of collaboration is amongst development teams and those who understand business - customers. Customers should form an integral part of agile development effort. While development staff and customers have different roles, bundling all involved individuals in the same team will foster the improvement of delivered results. This kind of collaboration can be difficult to attain, becoming a barrier to efficient implementation, so there may be need to identify specific roles on customer's edge, such as product expert or product manager [1].

2.2.5 Development lifecycle

Agile development is focused on delivery. Early deliveries allow obtaining early feedback on requirements, teams, and processes. They help teams feel continued compliance with objectives and also enable project adjustment throughout iterations to drive it into desired directions, improving the value delivered to customers. The frequency of deliveries may be agreed with clients, depending on their availability, bearing in mind that the more shorter iterations, the better [24].

On agile projects, constant requirement changing should be embraced, even during an advanced state of development. Constant adaptation to change on software projects helps grant competitive advantage to customers, even during periods of business crisis because the product is valuable at an early stage of development. Despite the intended objective to be common among different agile methodologies, specific details about mechanisms for mitigating constant changes during development differs in specificity [24]. Nevertheless, adaptation to change is only doable when there is an ongoing process assessment of development over iterations. Such progress is measurable only when the outcome product of each iteration is a running application, not by documents describing promising plans.

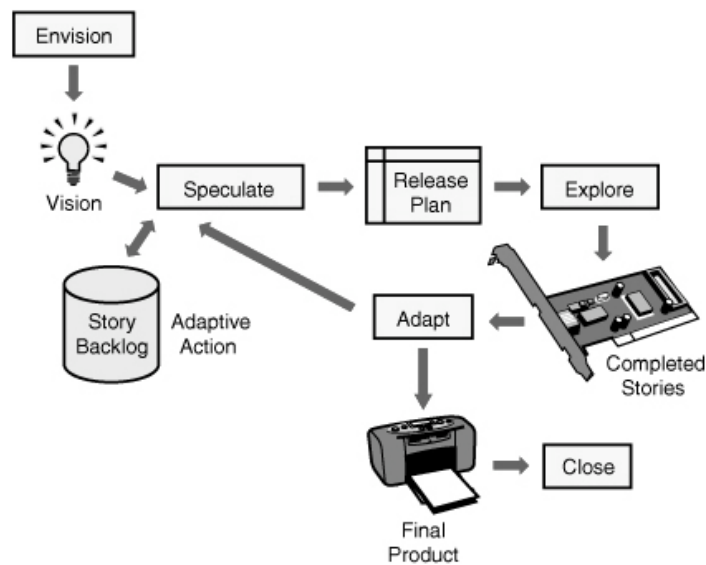


Figure 3 - Agile delivery framework [1].

Regarding mentioned considerations about agile development, and given the aim to produce software in a repetitive fashion, the specification of a development process is justifiable, always focusing on people and interplay between them. Nevertheless, the process must be organic, flexible and easy to adapt [1]. The management structure of agile projects is focused on delivery and adaptation, whose cycle is shown in Figure 3. The cycle starts at "Envision" which determines the critical factor for project success - the vision. Goals and constraints are defined, who is going to be involved, who the customers are and how the team will interact during development. The next stage is denoted by "Speculate" and relies on incomplete information to define the initial requirements, the iterative deliveries plan and risk mitigation strategies. Financial costs are estimated and other administrative aspects are set. Following is "Exploration" phase where the project vision is planned and implemented during small iterations, mitigating risk and uncertainty. This phase is where self-organizing teams and people collaborate to create value for customers. The achieved result in the previous phase is revised during "Adapt" phase, which evaluates team's performance and necessary adjustments are performed. The outcome of adaptation phase is part of a planning effort to initiate the next iteration. This cycle closes between Speculate-Explore-Adapt phases, where each iteration will refine the product. Eventually, the project will end up with a final iteration where it learned aspects are identified that could benefit future projects [26].

2.3 Testing

Software systems testing are inherently connected to software quality control because they are means to scrutinize whether those systems meet the minimum standards required by stakeholders. They also check if the solution correctly implements all specified requirements and help measuring the process that took place to develop the systems and thereby improve them. The

goal, in a wider view, is to stop defects to be introduced into production, preventing belittling the product image among customers, resulting on business losses [6]. This process of quality assurance (QA) aims to detect defects as early as possible, ideally as soon as they are introduced during development stage, reducing overall expenses as the cost of fixing an undetected bug increases exponentially as time elapses [27] and may become even more expensive than the implementation of the requirement that produce it [6].

There are two kinds of testing: black-box testing and white-box testing. Black-box testing consists of tests that do not require aware of implementation details and white-box testing, on opposite, are tests that require knowledge of the inner workings of software modules to be tested [27]. Different types of tests will be presented further, on subsequent sections.

2.3.1 Regression tests

Regression tests are set of tests created gradually as applications evolves through time to verify if the implemented requirements continue to work after new increments to the source-code, even if there is no direct relationship with the functionality originally developed [10]. Regression problems should be detected the earliest possible to prevent the cost required to eliminate them to increase, therefore they must be carried out on regular basis. If regression issues cause a defect, they should be fixed as quickly as possible to prevent them from spreading further. Regression testing also allows verifying whether the behavior of certain features has not been compromised after changes to the project code [12].

2.3.2 Performance and load tests

Performance and load tests are used for measuring and evaluating software systems' behavior in situations of regular use in situations of extreme load. On the first, performance testing enables development teams to adjust and record response times of applications. Moreover, load testing subject systems to extreme conditions leading to breakage of resources by exhaustion [10]. The subjection of the application to these types of tests can help measure their behavior and thus set a criteria of use for which the system can operate. So it is important that such examination would be performed immediately after the birth of the base architecture, in order to be able to perform appropriate changes for the case of being inappropriate for the defined performance objectives [28].

Table 1 shows the most relevant and popular load and performance tools for performing such kind of testing over different technologies.

Load/performance Tests Framework	Target technologies
Grinder [29]	Java, HTTP, WS, RMI, JMS, etc.
FWPTT [30]	Web applications, Web browsers, HTTP, etc.
Multi-Mechanize [31]	Web, remote services, Python.
JMeter [32]	Java, Databases, JMS, LDAP, FTP, WS, etc.
PyLOT [33]	Web Services, HTTP.
Siege [34]	Web applications, HTTP, HTTPS, etc.

Table 1 - Load and performance tests frameworks.

2.3.3 Unit tests

A unit test is a piece of code written by programmers that tests a small portion of business code to prove that those developments behaves in line with the objective for which they were created [35]. When testing all the units individually, much of the errors that might be introduced into the source-code throughout the evolution of applications can be detected nearly at the time they are introduced into the development pipeline and thereby prevent them to propagate through the rest of the system [10]. Besides all the advantages inherent to testing code in small units, this technique used consistently throughout the project forces the use of development best practices which lead to improved implementation of the solution design [36], helping to detect issues earlier in this construction phase. The high confidence established by unit testing and by bugs detection, caused as result of regression problems (for example), does not dispenses the adoption of other kinds of tests [35].

The term “unit testing” is commonly misused and may cause confusion, even on developers themselves. Typically a unit test has a one to one relationship with one class to be tested but occasionally this relationship can be extended with mock objects to mimic complex behaviors [37]. This subordination relationship must be simple and without depth hierarchy of objects. Typically those mock objects emulate databases or file systems but if instantiated on tests, instead of emulated, are no longer regarded as atomic and singular unit tests, but integration tests. It is crucial that unit tests remain simple, that run without setting up runtime environments, without resort to any configuration files and, preferably, without dependencies to other objects [28], otherwise, the project build execution time will increase, denigrating the effectiveness between the time of development and checking the results [38].

As described earlier, regression tests are used to find out whether specific functionalities continue to work after new developments. Unit tests can perform a major role in checking regression issues of a project due to their size and scope [9], especially when the source of errors is the amendment of other defects in code. The described achievement is only valuable if unit tests cover pertinent areas of the source code, if the full coverage is not possible.

Unit tests should be regarded as the lowest level testing kind, as a first line of defense against defects in the application. That line must only cover testing of individual interfaces one at a time, with no external dependencies (except when using mock objects) [28].

Table 2 shows the most popular unit testing frameworks used by software development community for the most popular programming languages and technologies.

Target technologies	Unit Testing Framework
C	Check[39], CMockery[40]
C++	CppUTest[41], CppUnit[42], Google Mock[43]
Internet	HtmlUnit[44], Selenium[45], soapUI[46]
Java	JUnit[47], TestNG[48], Mockito[49]
Javascript	JSUnit[50], QUnit[51], Jasmine[52]
PHP	PHPUnit[53]
.NET	NUnit[54], MSTest[55]

Table 2 - Unit testing frameworks.

2.3.4 Integration tests

Integration testing combine more than an external dependency, whether they are classes, databases or even file systems, typically created by multiple developers or teams of developers and tested in isolation from the entire system [27]. They check if interactions of all components are performing the expected behavior and should begin as early as possible, as soon there exist more than one component in the application, so that architecture is validated early and potential issues are timely fixed to prevent later changes, avoiding development costs to increase [38].

Integration tests (or component tests) are often mistaken as unit tests but it is necessary to split both roles. Integration tests are dependent on other components to be executed, with opposite to unit tests that don't rely on dependencies, and should only use fragments of the system architecture without resorting to the client tier interfaces [38]. Unit tests, which certify code interfaces, should be designed to run on the programmer's machine and be incorporated in the development build to be fast, flexible and portable to every workstation [56]. Integration testing, by contrast, verifies if all interfaces work together as supposed to and they not require to be carried out as quickly and as often as unit tests. They may require some sort of configuration or deployment action to an integration environment [28].

In tests which involve APIs, Web Services, file systems and other code interfaces, databases might be clustered on most of integration tests and even belong to a set of verified components crossing multiple architectural layers. Thus, a higher code testing coverage is achieved by each test [38].

There are several ways to perform integration tests but there available frameworks to perform integrations tests: Arquillian[57] for JBoss AS, Gint[58] for Groovy, Citrus[59] for SOA applications, and others.

2.3.5 Functional tests

Also known as acceptance tests, functional tests ensure that acceptance criteria are met, validating every attribute of the system according to functionalities, capacities, safety, availability, etc. [12], in harmony with end users expectations. This sort of tests typically are conducted by a team which attempt to validate if the application is in agreement with the functional requirements, by using the user interface in a similar environment to production environments. Ideally, these tests should also be automated.

As is required to test the application in a production-like environment from the human user interface, it is difficult to validate it in an automated approach [60]. Nevertheless, software systems can provide interfaces that assist the automation of functional tests. For instance, web applications, which allow a good separation between the features and the presentation layer, allow the capture of the graphical objects that make automation easier (see Table 3).

Functional Tests Framework	Target technologies
Selenium[45]	Web browsers
Watir[61]	Web applications, Web browsers
actiWATE[62]	Web browsers, HTTP, JavaScript, etc.
IeUnit[63]	Web applications

Table 3 - Functional testing frameworks for Web applications.

2.4 Software configuration management

2.4.1 Version control

The version control concept is commonly mistaken as the source-code version control but software versions are more than that. A version of an application is a reproducible state at any point in the time, including all artifacts that made possible to create that version, like the source-code, test code, database scripts, build and deployment scripts, documentation, dependencies and configuration files [12]. It should be possible from scratch to reproduce any given version of the produced software on any environment. In addition to the project's files versioning, it is critical to address a mechanism that enables managing the versions of the external dependencies [64]. Version Control Systems (VCS) offer teams the possibility to keep versions of the project files and people to collaborate without interfering in each other's work [12], as seen on Figure 4. These

systems allow to go back in history's files or to perpetuate a new release version. There are several VCS available: Subversion[65], Git[66], CVS[67], Bazaar[68], Mercurial[69], etc.

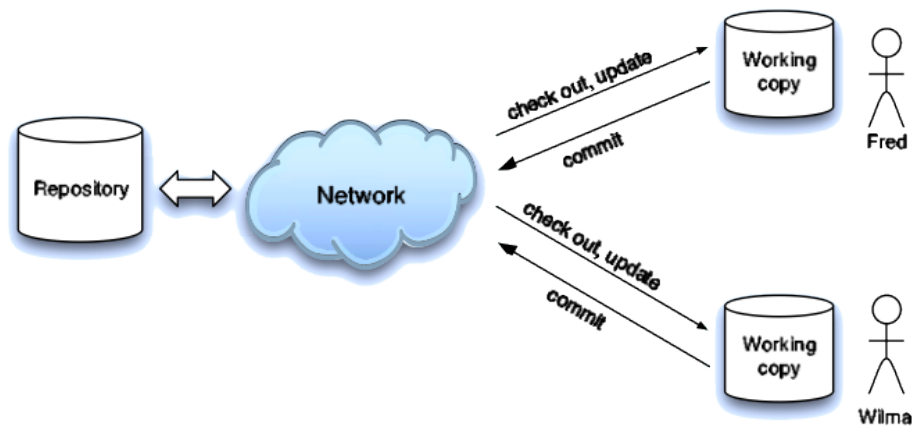


Figure 4 - The repository and working copies [70].

The faculty of gathering information about any change on every file is an essential aspect in managing versions. Such information should endure in the repository history, regardless of every change made on the project files [12]. Such capacity allows obtaining the differences between each change to any point in time, leaving clues for diagnosing new bugs. This process also allows reverting portions of the project changes to previous states, helpful in solving critical problems on production environments.

One of the most useful features in a VCS is the ability to reproduce earlier states of software projects. Typically, those states might be linked to particular versions with specific characteristics, such as the amendment of a set of bugs and implementation of new features [12]. Reproduction of versions is critical to fix issues, which are detected on production environments, therefore it is necessary to increment those versions with just the desired changes. The ability to quickly solve bugs found on production environments is vital to the system perception by end users.

An organization that produces software applications should define a versioning strategy to enable regular evolution of their products in a flexible and appropriate way. If the strategy isn't well planned, it might be unable to update a module or API without the obligation to launch releases of all dependent components. Or, on the other hand, the project may evolve with compatibility with future versions more than required. Such problems are called "dependency hell" that prevents the project to grow in a controlled and safe fashion [71]. This strategy should also establish a semantic, which enables pinpoint the kind of changes that are involved between versions increments. The versioning strategy affects the process of performing a release and the definition of such semantics should cover all cases where there is needed to release a version. There are tools for managing projects' files, beyond VCSs, which may prove vital in that strategy. Maven[72], for example, is a tool that takes a specific versioning model for software projects developed in Java that help

organize the relationship between the various applications' components and their versions, managed through time. Such tools may turn out to be crucial in the configuration management of a software project.

The mechanism for releasing versions shall consist of rigid processes that must not be thwarted because they ensure a greater control on implemented changes, quality parameters and risks on what will be put into production. Whereas that may exist various development versions of the same product, those processes need to consider the effective management of all versions without mutual interference [6]. Features or fixes that will be applied on the final version must be carefully selected and subject to quality parameters, in view on what was planned and the risks that such changes pose to the success of the release.

2.4.2 Software dependencies management

The continuous progress of the project and the different versions or states, which characterizes it, frequently implies changing the way the various modules of the system components communicate. Such modules or APIs may be internal or external and are also coupled with different releases as they are subjected to evolution as most software systems. Generally, internal APIs retain the same version the main project holds but, as external dependencies, internal APIs could live on different development lines from the rest of the components. It then becomes essential to control the versions of these dependencies.

The connection between the project and its dependencies shall be carried out through their compatibility, therefore different project releases may rely on different versions of external libraries without ensuring backward compatibility [64]. The usual practice for managing dependencies versions - typically binary files, if they have been built in a compiled language - is coupling them with the project source code [73]. This ensures its easy reproduction and the control of its versions on different reproductions of the application. While it is an efficient solution, is not the ideal one because developers must manage the dependencies by hand and, on complex projects, it may become impractical. However, there are tools that enable to automatically manage them, such as Maven [72] and Ivy[74] for Java applications. Simply to declaratively specify the direct dependencies and the respective versions or a range of versions, hierarchical dependencies management is performed transparently and automatically [72].

The reproduction of releases also extends to the related dependencies. If dependencies in a VCS follow the evolution together with the remaining files, it's always possible to obtain an early version of the application with the accurate dependencies. With dependencies management tools, obtaining the exact dependencies of a specific version should always be possible in order to be able to reproduce all released versions [6].

2.4.3 Tasks management

A good Issue Tracking System (ITS) is fundamental to report all problems, requirements and tasks assigned to a project. In addition to being a centralized repository for such items, where might be included every aspect of the development process [60], allows individuals to record ideas aimed for improving the application, even if not given immediate attention. Once these issues are recorded, they may be later found, examined, implemented or solved, depending on their priority, in a particular version of the application. Examples of such platforms are: Bugzilla[75], Launchpad[76], Trac[77] or Redmine[78].

ITSs should be a mean of driving between end-users, testers, the support team, programmers and managers. To take full advantage of such platforms, all the crucial information should be recorded and the contribution of incorrect and unnecessary information discouraged [36] to help define a solid and trustworthy support for the project's strategy. Those platforms register every activity of the project under the form of tasks, requirements, bugs, requirement amendments, changes in the documentation, etc. Even if not provided immediate attention, every reported issue will be listed and solved in a specific version of the application sooner or later, without being forgotten [79]. Typically, there are three main groups of items: defects, tasks and requirements. Requirements represent new application functionalities and may be imposed by customers or internally suggested by other project stakeholders. Defects are unforeseen problems or bugs found by testing teams, customers or even by the development team. Certain defects found by the testing team may imply nonconformity of the requirement implementation with its specification [6]. Another major group of issues are tasks that represent other generic activities to be performed by the project's staff. Upgrading the documentation is an example of a task. Usually, each of these items have its own lifecycle representation on ITS and may be toggled through time between different assignees in accordance with their duties and the status of the item [80].

Beyond having potential to organize all tasks of the project's contributors, ITSs provide a real time view of the project status and produce enough information to aid managers to decide on the course of the solution [60]. Such decisions are sustained by the priority given to each issue, but most importantly, help prevent bugs, requirements and tasks to be ignored and help to log all relevant information. The record of all project activity on a unique repository also facilitates access to committed errors and enables preventing them in future situations. If used properly, such systems allow, at all times, that every project member knows exactly what to do and what available time have to accomplish it. Therefore it is of utmost importance that each of the issues is constantly reviewed in terms of priorities, risk and timing in order not to lose control over the solution. Even if tasks' planning does not reach the desired goal, their flexibility enables teams to conclude them on future iterations, in accordance with customers' expectations.

Thus, in view of the completion of tasks, their priority, the risk they entail and which were validated through tests, it is possible to know exactly what bugs, requirements and changes make up each version of the application, including those planned.

2.4.4 Project elements traceability

Best Issues Tracking Systems (ITS) supply mechanisms to record the whole history of the project's activity and allow all changes, since the source code to the versions installed on the various production environments, to be replicated and revisited.

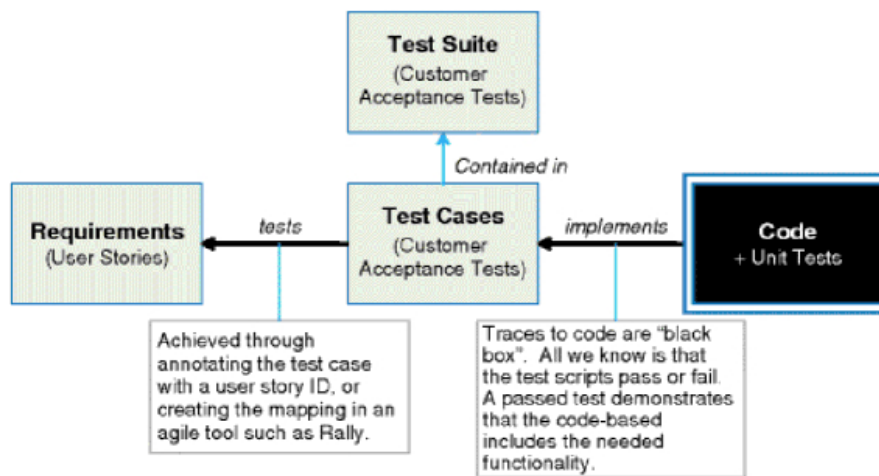


Figure 5 - A traceability information model for a basic agile project [81].

This concept is called traceability and is based on the interconnection between all the elements that constitute the activity that took place for a particular task in the project [6]. The elements that are connected together to form the relationship of all this activity are the requirements, bugs, tasks, specifications, design, architecture, code and test cases, which should be all recorded in ITS. That is, from the ITS, all specification information of a requirement should be available, from design to architecture, the related test cases and all the developed code [8]. Every requirement is linked to a particular version of the application, which may be installed in production environments. So it is possible to trace any change from what is on production to source-code, allowing to help solving any issues that may be found on client machines, or even in test environments.

2.5 Coding improvement

2.5.1 Code reviews and metrics

There are plenty of ways to measure the quality of software construction and the most efficient are code reviews, by using formal inspections, code reading, or walk-throughs (informal reviews). The effectiveness of such practices is so high that senses 60% of defects for the case of formal inspections and code reading detects 3.3 defects per hour versus the 1.8 errors per hour of testing activity [27], despite not being used in a consistent manner on the majority of software projects [82]. Therefore, a better alternative is to use analysis tools that could provide some of the benefits of code reviews, even though not providing the full benefits of human reviews, because the human eye detects problems that a machine cannot [27]. These tools can analyze if the conventions and best practices are being applied in practice and - most importantly - can perform static analysis techniques that look for potential bugs, bad programming practices, bug patterns and poor performance code.

Another major issue in quality metrics is code coverage. Most tools that implement this type of analysis use test coverage per line [38]. This kind of coverage examines whether a particular line has been covered during the execution of a test, and the collected data about tested lines are aggregated to produce a report that the development staff use to check which source code areas are not being covered by testing, for purposes of measurement the construction quality. Varying the coverage percentage of a particular zone of the source-code makes the determination of these measurements [83]. For example, the coverage percentage declining of a certain class means that new developments were added without the execution of respective tests, lowering the chances of finding errors during development phase and increasing the likelihood of being detected in testing environments or even in production environments.

Tools for static analysis are available like: FindBugs[84], PMD[85], Checkstyle[86] or Cobertura[87] for test coverage analysis.

2.5.2 Coding rules and conventions

In order the system to work optimally, sub-processes should be followed with rules and definitions agreed by the parties involved in the project. Such practices can help to measure and compare efforts spent in the different components of the system [9]. The organizational characteristics of such processes make a huge difference when comparing to the individual characteristics of people involved [27], so if the collective ability is not satisfactory, there are no individual capacities that supersede the difficulties of working in a team activity, making those methods for developing software extremely important in the overall success of the project. As with

all others, the low-level processes - such as those used on writing code - should be refined in order to get the most out of the creative process of developing software [27]. For code development, rules and conventions are often chosen by software industry organizations to enable improved communication and cooperation among individuals, making the solution more effective, avoiding the tendency to inefficient behaviors [9].

Different programmers use different programming styles, the problem starts when they need to work on the same code. Often there are no grounds to defend that a style is better than another, what matters is the existence of a well-defined approach by the development teams. That style applies to variable names, classes names, code comments, etc. but the existence of a convention provides programmers more time to think in what more relevant: the logic. However, rules and programming practices must not be implemented blindly or used as a means of assessing the performance of developers but used as feedback about the quality of the written code, allowing the team to learn and progress as a unit [82].

Conventions are a way to manage the complexity a certain activity that prevents developers to wonder about the same issues numerous times and also avoid taking decisions about the same problems in different ways in a completely arbitrary manner [27]. These rules cover different aspects such as formatting the lines of code, code documentation, use of names for variables, methods and classes, use of data structures, input handling, exception handling, particular programming language conventions, etc. After setting these parameters, the source-code will be more consistent, readable and easier to maintain, helping to avoid more functionality defects and increasing performance, security and usability [10]. Coding conventions are a way to protect against recurring problems and to help eliminate the use of hazardous practices. Applied at lower level, tasks become more foreseeable and intelligible to any programmer, improving cooperation [27].

2.6 Continuous integration

2.6.1 Overview

A very common practice seen on many software development teams that produce different architectural modules is to use a long and unforeseeable integration period when the deadline for releasing an application version is getting closer. In the world of software development, this period is called “Integration Hell” [82]. This way of developing software assumes that the application is not at a functional state most of the time, because no one is interested to check that everything is working as planned before the application is finished [12]. In the worse cases, the integration phase is followed by the organization conclusion about the failure of the project [2]. The solution to this problem is continuous integration.

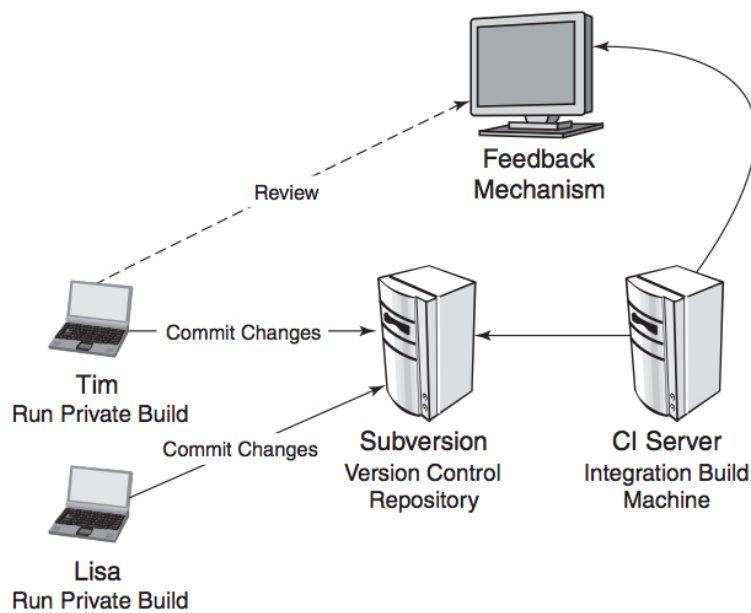


Figure 6 - Components of a continuous integration system [38].

The use of continuous integration (CI) makes the development process smoother, more predictable and less risky, even on advanced stages in the lifecycle of software applications [38]. Additionally, bugs can be traced soon after they are introduced into the project code and, after notification, developers can solve them quickly. Team communication also gets benefited by CI because minimizes the hazard of personal conflict making software development a more harmonious process. Also, reduces the time to release a new version and a demonstrable version of the application is virtually available all the time [12], [38]. Figure 6 shows the basic components of a CI system: the developers' machines, the code repository (VCS), the CI server and the feedback mechanism. Simply put, CI consists on building the project regularly and automatically with the execution of automated tests, quality checks, integration mechanisms and deployment of binaries on runtime machines or sharing repositories. The CI process can be seen as a fundamental factor for the software project overall quality, which is guaranteed by frequent builds, stimulated by every change made by developers [38]. Possible CI server implementations: Hudson[88], Jenkins[89], CruiseControl[90], Luntbuild[91], Apache Continuum[92].

2.6.2 Continuous feedback

The main principle of CI is the motto "integrate early and often". The most frequent the integration process is, the less painful will be for everyone involved [28]. Developers should often commit their code to benefit from the CI process. Expecting more than a day to submit personal changes to the source-code repository may arise integration issues [38]. Errors are easier to detect in an earlier stage, near the point where they have been introduced in the system, since the issue is more present in the mind of the person who introduced it in the system. The detection mechanism of such bugs becomes simpler because the natural step in diagnosing the problem is to check what

was the latest submitted change. That is, problems followed by atomic commits are easiest to correct than to fix several problems at once, after bulk commits were made on code repositories [93].

It is important the development staff to react to all feedback provided by CI servers, therefore there must be an effective mechanism that automatically informs programmers, testers, database administrators and managers about the status of construction. The reaction to feedback entails disciplining collaborators for the possibility to interrupt tasks being carried out and deciding on actions to be taken if the process alerts about an integration problem [38], [94].

Without proper feedback, continuous integration is useless. The feedback is the reason for existence of rapid and continuous builds, which fail as early as possible [38]. For example, if it is merely possible to know several hours later that build tests had failed, is difficult to promptly act on solving the problem before its propagation through code. The feedback aims to generate notifications that encourage reaction in a more accurate and fast way. Such reporting mechanisms include sending SMSs, e-mails, visual alerts in the browser, etc.

2.6.3 The build process

The build process is a series of steps that transforms the various project components in an application ready to be deployed. In other words, a build follows a sequence of statements previously prepared in a specific order resulting in packaged files, prepared to be installed into production or into other types of runtime environments. Typically, build instructions are outlined in one or more description files and its inputs are the source-code, test code, dependencies, the documentation, etc. and the output are the executable files, the user documentation, and libraries on which the application depends.

On small projects, the execution of the integrated build on programmers' Integrated Development Environment (IDE) might be sufficient but, on larger projects, kept by various persons or teams who produce more than one file, demand better control for managing its complexity [12]. Besides, it is essential for large projects that multiple processes and tools such as unit testing and the generation of the deployment files are incorporated into the build process. Software builds automate software generation processes making it less adherent to human errors saving developers from tedious and repetitive tasks [82].

Next, the characteristics of an ideal build will be listed:

- Builds should be portable:

This does not necessarily mean that a build has to be executed in a UNIX environment as much as in a Windows environment [95], but should be easily portable across all machines running

the OS for which the software was designed with least possible configurations, regardless of IDEs used by programmers and IP address of the involved machines. It should be possible to perform checkout of all project's files from code repositories and able to run the build with relatively little effort [96].

- Builds should be replicable:

Should be possible to reproduce, with the aid of code repositories, any build from previous versions of software projects [97]. This feature is essential to generate executable files from previous releases to diagnose issues affecting the costumers' instances, for example.

- Builds should aggregate processes and tools:

Prior the generation of executable files of a given release, it may be necessary, in addition to compile the source code, to obtain the dependencies, to perform different kind of tests, to check the source-code quality, to generate installation files and to deploy them on test environments [82]. Builds should be able to connect all these processes and tools in a single and sequential flow.

- Builds should be flexible:

They must adapt to different environments where they are executed [28]. For instance, a build in a development workstation should behave differently than when it is executed in CI scope. Does not concern developers to perform functional and integration tests every time the build is executed privately, otherwise construction periods could be increased unacceptably. Programmers are just interested in compiling the code, obtaining the dependencies and conducting unit tests. Conversely, on the CI machine, the build should be set for integration tests at suitable intervals following the configuration and deployment of the deliverables on suitable environments.

The most effective way to develop a repeatable and consistent solution for processing builds is using a dedicated tool at the expense of using custom scripts developed by teams [38]. By automating build tasks, the number of manual, repetitive and error-prone processes is reduced [27]. If the tool that allows running repeated builds present all the characteristics given in the above paragraphs, it becomes possible to automate the whole process without human intervention. Continuous integration (CI) is based on automated builds to continuously integrate all the development team's work at a regular basis [12], [38], [82], [95].

There are innumerable build solutions for all programming languages, like CMake or qmake for C and C++, or Rake[98] for Ruby. For Java programming languages the most popular are Apache Ant[99] and Apache Maven. Unlike Ant, Maven is more than a build tool, as stated before: it can automate the all process of building a software application, including version control, external dependencies management, deployment and binaries sharing[96]. All of those features can be easily integrated on CI servers.

2.6.4 Continuous testing

Most software systems consist of various components or modules that communicate with each other. Each of those components is composed by source files, which employ algorithms. Therefore, to ensure that these systems are globally reliable, it is necessary to check at procedure level that each of those components is also reliable. The desired quality for a software application can only be achieved if each of the constituent modules is tested every time a change is added to the project by development teams [9], [12], [38]. Many projects rely only on high-level testing, however, a broader testing strategy should be adopted by performing quality checks at all system levels and involve all individuals, not just the elements of the QA team [100]. Such levels comprise the functional testing, unit testing, integration testing, regression testing, system testing, load and performance testing, etc. So that every change in the application is properly verified, most of these tests can be automated and should be run in the CI pipeline to be carried out repeatedly [38], [101]. Manual testing - the most of functional testing, usability testing, showcases, etc. - should also be part of the testing strategy and, if possible, also included throughout this automated process [36], [82]. The successful implementation of all these tests demonstrates that the system complies with costumers' requirements, meeting their expectations. Build tools can take a crucial role on automating tests. Ant and Maven, for Java projects, are perfect for such tasks. While Ant need fully specified tests declaration, Maven with just a few lines in the declarative model file (configuration file), can automate all available developed unit tests in the project[72], [96].

The most indivisible sort of tests on a CI system is unit tests. They individually test the smallest elements present in a software system, which, in the majority of cases, are procedures. If the development team assume a Test Driven Development (TDD) approach, automation of unit testing shall be a natural process enabling more effectively to sense regression issues during the construction period [6], [36].

In contrast to unit testing, integration tests requires the existence of at least two components being tested together. Such testing may require the existence of a dedicated runtime environment, correctly set up and accessible to the CI server for deployment purposes. The integration tests being carried out continuously helps lessen the risk of incompatibility between APIs during the constant change of code. The period of "integration hell" ceases to exist and the integration process shall be distributed throughout the construction cycle [82].

At the user level fits in all high-level tests and refer to functional testing, usability, acceptance, and others. Independent teams of testers often manually execute these tests, however, some may be configured to be included in the CI pipeline for checking certain relevant use-cases. The automation of such tests increases the CI server load and must only be run during times of little server activity, such as during the night [28], [38].

Finally, the CI server can also perform load and performance tests within different contexts

emulating different environments, continually assessing whether the system have the expected behavior and if any of the latest developments will impact the performance [12]. Continuously testing system load and performance avoids potential problems to be injected into production environments.

3 Industry survey

3.1 Overview

With growing competition from developing countries and the proficiency of the current world powers in the field of software production, it is necessary to realize if Portugal has the potential to develop and maintain high quality products in order to survive in the globalized market. It does not matter merely to create the solution by any means necessary, it is essential that construction processes are well defined and suitable to the multiple production environments to satisfy the needs of stakeholders [102]. Small and medium sized enterprises (SME) play a crucial role on software industry in Portugal, because technological SME are responsible for almost 30% of volume business for technological products in that country [103].

It is often said that software construction is a bumpy and uncontrolled process, but little evidence is available from real world surveys. Are the industry practices as fragile as those often seen in academic projects by students and many research projects? Which are the problems deserving less coverage and need more attention? Can this work use industry inputs to enhance the practices proposition?

To address these questions and to characterize the current practices in the process of constructing software for SMEs of the Portuguese industry, we carried out a survey in which companies were invited to consider a single representative project of their activity. Annex A presents the sent questionnaire to carry out the survey.

The survey has been distributed by email, by telephone and face-to-face between October 15th and November 4th of 2011 to a list of 241 companies around the country, but mostly in areas of Lisbon, Oporto, Aveiro, Coimbra and Braga (see Annex B for acknowledgments). Companies were selected from personal recommendations, web content search engines, industrial associations and other sources. From 60 replies received, resorting to an online questionnaire, 59 were considered valid (24.5%).

The survey was directed preferentially to be answered by project managers, however, there was also participation from executive officers, technical managers and other employees.

3.2 Characterization of the sample

The results presented on Figure 7 and Figure 10 confirms that the objects under observation were the small and medium sized enterprises in the Portuguese software industry. Most of the surveyed companies have fewer than 11 employees and nearly half of them have from 4

to 10 workers (Figure 7), but a significant amount (19%) has between 11 and 25. The amount of software projects in all these software businesses has never been less than one half of the totality of companies' projects but over 35% of them dedicates themselves almost entirely to construct this sort of solutions (Figure 8).

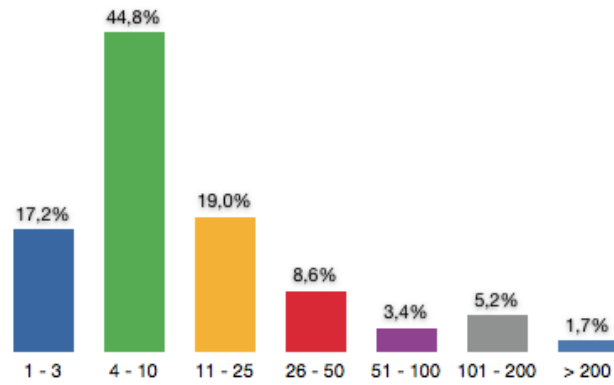


Figure 7 - Number of employees.

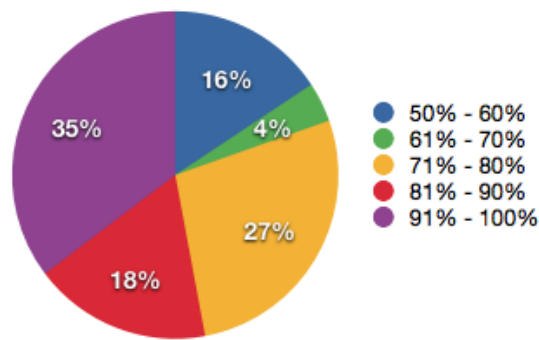


Figure 8 - Weight of the software projects.

Figure 9 indicates that the overwhelming majority of projects (71.2%) have disposed the team in the same physical location and 20.3% of them have nationwide collaborators but merely 8.5% of the projects rely on international participation. Most organizations choose to implement projects with a small number of employees where 44% of the total counts with 4 to 10 persons and 39% have from 1 to 3 people. Despite the reduced number (3.4%), there are also enterprises implementing large-scale projects with more than 50 collaborators (Figure 10).

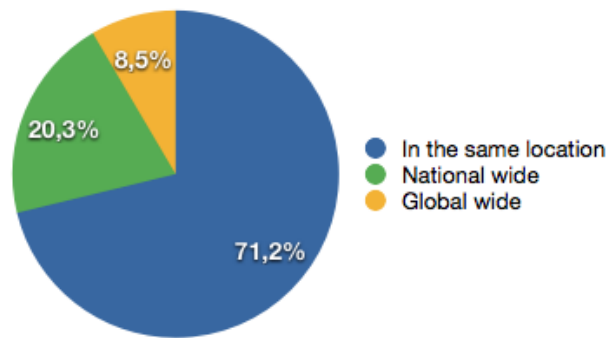


Figure 9 - Project team disposition.

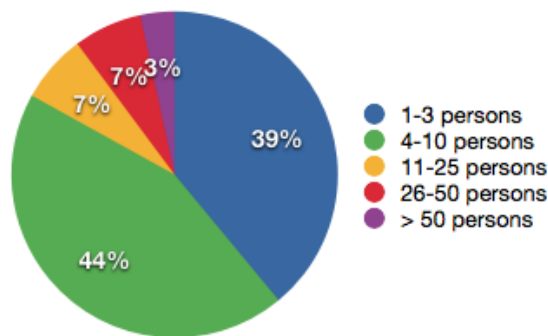


Figure 10 - Project team size.

Once the questionnaire was published (Annex A), typos were found in the question about the project's age and in the question about the time in production. Particular intervals on both questions were intercepting making impracticable to single out on what intervals the projects are inserted, on those particular ambits. Besides that, it's possible to infer with confidence that 17.2% of the projects have less than 6 months old and 24.1% were started, at least, 6 years ago (Figure 11). In the same way, Figure 12 depicts that 13.6% of the inquired solutions are not installed in production machines, 28.8% of them started the production activity for less than 6 months ago and 40.7% are installed on those environments at least for 3 years.

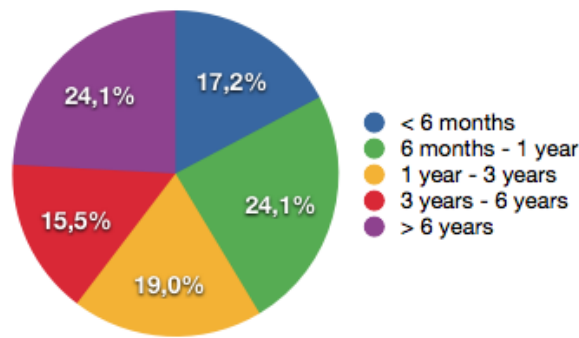


Figure 11 - Project age, since the beginning of the activity.

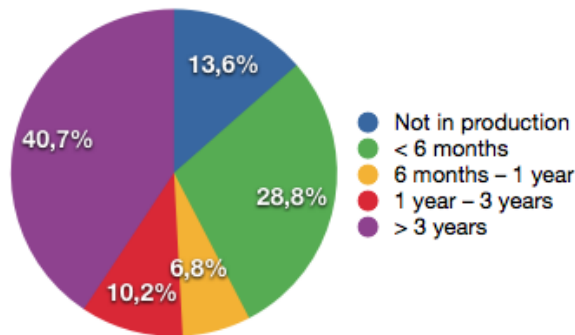


Figure 12 - Time in production of the project.

Figure 13 displays the most widely used programming languages in the Portuguese software solutions, taking into account that one project may support more than one. The JavaScript programming language is used by more than 54% of them and suggests that companies are investing on web solutions, where the use of PHP (32.2%) corroborates this tendency. The C# and Java programming languages, widely used on enterprise products and mobility solutions, are bets from 52.5% and 42.4% of the projects, respectively. As shown in the figure, the Portuguese software market keeps pace with the growing popularity of Apple products via 16.9% of the solutions using Objective-C programming language.

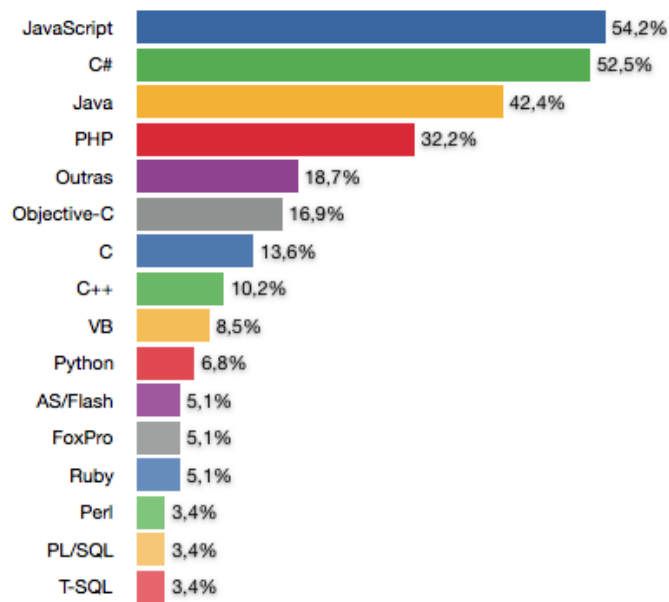


Figure 13 - Programming languages used in the projects.

As the previous paragraph implies on the basis of the used programming languages and the data shown on Figure 14, almost 70% of the analyzed solutions fall into the category of Web projects, which exposes the focus on using this kind of technologies in Portugal, also counting significant number of solutions qualified as Enterprise (37.3%) and Mobile (37.3%). Only 6.8% of the projects are categorized as critical software solutions (Figure 14).

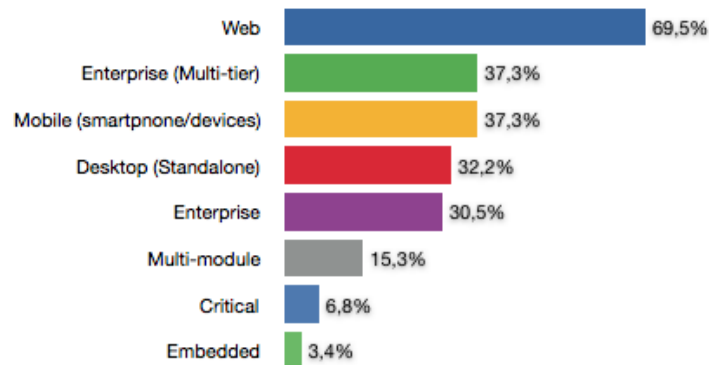


Figure 14 - Project categories.

3.3 Analysis of results

3.3.1 Version control

For certain projects - depending on factors such as the used technologies, the programming languages and the kinds of project – it's not easy to solve the problem of managing dependencies

[104], yet, 91.5% of the studied solutions control their dependencies and respective versions somewhat. Despite not being the ideal resolution, the overwhelming majority of the companies solved this problem with relative ease.

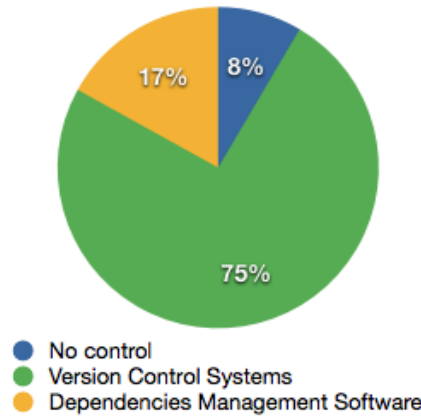


Figure 15 - Dependency control.

As shown in Figure 15, most of the subjects, 74.6% of them, control the dependencies versions (internal or external) by using a Version Control System (VCS). Although this is not the most effective way to handle dependencies versions, nevertheless, 8.5% have no control over its dependencies and only 16.9% of the projects use enabled software to manage them. Of that group, which consists only on 10 projects, Apache Maven for Java is used in 5 cases (Table 4).

	Projects using dep. management software
Total	10 (16,9%)
Maven	5
RubyGems	2
Ivy	1
NuGet	1
WindDev	1
Internal solution	1

Table 4 – Number of projects using dependencies management software.

One of the most significant aspects in the practice of software construction is the capacity to maintain versions because the software evolves, has defects which need to be fixed and companies must have the ability to address the specific problems of each version to production and test environments [6]. Since this is one of the cornerstones of building evolving software, its weight on the projects survival is heavy, even on very small sized ones [13].

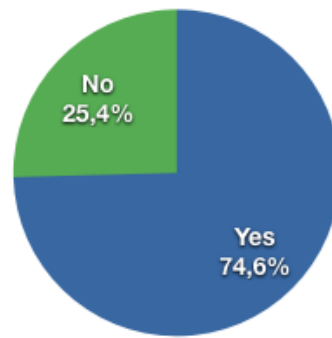


Figure 16 - Capacity to reproduce existent versions.

Among the 59 studied projects, 44 of them (74.6%) are able to reproduce versions, i.e., to restore the state of the project preceding a given moment (Figure 16). If given due importance to this matter, those values should be approaching the totality.

Of those 44, like specified on Table 5, 36 claimed to have the ability to increment existing versions, 40 manage to get information on any change in files and 34 report having capacity to handle changes in the database. Since database change management is very complex to deploy [105], this result is surprisingly positive, even taking into account that may exist projects in the sample not using them.

	Projects that can reproduce existent versions
Total	44 (74,6%)
Can increment versions	36
Can obtain information on changes	40
Can manage changes on database	34
Can fully manage versions	36 (61%)

Table 5 - Projects that can reproduce versions.

Taking into consideration a project with full capacity to manage versions is to have simultaneously all the features stated in the previous paragraph, except for the ability to manage changes in the database, only 61% of projects fulfill this requirement, as shown on the last line of Table 5.

As seen in Figure 17, the use of VCS is performed by 89.8% and the most widely used is Subversion (Figure 18). These findings suggest that a good share of projects are not properly using

software platforms for managing versions because most of them, such as Subversion and Git, offer the features required to perform effective and complete files versioning.

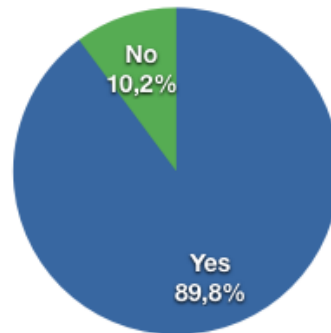


Figure 17 - Use version control servers.

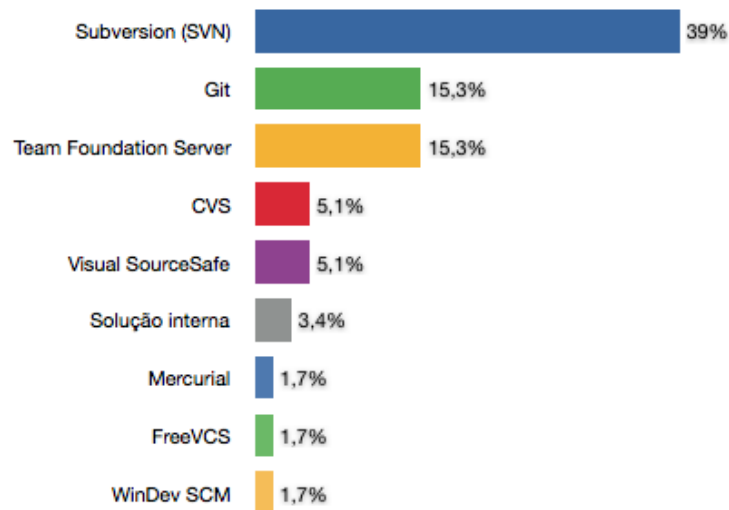


Figure 18 - Used version control software.

3.3.2 Tests

From the whole sample, around 53% of respondents maintains a dedicated team for performing functional tests, as indicated by Figure 19, which means that Portuguese companies stake time and effort in this process of quality assurance. Not taking into account a team of "testers", 89.8% of the solutions are subject to some kind of functional testing and 35.6% use continuous integration to implement them (Figure 20), which confirms the concern of the software industry in this matter.

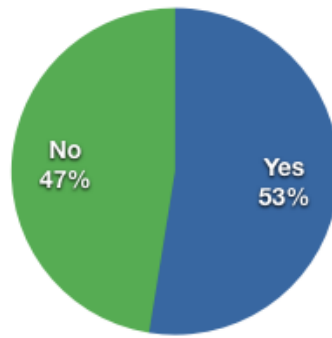


Figure 19 - Use dedicated functional tests team.

Figure 20 illustrates that unit tests are conducted in just 62.7% of cases and 39% claim that use them complemented in continuous integration. The small percentage usage of this practice may be explained by a possible erroneous interpretation, because unit tests are crucial to ensure that a particular functionality is implemented successfully and it is virtually impossible to develop software without programmers performing their own tests. The most likely interpretation so that 37.3% of respondents claim the dismissal of unit testing is the assumption of mandatory use of specific platforms to carry out such practices, like the TestNG and the JUnit for Java applications. Unit tests may exempt the use of such tools, though it is advisable their adoption [35], [37].

Integration and performance/load testing present themselves as being useful in certain types of projects. Integration tests are suitable for large scale systems with multiple modules or instances and performance/load tests are used to realize if a solution meets the expectations in terms of behavioral performance on normal operation or on overload situations. Despite of the specialty of such tests, their use is fairly adopted among the sample, as presented in Figure 20. The less performed type of tests is the regression testing which ensures that existing features are not adversely affected by the implementation of new ones, performed just in 37.3% of the projects and only 20.3% use them in continuous integration.

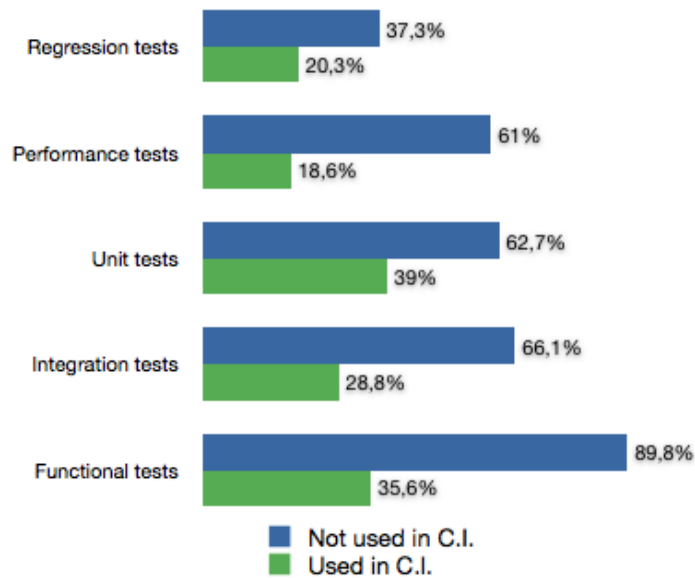


Figure 20 - Performed test types.

Figure 21 represents the crossing of data between the different quality parameters and the use of a dedicated team for functional testing. It is evident that solutions with dedicated testing team are more concerned by using quality practices when constructing software than solutions without such teams. This fact is strengthened on the use of methodologies and development processes (80.6% vs. 53.6%), on the adoption of code reviews (51.6% vs. 17.9%) and on using continuous integration (51.6% vs. 28.6%).

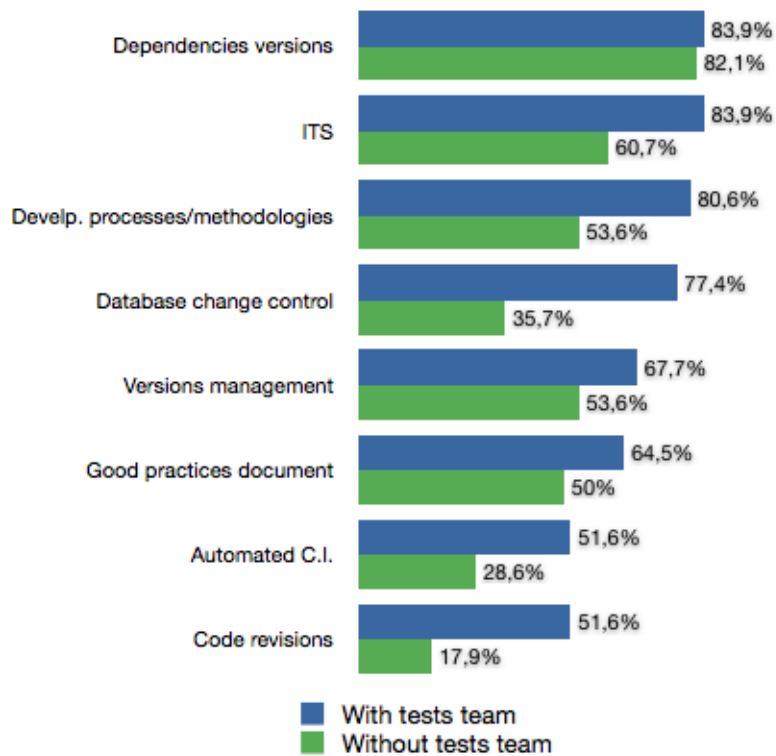


Figure 21 - Construction practices versus tests team.

3.3.3 Continuous integration

The automated continuous integration is used to cumulate portions of code in the global solution to detect problems virtually in the instant they are created, therefore it is always advisable for every project to use this practice [38].

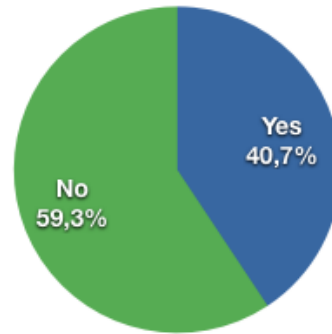


Figure 22 - Uses automated continuous integration.

This technique is used on 24 of the 59 studied projects, that is, 40.7% of the overall sample (Figure 22), and the frequency at which new changes are consolidated on the global solution is performed when possible on 15 out of 24 projects (Figure 23). Of those using automated continuous integration, 23 use it to ensure good compilation of new code, 17 of them use it for testing and 8 to produce quality metrics (Figure 24). While more than 40% of respondents use continuous integration, not all use it to carry out automated testing and to guarantee the quality of the produced code. The adoption of automated continuous integration for deployment into runtime environments is done by just 8 projects.

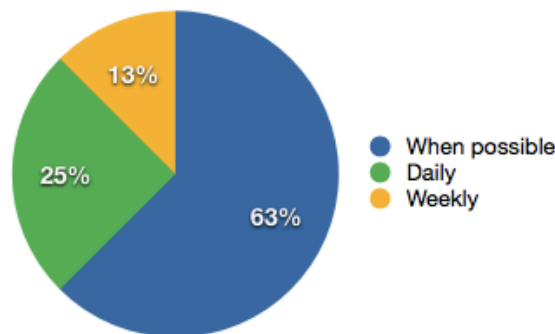


Figure 23 - Automated continuous integration frequency.

From the 35 projects that perform manual integration of the new code, 25 make it whenever possible and 5 execute it in the worst possible moment: at the end of construction phase (Figure 25).



Figure 24 - Phases affected by continuous integration.

Like the comparison of quality parameters among projects with and without functional testing team, we conducted a similar comparison of these values for solutions with and without automatic continuous integration. As Figure 26 displays, the projects using automatic continuous integration reveal higher adoption of quality practices.

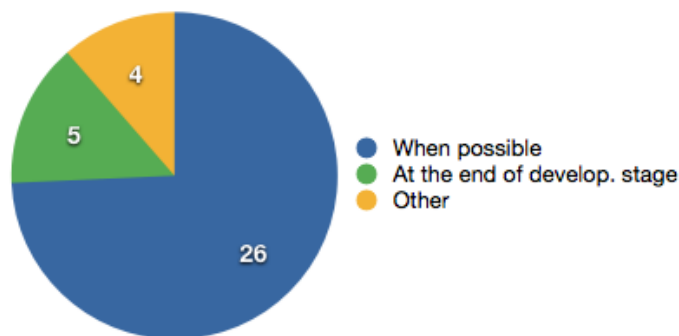


Figure 25 - Manual integration frequency.

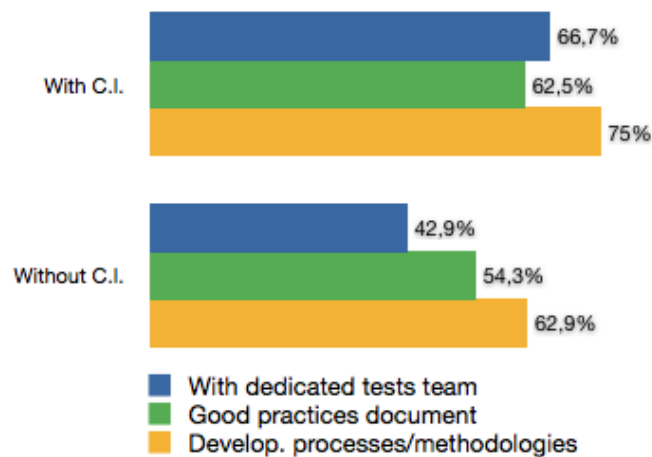


Figure 26 - Quality parameters versus continuous integration.

3.3.4 Quality control

We questioned companies about the use of any good practices document, rules or coding conventions for use by developers and 57.6% of the replies were positive (Figure 27). The following up of this document by those involved in the project helps focus programmatic construction for a

common purpose where everyone involved speak a common language contributing to the evolution of development with improved quality and proliferation of the working experience by all programmers. The fact that almost 60% of respondents claim its use proves that there is some concern in the aspects mentioned above.

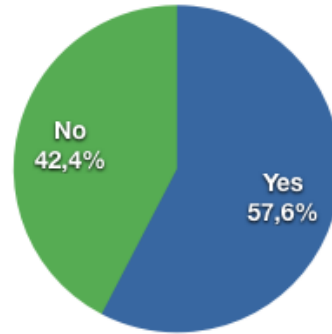


Figure 27 – Use of rules, conventions or good practices document.

Like the best practices document, the use of well proven development processes and methodologies outlined in literature guarantees that the industrial process of building software flows towards ensuring the feasibility and the quality of the solution, maintaining control over it [106]. As the Figure 28 depicts, 67.8% of projects use some sort of development methodology or process. Although being a good percentage, it would be ideal that the majority of companies adopt these processes because software development is a highly complex industry, which can lead to flop with relative ease [107].

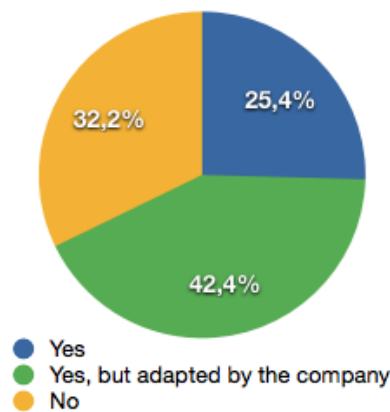


Figure 28 - Development process and methodologies adoption.

Figure 29 shows that, of these methodologies and development processes, Scrum (57.5%) and Extreme Programming (27.5%) are most commonly used by projects. Test Driven Development (7.5%) and the Unified Process (7.5%) showed being slightly proliferated through the sample.

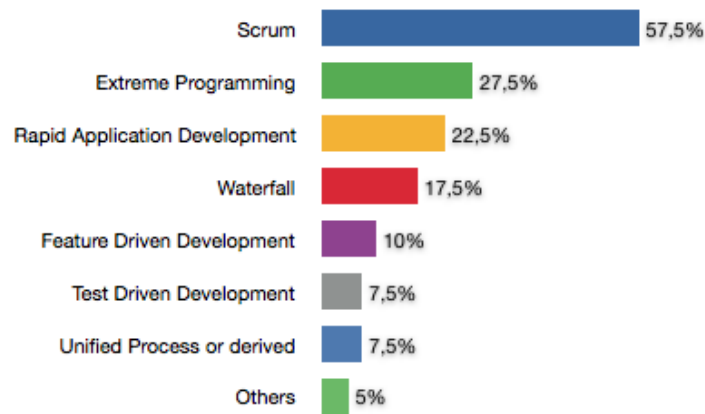


Figure 29 - Most used development processes and methodologies.

From 59 software projects, only 21 use code reviews, whether human (formal or informal) or automatic (Figure 30). Of these, 20 do visual code reviews by employees (other than the programmer) and 6 are subject to static analysis, 4 of whom by the continuous integration server (Figure 31). According to Steve McConnell [27], code reviews are most efficient than functional testing with regard to the detection of bugs, therefore there is still much to evolve in Portugal at the adoption of this practice.

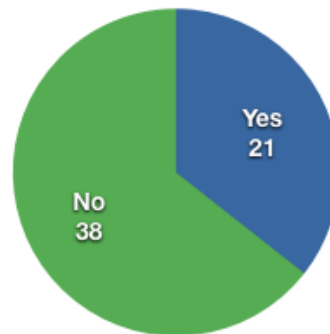


Figure 30 - Regular code revisions adoption.

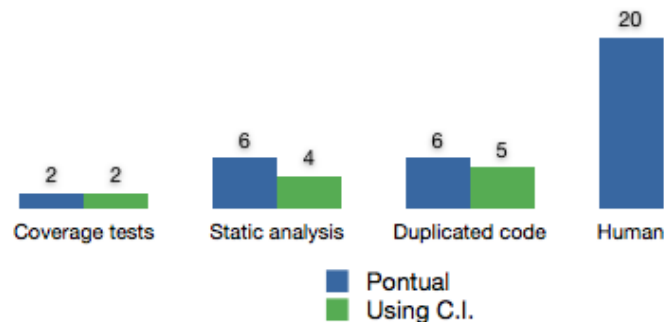


Figure 31 - Code revision types used.

3.3.5 Task management

According to Figure 32, 20.7% of the projects does not use any Issue Tracking System (ITS). This is a large value given that task management is crucial for the proper management of the project itself. Such importance is proportionate to the complexity of the solution and the number of employees involved. The most commonly used application for managing tasks, requirements and defects is the Redmine (19%) followed by the Team Foundation and own solutions (19%).

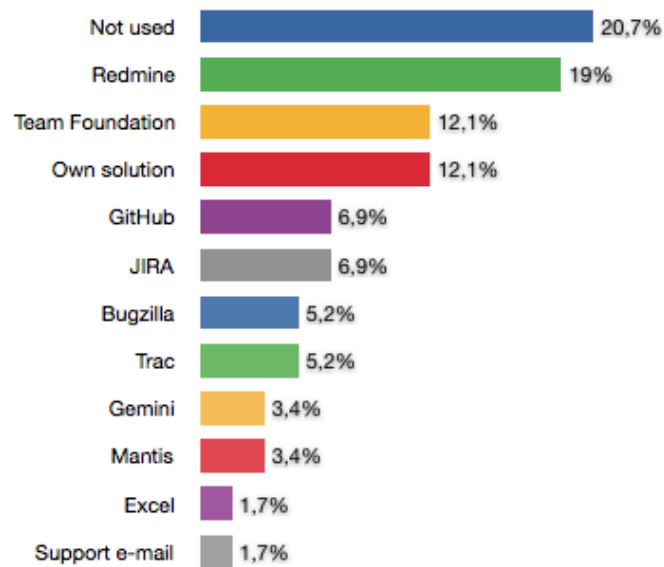


Figure 32 - Used issue tracking systems.

3.4 Conclusion

According to the obtained results, small and medium sized enterprises in the software industry of Portugal show preoccupations on the adoption of quality assurance practices, but there are still margin for improvements on several aspects, especially in some crucial practices such as version control, adopted in full by only 61% projects, and task management (requirements, tasks and bugs) over suitable software platforms (ITS), which are not used on 20% of cases. Also the automated continuous integration is not performed in 35 of 59 solutions, as well as code revisions, carried out in 21 projects. Nevertheless, some kinds of methodology or development process described in literature are used in a large number of the surveyed companies (67.8%) and they assume its relevancy to the project's success. Another quite positive aspect is the execution of functional tests in the vast majority of the sample.

Figure 33 summarizes the opinion of projects representatives about the significance level of the various practices presented on the survey. According to the findings, functional testing is at the top of the concerns in the construction processes of quality assurance. The surveyed companies also

regard the task management and version control as significant but continuous integration is, according to the companies' vision, a process of relative importance, which is consistent with the projects that use this technique.

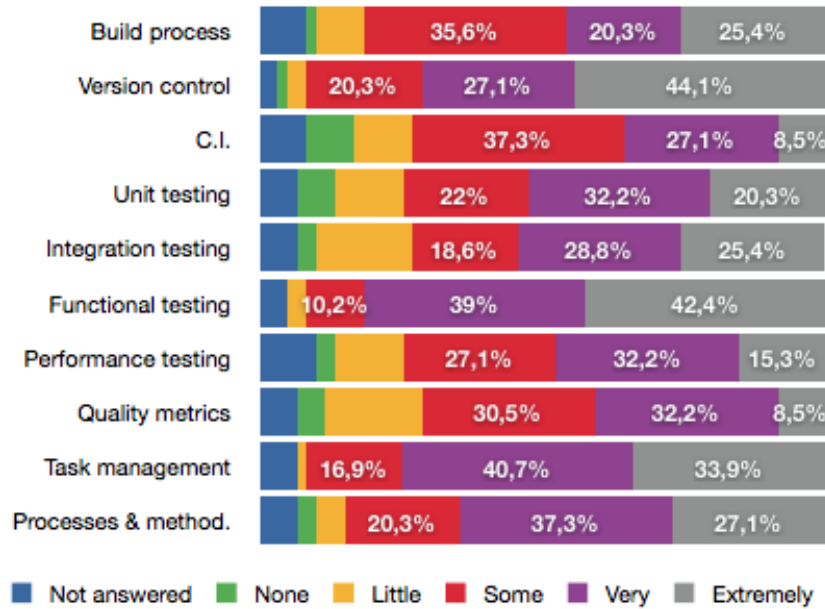


Figure 33 - Impact of the different aspects in software construction.

With the exception of functional testing, all other kinds of tests presented in the questionnaire were given a similar significance, but still higher than the continuous integration and quality metrics. The development practices and methodologies also receive attention among respondents. Following the continuous integration, the software build process is the practice with less importance, according to the sample.

Table 5 and Table 6 aggregate the survey results as a function of time in production and in function of the number of employees. Thus, it is possible to withdraw some information regarding the concern of the companies about the quality of construction of its software according to the evolution over time and the need to have greater control over the development having regard to the amount of participants involved. Although not a clear result, there is an upward trend regarding the use of quality practices in larger projects (Table 6) and in older projects (Table 5).

	Not in production	< 6 months	6 months – 1 year	1 year – 3 years	> 3 years
Nr. of projects	8	17	4	6	24
Version control	2	11	2	4	17
DB changes	2	9	3	2	18
Dep. versions	6	15	3	5	20
CI	1	7	2	3	11
Tests team	3	7	3	1	17
ITS	3	14	3	3	20
Best practices	5	9	2	5	13
Dev. processes	6	12	3	3	16
Code revisions	2	6	3	1	9
Cumulative	41,7%	58,8%	66,7%	50,0%	65,3%

Table 6 - Quality parameters in function of time in production.

	1-3 persons	4-10 persons	11-25 persons	26-50 persons	> 50 persons
Nr. of projects	23	26	4	4	2
Version control	14	13	4	3	2
DB changes	12	14	3	3	2
Dep. versions	19	23	3	3	1
CI	10	7	3	2	2
Tests team	8	14	4	3	2
ITS	15	18	4	4	2
Good practices	14	12	4	3	1
Dev. processes	16	16	4	3	1
Code revisions	8	7	2	3	1
Cumulative	56,0%	53,0%	86,1%	75,0%	77,8%

Table 7 - Quality parameters in function of number of participants.

4 A proposal for construction process improvement in the RTS project

4.1 The previous construction process

4.1.1 Construction processes, practices and tools

RTS is a research and development project, which, unlike other research projects, spawns over several years (active since 2004). Since students and grantees also contribute to the project, means that comprises a highly rotational team, with people frequently entering and leaving the project. This is a big challenge to the software construction practices, which should tolerate the variability in programming styles (and even programming skills).

The old RTS' evolutionary process began with a new release development and finished with its deployment on production machines. New features addition and bug fixes characterized a release implementation. Those bugs were typically found on production machines or during the construction phase which developers immediately fixed after reported. The implementation cycle of a task used to end if the written code passed the conducted development unit tests performed by programmers (Figure 34). Finally, the deployment date of the release's deliverables was agreed with client healthcare institutions where a member of the development staff physically traveled to customer premises to carry out the product installation and perform supervision at earliest moments of the application execution.

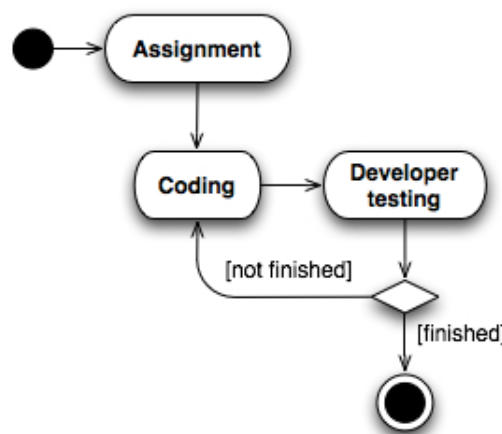


Figure 34 - Task development activity.

Releases maintenance was handled in a similar way. Despite versions generation on RTS project was performed at low pace, sets of defects were grouped together to improve the application's releases, along with new functionalities as a matter of convenience to avoid releasing corrective patches. Copying all files from previous releases to a new directory in the VCS began the internal process of developing a new version and bug corrections changes were directly submitted to the original version's directory on VCS after the release. As seen in Figure 35, each line matches a version (released or not) and each dot subsequent to a new line corresponds a later correction to the original release.

External dependencies management, or matching the dependencies with application's versions, was usually attained on placing the dependencies binary files along with other project files on VCS. This procedure guaranteed consistency between versions of external dependencies and the version of the overall application. Maven, a build tool used by the project, resolved the remained dependencies management not placed on VCS.

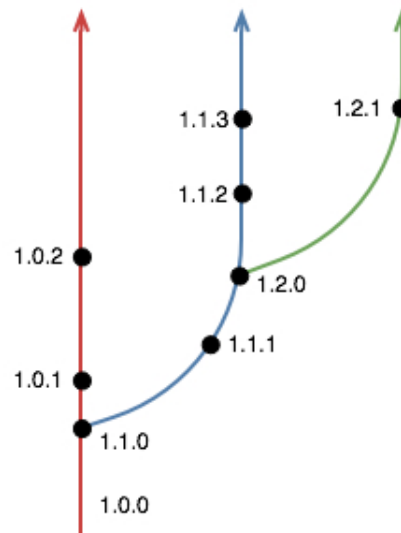


Figure 35 - Old versioning strategy.

The installed Issue Tracking System (ITS) on the project's development environment was used to record some tasks (features or bugs) endorsed to developers. This platform was used only for tasks allocation, however there was no correlation of such issues to specific application versions or any evidence of continuous progress update for high-level management purposes.

The RTS project uses several databases (DB) instances therefore it is crucial to maintain their mutual consistency among the remaining application. Modifications to DBs were only documented in the project's collaborative *wiki* in plain text and its deployment on runtime instances were done manually by a project collaborator during installation of releases on costumers' machines. On development environment, changes used to be installed just after their

implementation in DB schemas. Databases migrations process assumed the individual knowledge about the exact changes to apply on the particular runtime instance.

Code quality control was constrained to static analysis provided by the Integrated Development Environments (IDE) on development workstations, just providing low intensity checks without bug patterns detection. Development was undertaken using private unit tests and integration testing - performed on the shared RTS' development machine – during features or bugs implementation, each conducted by the same individual. New features implementation or bugs fixes ended when developers assume them as complete along with individual tests deemed as successfully implemented.

All activities outlined in previous paragraphs were implemented resourcing the following set of tools:

- **Subversion:** Used for files version control, to identify existing releases and to manage versions (development or released);
- **Redmine (ITS):** Used for development work allocation and to index the entire project's documentation;
- **Maven:** Tool used mainly to build the project's source-code files, to package the compiled binaries and resolve dependencies;
- **Netbeans:** IDE adopted by the whole development team. Used for code construction and other helpful integrations to development process like data sources access.

The implemented practices used in previous construction process are summarized on Table 8. The first column defines the analyzed development practice and the second column describes how the RTS project's development team is using those practices.

Development practice	Existing Implementation
Version control using appropriate software	Yes, with Subversion
It's possible to reproduce previous releases	Only some of them
It's possible to fix or increment previous releases	Only some of them
Database subject to change management	No
External dependencies management	Yes, with Subversion
Tests	Manual development unit tests only
Code analysis	Basic, provided by IDEs
Issues and tasks tracking	Incomplete usage

Table 8 - Diagnosis of dev. practices implementation in previous development process.

4.1.2 Identified problems

Numerous issues were identified on the project's development process given the goal of improving its evolution performance. Despite the low rate of new releases generation, the agility in which they were released was the first found setback. Likely, this frequency arises from lack of facility on launching releases because they depend on the previously specified manual procedure. Besides, the know-how for performing releases was focused on only one developer and, therefore, not all participants were able to easily trigger the releasing process. The installation on customer machines suffered a similar issue by depending on experience and knowledge of a single individual. Besides, that individual needed a long period for checking the success of the operation at customer sites to ensure the installation operation was performed successfully.

Further problems also have been found on maintaining releases. The ability to replicate and fix production problems may be a crucial faculty for the project survival because one can never assume absence of bugs on delivered releases. Restrictions were detected in the ability to apply fixes to final releases of the application: whether different RTS versions are deployed in multiple production environments, the development team should enable to fix them all without needing to update the customer systems to the most evolved version, which is not the case for RTS project. Direct mapping between a directory on VCS and a RTS specific release without distinction between development versions and final versions, hinders the scalability on the number of customers the project is able to maintain without threatening the consistency of all application's releases. Using such strategy, the only chance of ensuring versions consistency is to install the most updated version to fix bugs, even though the purpose of installation might be bug corrections, assuming the most updated version as the most stable, which is not always true.

Another releases consistency related question detected on the RTS' construction process was the powerlessness to handle DBs changes. Code to reflect changes on DB schemas were only held on the project's contributive *wiki* using plain text, without reference to any version. By installing new releases, updating DBs schemas on customer machines was based exclusively on the collaborator's knowledge, which would undertake the job. That individual should know the earlier version used by the particular customer and the specific changes that should be carried out on each DB schema. Consistency of DBs schemas and releases deployment process was based on manual and risky activities, which dangerously threatened the data survival on customer environments.

Managing dependencies was the configuration management area where no serious problems were met, although the outcome of lacking an overall strategy for versions as well affects the project's dependencies. The model that consists in placing external dependencies' binaries along with source-code files in the VCS can guarantee the whole application's version to match accurately the dependencies' versions. Even though, this process might be improved because Maven (used on RTS) provides a highly effective dependencies management mechanism.

Issue Tracking Systems (ITS) help support management and plan strategies for software releases with capabilities to keep pace with every aspect of software evolution. ITS employment proved to be incomplete by the RTS development team, disabling monitoring the development progress with the partial data placed on the platform by developers. Its exploitation was confined to tasks allocation and work division among developers team without any reference to which versions have been implemented, or what their relevant progress, preventing tracking developed work and the target versions. With some effort and due to VCS capabilities, it is possible to trace specific code changes with a particular version because files were placed into the releases directories, which were properly named. If not so, versions were unable to be found on any place.

Validation of developed tasks also presented significant flaws. All work performed by each developer was subject to unit and integration tests solely as a consequence of code construction within the assigned task and conducted by the developer itself. Also, the built code was not subject to relevant quality checks.

4.2 A new construction process

4.2.1 The development pipeline

The proposed construction process and practices assumes all work done by the developers will be checked before being deemed valid within a particular version of the application. This is the base ground that governs the entire argument advocated towards boosting the performance of RTS project's development activities, having as ultimate aim enhancing the quality of product delivered to customers. Yet, just checking everything that has been developed does not ensure achieving such goal, therefore, modifications will be stated which improve each process activity individually by safeguarding the intended outcome, with the addition to be raised in a faster and agile fashion requiring less effort from the development team. The backbone of this proposal is to adopt a continuous integration (CI) system that runs part of those checks and that introduces new habits in development culture of the RTS project, pursuant by what is defined on this proposal. CI will be introduced in the project with recourse to the installation of a CI server on the development environment, which is intended to form part of the programmers' daily habits. The CI server will interoperate with multiple actors from the project construction process.

There are three key players in the studied CI scenario: developers, VCS (code repository) and the CI server, as shown in Figure 36. Developers will be responsible for producing source-code by implementing features or fix bugs and will drive each run of CI build by submitting their work in the files repository (VCS). The VCS will centralize all files that make up the RTS project and it is considered as the primary interface between developers and the CI server. Finally, the CI server's role is to check the correct integration of all submitted work by development team.

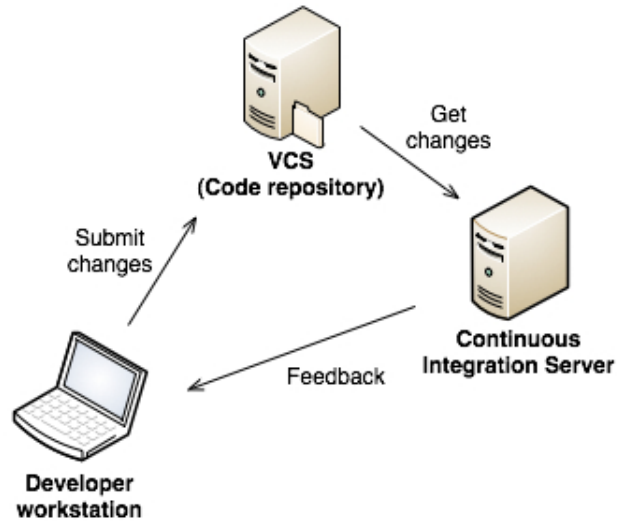


Figure 36 - Continuous integration base.

Next, one CI iteration cycle will be explained referring to the interaction between aforesaid players (Figure 36). The whole cycle starts when a developer submits files changes in VCS. Regularly, the CI server will poll VCS to check for new changes, if any, will confirm if the modifications were properly integrated. Detecting an integration problem drives the CI server to notify the developer that introduced the error to immediately precede the correction. It is assumed that these cycles are performed promptly by each triggered change to the project files.

The suggested CI process will have an automated nature secured by a CI server and relies on the impulsion of the developers' work. One objective of this approach is to seamlessly integrate work from the entire developers team and to ensure that errors are detected immediately upon being introduced into the solution. To ensure a constantly healthy development environment, RTS project's developers must quickly fix the identified problems so as not to hinder the natural project progression. This procedure argues that all programmers must check whether the carried work will not defuse a negative feedback by the CI system. They must perform local unit tests and to insure their changes are not going to subvert work of the remaining team.

Following each change to RTS project, the CI server will internally run an ordered sequence of stages, which jointly ensure the successful integration. Figure 37 presents the stages processed by CI server. This sequence begins with the external project's dependencies resolution by checking their availability. Then the source-code files will be compiled for which unit tests are performed over the Java classes' methods. Following the testing phase, the binary files will be packaged in order to be ready for installation on runtime scenarios. With this assumption, integration tests are met after their installation on an integration machine. Then the server analyses the produced source-code quality and reports will be generated on the output. Lastly, the cycle will end by sharing the generated packages, installing them in a distribution machine for eventual external use, in the event of APIs reuse benefit, for instance.



Figure 37 - Continuous integration cycle.

The integration cycle of this proposal builds on premises that Maven will continue to be the used build tool by RTS project and construction will be based on the Java programming language. Concretely, the outlined above stages are based on the Maven's default lifecycle phases and packages that contains the compiled binary files are Java archive files (jar, war, ear, etc.).

The stages described previously are illustrated in Figure 38, which specifies an activities diagram depicting the flow of operations carried out by CI server when performing integration builds. As already mentioned, the process starts when a developer submits work to VCS that fosters CI server to obtain such changes. Every activity depicted, except the quality checks, will be able to stop the build when files are not properly integrated. If one of such activities notices any nonconformity, the CI server will notify the responsible developer to fix the problem without delay, however, if the sequence flows without issues, the CI system will assume the build was run successfully. The quality checks conduct source-code static analysis, which generate a set of reports to point out possible bugs patterns.

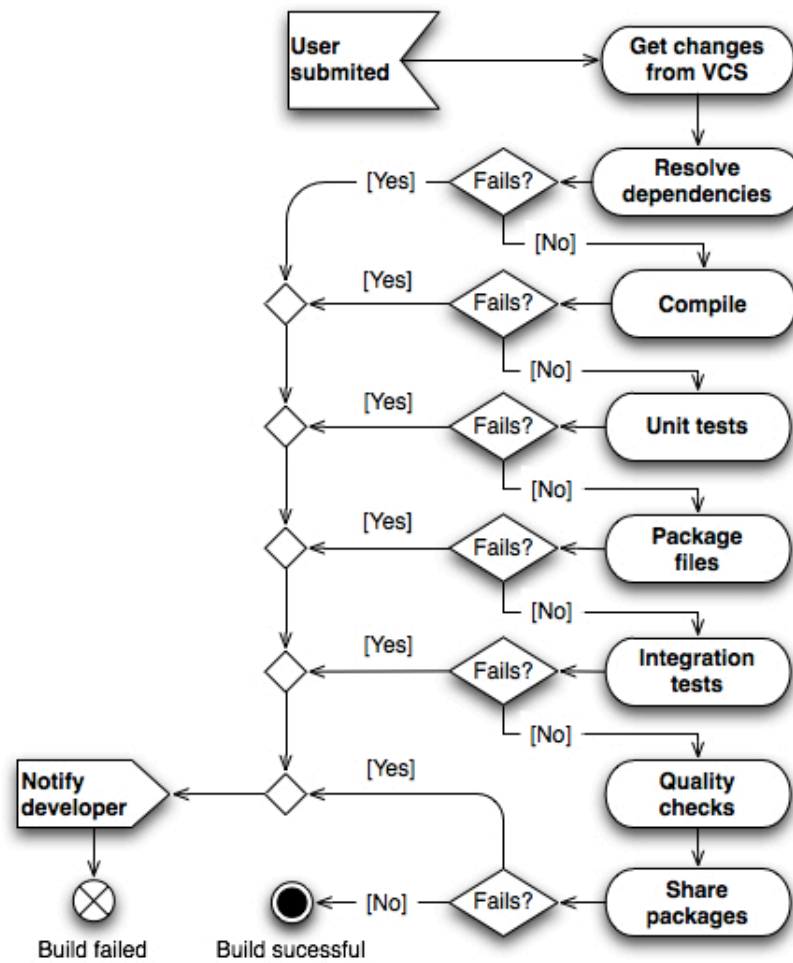


Figure 38 - Integration activities.

Automating CI depends on the development staff's attitude and their awareness of all of its benefits. In order the activity of building software can take proper party of the proposed CI environment, developers must frequently submit their work for it to be continuously integrated. Hence, this proposal defines the period not exceeding one day without individual integration so that developers not end up their journey without submitting their work, not to inhibit proper integration. Developers own the integration initiative, therefore the RTS development team's culture should be changed to use frequent small integrations, reflecting evolutions of an implementation of a particular task, contradicting the usual method of submitting the full set of changes when the task is finished. This culture helps to keep a cleaner and stable developmental line by killing bulk file integration periods.

A critical aspect on CI process is the aforementioned feedback and staff's response ability. Without this faculty, any effort on always keeping the project in a stable state to ensure the ongoing integration of work may become useless. Thus, the CI server used in this work will notify the development team whenever it detects integration issues. Without prompt feedback and rapid reaction, risks of bug contagion may arise in the RTS project.

Is relevant at this point, by defining the foundation for the new developmental process, to refer about the connection between developing RTS application versions and integrating continuous changes. Each integration cycle will be coupled with the implementation of a specific task, typically assigned to developers. Every operation on VCS should match a small breakthrough in the evolution of that task, without implying a VCS transaction to match a full task implementation. A set of tasks, whether these are new functionalities or bug fixes, will define an application version. Its consecutive integration cycles, when successful, will provide progress to versions with the purpose of finishing them. Depending on the project's approach, the completion of all release's tasks or the closeness to the final deadline must entail the stability of the development version. The CI has a crucial role in this mission however the following sections will suggest measures for improving RTS' releases stability and to help reacting when difficulties arise in attempting to achieve this goal.

4.2.2 Releases maintenance and consistency

A fundamental aspect on successful integrate the RTS team's work is to define a main developmental line. This work strongly supports its existence to all application functionalities should be implemented therein and to be instantly integrated by the CI server, even though they might belong to different target releases. The mainline is the core of the application and where the development team will focus all project progress, thus not dispersing new requirements implementation to other parallel development lines, helps strengthen the CI concept. Techniques to enable adoption of such development method will be shown later in this section.

The mainline will bear only new releases development and has the objective to allow a steady progress of the project as a software solution. As seen in Figure 39, the parallel production lines, which correspond to application releases, arise from the mainline, making it a releases factory. The branch lines enable pre-production releases stabilization at first and afterward maintaining production releases. Thus, applying new features from RTS project on branches will not be acceptable as maintaining releases merely involves fixing problems found after definitive launching.

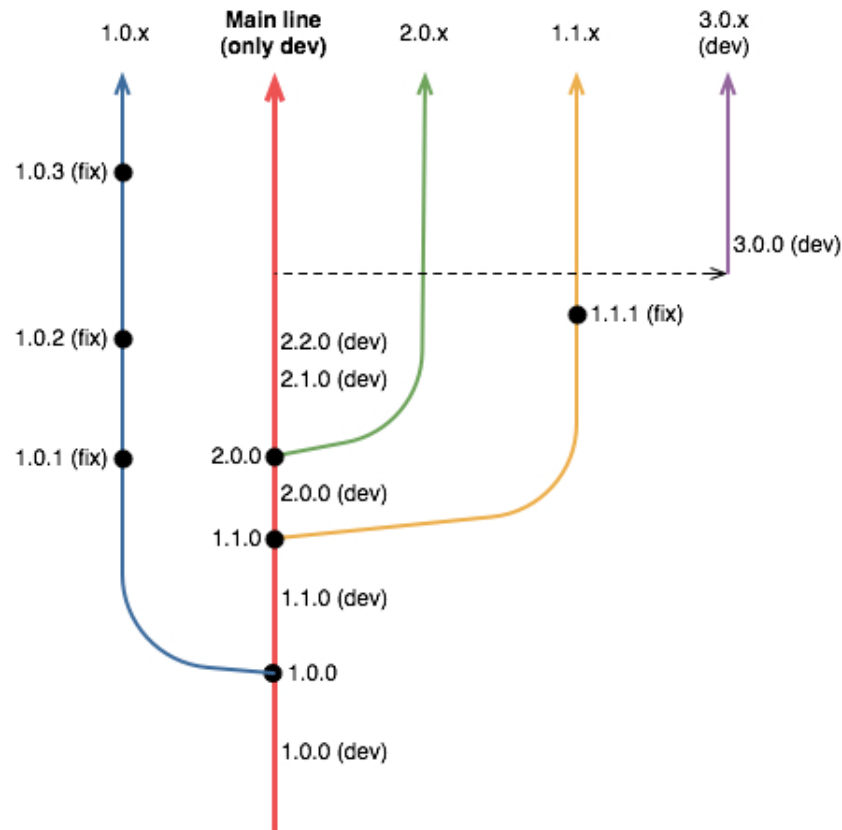


Figure 39 - Proposed versioning strategy.

As noted, branch lines have two functions: a permanent one and a temporary one. Figure 39 shows that one decided at a given moment to create a maintenance branch from the mainline. This proposal specifies the exact timing to create such lines, which will be accounted next. Once the specified deadline for next release is approaching the end, or when all tasks targeted to the release are finished - depending upon the defined plan by the RTS project management - a branch line will be forked from mainline in order to steady the future release, untying it from other unrelated developments being carried out in the mainline. Since then, the version shall be appointed as Release Candidate (RC). To be noted that new features should not be added into the newly created branch because it is meant to stabilize the version prior going into production environments, by remedying any issues found during validation stage - in case not being in agreement with requirements - or bugs detected. Just when all issues and features aimed to the release, on verge of going into production, are properly valid the release can be closed. Stabilizing a RC may be required if the mainline supports developments aimed to other releases, but to simplify process' rules, branches creation will always happen at the referred instant, irrespective of needing stabilization as its existence is mandatory for maintenance purposes after releases are putted into production. Maintaining versions along branch lines requires just adding fixes but not new features, as already explained.

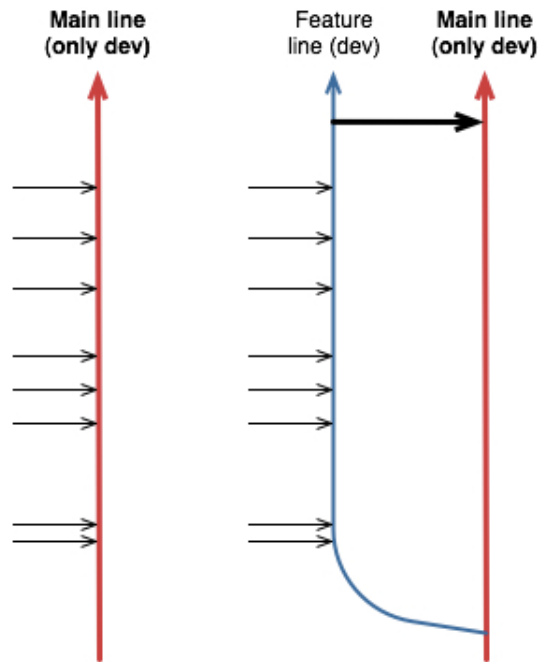


Figure 40 - Using CI on the main dev. line versus "integration hell".

This proposal builds the development process in sustainable integration of work produced by RTS team using a central developmental line, from which will arise all application versions. This technique blends the implementation of features from different releases in the same pipeline, which may difficult the construction process. Will be tempting to detach the implementation of complex tasks from the mainline but using feature branches can lead exactly to the situation that the very CI seeks to prevent: integration problems and "integration hell" period. That is precisely the purpose of CI: to frequently add small amounts of code into the overall solution to avoid adding all changes at once therefore is incoherent to separate development of complex tasks from the mainline when the RTS project will comprises a complete CI environment. Figure 40 shows an example of how it is often advantageous to fit in small changes when developing complex tasks. The mainline from left shows a continuous integration example, while the right mainline shows the use of another line to break up the implementation of complex functionalities that requires integrating large changes at the final period, forcing the "integration hell" period. To ease the acceptance of this practice by the RTS' development team, "feature hiding" should be used. The trick is to add code increments (functional or not) belonging to a complex task, which are not displayed in global solution. This procedure will require some care with architecture, some planning and discipline but benefits far exceed the existence of a devoted phase to integrate functionalities.

The "feature hiding" can also be used when the development team is faced with the development of different releases in the mainline. Figure 39 shows that example in the red line (mainline). Nevertheless, despite the benefits of integrating all features in the same place, eventually be assigned to different versions, there is a particular case where it is suitable to separate development in another line: when a version completely cuts off the compatibility with the

remaining still under development. Figure 39 lists the version "3.0.0" apart from mainline since it was assumed that most APIs were not backward compatible. This example could reflect a profound change in the system architecture.

The submitted strategy for managing RTS project's versions enables consistency across all versions of the application, whatever multiple development versions would still under development or various production releases supported by the team. Using maintenance lines allows insulating the development of new versions from remaining releases, excluding mutual interference, which allow the propagation of instabilities between them. Thus, it is virtually possible to manage endless number of versions, maintaining consistency in all of them. For example, a stable RTS release installed on a customer machine require to be evolved with a fix, its stability would continue to be ensured because the only difference between that version and fixed version would be the correction changes. With no interference between patches and under development releases or with other production versions, isolation between all types of releases is ensured.

4.2.3 Project progress and rapid delivery

To confirm the plan is being fulfilled, however minimalist it may be, versions development progress must continually be assessed during implementation cycles. If there are deviations - unforeseen situations or new circumstances - the development team must react immediately. During the course of development of every application version, the functionalities definition and defects that characterize them can easily vary over time, so the project leader should continually evaluate the progress towards adapting to changes and unforeseen events.

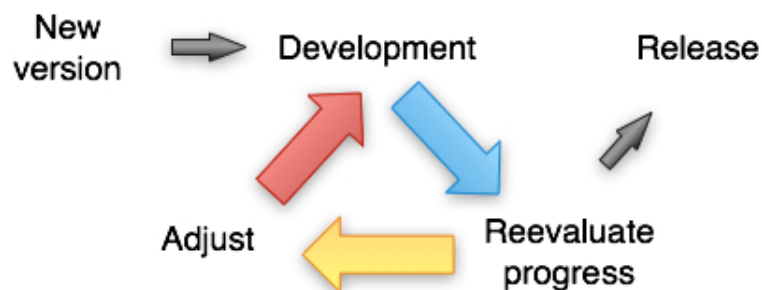


Figure 41 - Continuous version evaluation.

Figure 41 presents a form to continuously adapt the project to new circumstances. Developing a new release begins with tasks definition and its implementation by project's collaborators. Progress should be held with resolving high priority tasks first, whose importance were previously set. Success in this procedure depends upon proper choice of priorities thus the ongoing assessment of this parameter must be made and, if necessary, be changed during the progress. Continuous assessment will monitor the course of RTS project to enable adjusting the

tasks according to needs, as many times as required. The RC will be released after every planned task is resolved.

Every task on ITSs has a related lifecycle, which indicates its state of progress at each instant. The RTS developers shall update the tasks status in the course of changes throughout its implementation, just so is that the proposed process will enable the project manager to evaluate status of versions evolution in real time manner. Strategic decisions can be held then, leading to the rapid reaction over the unexpected changes over project and thereby releases development can flow in the desired direction. This is made possible because ITSs are fitted with capabilities that allow monitoring status of software projects, from the management viewpoint by using diagrams and charts.

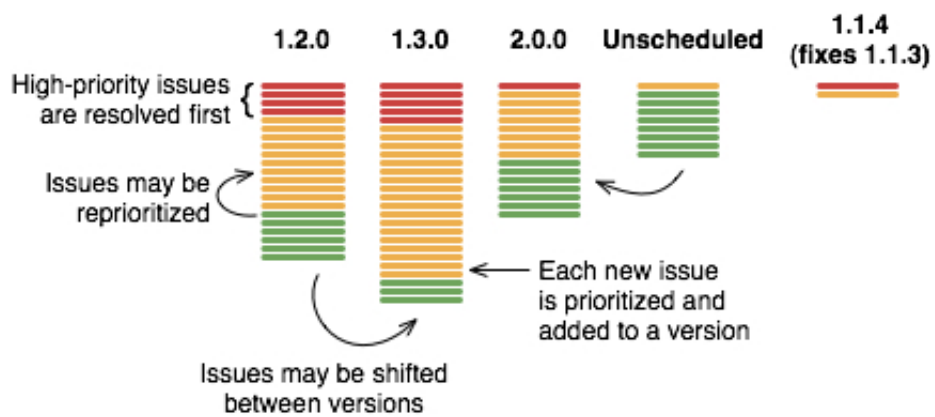


Figure 42 - Issues management.

The present work suggests active targeting of versions to tasks in ITS. Starting from the ground that tasks will have priorities, that will be assigned to somebody from the RTS' development team and have a lifecycle that must be constantly updated, its connection to a particular release is fundamental to successfully construct each release. So, each task (or issue), using project continuous reassessment, will never be forgotten and will always be incorporated in a release. Figure 42 shows an example of the offered method for using on RTS' development environment. The figure shows the example of three development versions (1.2.0, 1.3.0 and 2.0.0) and one maintenance release (1.1.4). Task priorities are depicted by colors red (high priority), yellow (medium priority) and green (low priority). For each version, the highest priority issues should be addressed first, however, priorities can be amended following each assessment. New issues may be included on every version at any moment, always connected to a priority. Every reassessment may establish which tasks can be shifted to later versions or vice versa and issues with no target version (unscheduled) may be caught by any development version - the project leader so decides. Figure 42 also shows a release patch (1.1.4) imposing a correction to 1.1.3 (in production), which itself also already patched the late 1.1.2 version and so forth. For consistency effects, this dissertation

proposes only inclusion of tightly controlled fixes as not to increase risks by removing stability to production releases, so is not advisable to indiscriminately shift issues from development versions to patches and never to include new features.

Following the adoption of suggested practices inducts the building blocks for agile development. Adoption of constant change and continuous evolution in the project endows development team skills that enable to quickly produce working versions of the application, whether for evolution purposes, or to fix production releases or for demonstration purposes. The fast deliveries model suggested in this document relies on continuous integration (CI), accounted in the previous section, and the ability to promptly engender a working version of the application proves the concept of this methodology. Emphasis is given on the value to maintain the RTS' production line always clean, only accomplished by adopting a culture of frequent integration and backed by development mainline where all new features are built. Have as theme conservation an always-working version helps to embrace the culture of fast delivery and continuous integration on the mainline.

As RTS project has an architecture composed by multiple modules somewhat complex, it may be useful for binary files to be available for external use, like APIs reused by other projects as external dependencies. The proposed solution suggests the introduction of a centralized location where CI server shares project's binary files. Figure 43 shows the inclusion of a repository distribution within the project RTS' continuous integration environment. Thus, the integration cycle ends by placing such files on the repository server following each successful CI build.

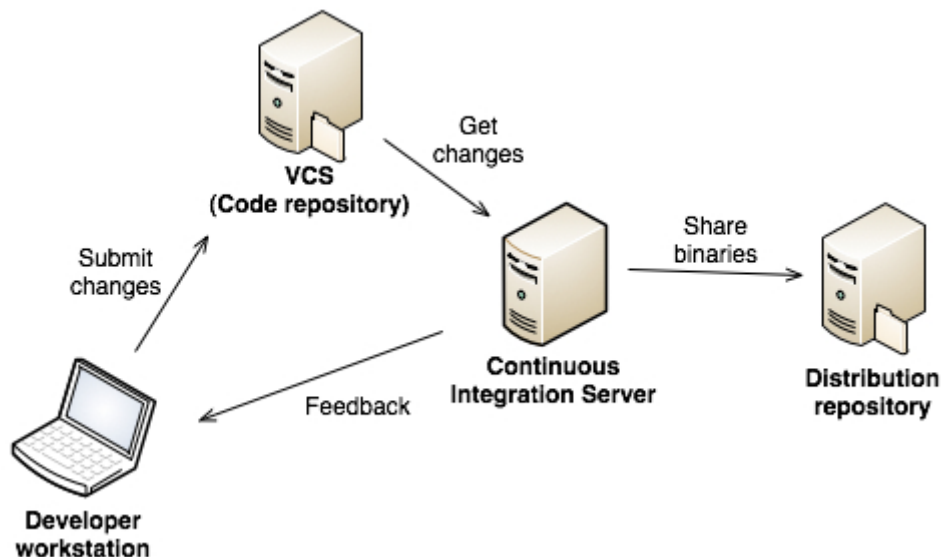


Figure 43 - Using a distribution repository.

This repository will contain every version of all binaries produced by RTS project for that in a short time, all versions of the application can be obtained and so that dependencies are continuously available without the need to conduct VCS checkout and subsequent project

compilation. VCS will allow easily get all RTS versions but thereby obtaining earlier versions will be almost instantaneous, disabling manual labor to yield the desired release. Cataloguing all versions' binaries merely applies to final releases because produced files by developing versions after each CI cycle should be deployed in a limited number on distribution repository, since sharing all-time development binaries can rapidly reach its maximum storage capacity.

4.2.4 Assuring software quality

The overall quality of RTS application is a characteristic that must be cautiously controlled and ensured because the solution manages confidential data in the health sector. Therefore, beyond the safety precautions over the information contained on the database, the overall quality will help keep the application consistent thought runtime while decreasing the number of problems found on costumer environments.

In this proposal, the construction cycle will consist of a set of activities carrying out quality assurance on various development levels. The first step in quality control starts at the developers' workstations. Every developer must ensure development tests are conducted, including unit tests and integration tests run on RTS' development environment. Unit tests can be coded in a proper framework and placed together with the project's business code. Persisting these unit tests on VCS, they can be executed by CI server over all integration cycles. Unit tests automatically run by the CI server ensure the behavior of regions covered by them to stay unchanged over time. That is, each CI cycle will check the eventual regression of behavior in the covered areas.

To improve code quality, developers must complement their IDEs with static analysis tools for detecting bugs patterns and programming errors. Moreover, prior to submitting any modification, each developer must guarantee a local project build execution so that untested code is not introduced in VCS. Figure 44 shows that developers, after task assignment, will start the programming activity and subject work to development tests as often as necessary to ensure its good implementation. Following these activities, developers will submit their work to VCS, which will be integrated into the overall solution. To be noted that the integration should be carried out several times until the task is completed - a best practice defended in the preceding section.

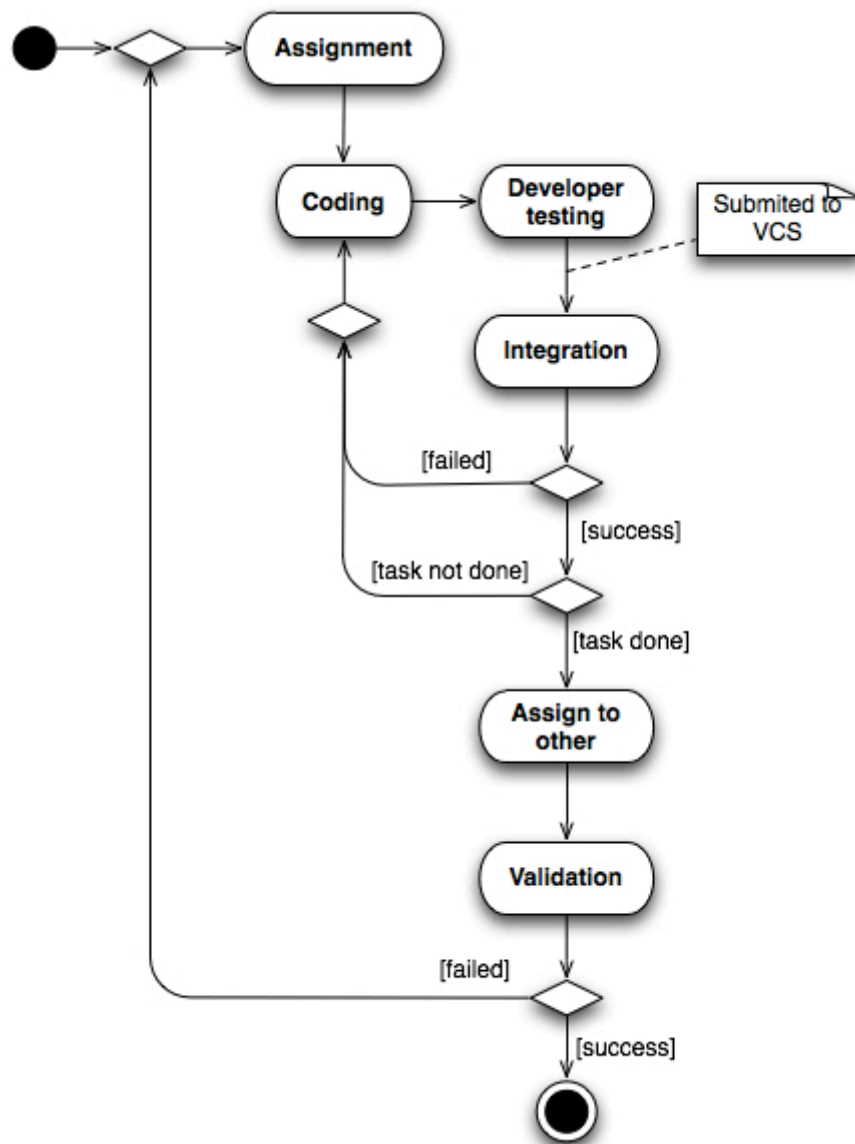


Figure 44 - New task development activity.

As mentioned on previous paragraph, the CI server will conduct a set of tests on everything that will be developed in the RTS project. As seen in Figure 44, this process is taken after every VCS commit, which will subject the code to automated unit tests. It is also achievable to adapt the process by adding functional, load and performance tests but this proposal provides greater relevance to the establishment of an integration scenario devoted to automated tests. The CI server will prepare the environment and then execute a series of tests for integration purposes. Figure 45 shows an example of how CI server will perform this sequence from the deployment of application to execution of automated integration tests. Following producing the installation packages, the first step is to prepare the environment: the server will first migrate the database with the necessary changes, consistent with the development release, and then deploy binary files in the RTS' Application Server. After deployment, the CI server will accomplish integration testing on the

execution environment, and if one fails, the developer that introduced the problem will be immediately notified. The sample of Figure 45 presents a series of tests where the latter fails, stopping the integration cycle and disrupting the build. This procedure is wrapped in "Integration" activity of Figure 44 and will compel the responsible developer for prompt rectification.

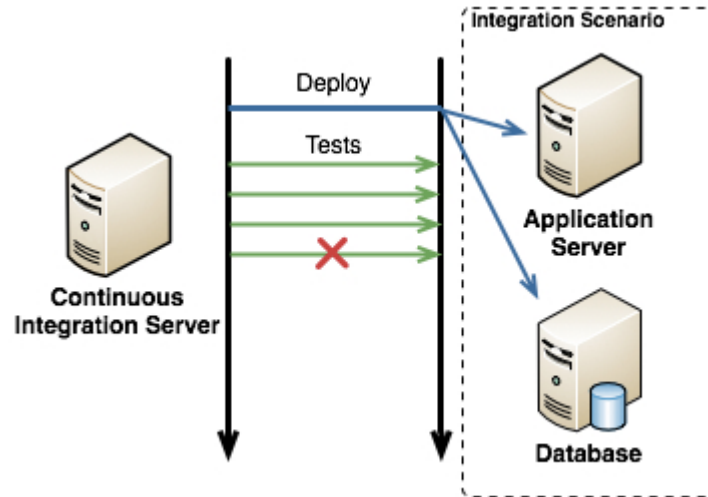


Figure 45 - Automatic integration tests.

The RTS' CI server will also conduct automated checks to source-code quality which include detection of bugs patterns, proper use of coding conventions defined for the project, excessive complex code detection, unused code detection, bad performance code detection and duplicate code detection – a.k.a. "copy / paste" detection. The CI server will be responsible for regularly analyze the whole project source-code but these scans will not raise notifications, or demand the prompt correction like with unit tests or integration tests. However, reports will be generated, which will enable to identify each detected issue.

To overcome the lack of a testing team for checking proper implementation of tasks on RTS project, development team will perform validations on what is implemented. Validating tasks maps out on "Validation" activity of the diagram in Figure 44 and occurs after the developer deem the implementation of a particular task as completed. In order to ensure whatever will be added in a project specific release agrees with the expectations of customers, developers will conduct validation tasks, using functional tests, with the constraint of not being themselves to validate their own work. This provides some exemption about what will be implemented in the solution and includes more than one person in the tasks lifecycle. In terms of ITS, when developers complete implementation tasks, they should change its status to resolved (not validated) and assign the task to another individual to validate it. The validation activity, carried out by a 3rd person, will conclude whether the task has been completed successfully or not. If validation fails, the individual in charge of validating the task will have to document the found problems in ITS and ultimately

return it to the implementer again to fix it. The validation ends only when the task is implemented effectively, ensuring it is carried out according to specification.

4.2.5 Team communication and interaction

In order to improve communication between individuals on RTS' development team, this proposal presents some measures to assist the dissemination of information. As deemed above, the RTS project has a rotational team of developers who does not carry full-time job, disabling smooth communication among individuals and hindering collaborative work. For this reason, placing all necessary information in an easily accessible location could be a possible solution. Solutions discussed later will not disentangle such communication issues but will help improving communication and reduce learning curve of new employees.

The use of best practices is applicable to all project activities. They never overlap the expertise that an individual has in a particular area, however, the existence of one or more documents, which record some of the experience gained over time, helps disseminating all this knowledge. These documents may also be based on the experience of other professionals oblivious to the project, often found in books, academic papers and even on the Internet, but the good practice learned in the course of the project should be registered so that everyone can access this information making more efficient the execution of the activities they address. It is important to note that best practices are relevant to the project and they are developed out by the end of the project itself with the learning and experience of everyone involved.

Good practices can benefit all activities of the developmental process. There may be good practices in creating tasks on the Issue Tracking System (ITS), on using the Version Control System (VCS), on unit testing activity, and so on. The whole experience acquired and the learning arisen by the committed errors throughout the life of the project should be proliferated by the team.

The RTS project, before this work, previously used Redmine for documentation, a quite popular open-source ITS. This proposal recommends its continued use to help managing the project and related documentation. This tool offers a contributive *wiki* where every project participants will be able to freely publish documents and other topics of relevance like: RTS application user manuals, requirements specification and implementation documents, record the learned lessons throughout project evolution; manuals for used development tools; solution to recurring, among other subjects of interest.

Towards standardizing certain aspects of project enhancing communication among RTS' collaborators, a best practices and conventions document will be written for this work. This document will contain code conventions definitions to be used by developers, development best practices and the documentation of the whole development process suggested by this proposal. That handbook will be made accessible in a digital format in the contributive *wiki* for it to be

evolved and thus collect gained experience by the development team during development of RTS project.

4.2.6 Summary of the new development process

The following table (Table 9) summarizes the importance of the suggested practices on the resolution of the found problems in the previous development process.

Caption: +++: Solved the problem; ++: Contributed greatly to solve the problem; +: Contributed some how to solve the problem.

Proposed actions	Problems								
	Lack of agility on releasing versions	Difficult to reproduce and fix previous releases	No scalability or consistency on releases	No database change management	Inflexible dependencies management	No version control on ITS	No validation of developed work	Lack of quality checks	Lack of communication
Continuous integration culture	++						++	++	+
Automated tests							++	++	
Automated code static analysis							+	++	
Using Maven as dep. management system	+	+	+		+++				
Database migrations	+	++	++	+++					
Using a main line and releases branches	+++	+++	+++	+					
Tasks and issues as subsets of releases on ITS	+	++	+	+		+++			+
Continuous release assessment	+					++			
Using Release Candidates	++		+				+		
3rd person validation process						+	++	+	+
Best practices and conventions document									+

Table 9 – Problematic practices and suggested actions.

4.3 Applying the construction process to other projects

The defined processes and practices are within the scope of improving the development practices of the RTS project. Though, this work may apply to other software projects. All presented definitions, such as fast delivery, agile development, consistency of versions, continuous integration, etc. are generally applied concepts extensible to projects with similar characteristics.

The general projects characteristics for adapting this solution are: distributed applications; teams with several collaborators, projects whose quality must be ensured; projects with constant evolution and ongoing maintenance, projects having multiple costumers.

The sole difficulty to adapt the process to other projects is the potential incompatibility of presented technologies to be used in the development environment. The RTS project adopted four core technologies that speeds up the process implementation: Hudson CI server, Java programming language, Maven build tool and Subversion. Those technologies interconnect smoothly among them, enabling to assemble the pipeline without development of own solutions or major adjustments. Next, it will be shown how such technologies enable the process clean implementation.

- Hudson:

This CI server supports Subversion and Maven natively, and consequently Java programming language;

- Java:

Tools that enable enforcing some outlined features of the process are available to this widely used programming language. In particular static analysis tools, unit testing frameworks and others;

- Maven:

This tool provides more than the ability to perform project builds. In addition to automate activities like testing and package generation, it allows teams to easily manage external dependencies and to share project's binaries. It can be bundled with a set of *plugins* that extend the project with automatic features like documentation generation and application deployment on runtime environments;

- Subversion:

Like most VCSs allows controlling project's files versions, integrating smoothly with Maven, Hudson and Redmine. Helps keeping the consistency of file versions over time.

Projects that cannot make use of such technologies must encounter other possible solutions. Maven is only for Java but other programming languages have efficient tools to build projects. For instance, C# applications can use NAnt or MSBuild. To control file versions, Subversion is commonly used on many technologies, so it can be used in most software projects but there are excellent alternatives: Git, Mercurial, Bazaar, etc.. Even several CI servers are suitable to use on many project types: Hudson - used on RTS - can be used on C++ projects with CMake (build tool) and CppUnit (framework for unit testing), which can be adapted by writing a few scripts.

In short, the proposed process is customizable to most software projects with the eventual need of selecting other technologies than the ones used on RTS. They might be quite different, alternatively organizations can produce own solutions to fit their needs or adapt existent tools with custom scripts.

5 Implementation of the proposed construction process

5.1 Selected tools for the construction process

Some of the elected tools for the process have already been mentioned in the preceding chapter, others will be referred and their function will be detailed along the sections of this chapter. The remaining, such as testing frameworks, shall be revealed but are not included in this section as elected because it is intended that development staff has freedom to choose such tools. Taking the cue of testing frameworks, there is a wide range of options and many of them with different capabilities, therefore it is valid in particular situations to use multiple testing frameworks in the RTS project. The following presented tools are the sustainment of the proposed development process and they cannot be easily superseded, at least without a formal decision by the RTS project team. The core tools are: the continuous integration server, Maven, the Maven repository manager, the database migration tool and the Issue Tracking System (ITS).

- Continuous integration (CI) server:

Choosing a continuous integration (CI) tool relied on multiple requirements but the main ones were: free to use (preferably open-source) and straightforward support for Maven 2. Considering these factors, the tens of tools available, just five comply with the requirements: Apache Continuum, CruiseControl, Jenkins / Hudson and LintBuild. Upon elimination of most of those tools and in view the popularity, flexibility and constant evolution of the product, it was found the server that fulfils all needs of the RTS project: Hudson.

Hudson is a server that runs on J2EE containers and its installation on the RTS' development environment was swift for the reason that Tomcat is the J2EE web container of election for the project. Jenkins further emerged as a fork of the Hudson project, after it had become the property of Oracle, but on the RTS project was decided to give continuity to the first choice. There were no reasonable grounds to outcome the change, as the essence of this tool remains the same. Hudson is deemed the obvious choice for Java projects. The CI environment setup is easily carried out when dealing with Maven, which is done in just a few minutes, and provides native support for Subversion, both used in the project RTS. Its rising popularity led to a community who actively develops *plugins* that comprise with many other tools like JUnit, Redmine and other build and Software Quality Assurance (SQA) tools. Due to its extensibility, Hudson is incredibly flexible and adaptable to every software designs developed in Java. The ease setting makes it possible to assemble a CI environment within minutes but simultaneously has the ability

to support complex integration scenarios with multiple parallel Hudson instances interacting, if necessary.

- Build tool:

Before this proposal, Maven was already being explored on RTS project but as it has a key role in the submitted development process, it was not deemed moving to an alternative. Nevertheless, a comparison with Ant - another popular build tool for Java projects - will be made.

Maven is a build tool because it allows generating deployable files from Java source code. However, it is more than that: a management tool that includes a project object model (POM), a set of standards, a lifecycle, a dependencies management system and logic to execute *plugins* on every phase of its life cycle. For Java projects, the most commonly used alternative is Ant, a tool that focuses on code compilation, packaging, testing and distribution. There are several advantages of Maven over Ant beyond those referred: Maven uses conventions, unlike Ant, which forces the exact definition of source files location and where to place the output; Maven is declarative as it enables to build a project without being necessary to specify what should be done and when, unlike Ant, which needs to be told on how to compile, copy, compress, etc.; Maven has a lifecycle that performs a fixed sequence of steps that allow to build the application.

Besides being a build tool, Maven also makes the management of internal and external dependencies, directly described on text files with specific XML notation on each of the constituent modules. The dependencies are declared in Project Object Model (POM) files and not placed in a specific directory like on other build tools like Ant. Maven will do its best to find those dependencies using special remote repositories, typically available to the general public. Once the build is executed and the dependencies found, Maven will store them locally in a private repository of the machine that is running the build. It will only download from the public repository when the dependencies are not found in the local repository. In addition to the declared dependencies, Maven will use transitive dependencies, i.e. dependencies that are undeclared but necessary to the declared ones.

- Maven Repository Manager:

There are some freely available Maven Repository Managers for the software development community. The most popular are Artifactory[108], Nexus[109] and Apache Archiva[110]. Any of the referred Maven Repository Managers as well having the ability to serve as a proxy for external dependencies used by the organization, saving bandwidth on the internal network, they have the capacity to mirror public repositories, making the task of configuring the repository section inside POMs easier. But the most significant requirement that any Maven Repository Manager should implement is the possibility to easily distribute the organization's compiled products (artifacts). This capability streamlines the process of sharing work between different development teams within the organization, or even among members of the same team.

On RTS project it was decided to use Nexus due to its simplicity and incredible small footprint of only 28MB of RAM, against the 128MB of RAM used by other solutions. The RTS' Nexus instance is running on a shared machine where multiple other servers are configured, including the SVN server and Hudson (C.I. Server), thus the less impact on the global architecture, the better. Its installation is also an advantage because it requires no container configuration on the J2EE Application Server where the Nexus WAR file will be deployed. Despite Nexus server not having all the advanced features used by Artifactory, like parallel download of dependencies, or statistical tools for analyzing repositories and artifacts, it fits perfectly the purposes of RTS project. But probably the greatest benefit of Nexus on the remaining solutions is its extreme efficiency derived from its development team's know-how at Sonatype enterprise. The staff is composed by the founder of Maven and by several core Maven developers. In addition, this team maintains the Maven Central Repository, making Nexus one of the more robust, secure and light Maven Repository Managers available.

- Performing Maven releases:

Maven is extensible with the use of *plugins*. As previously stated, Maven, as well as being a build tool, it can perform software configuration management actions over a project developed in Java. The RTS project will adopt the default releases management mechanism provided by Maven: release versions and snapshot versions. Hence, conducting RTS releases, as well as entailing changes on Subversion and on Redmine, also implies changing versions on the Maven items, particularly on POM files. While Maven features benefit the RTS project, while releasing versions, it will imply additional set of actions, such as: changing POM versions, commit POM changes into Subversion, perform binaries deploy to the sharing repository (Nexus), among others. To streamline and semi-automate the process, a *plugin* for Maven will be used: Maven Release Plugin[111], which will attend RTS releases. Its use in the context of project will be detailed later, but its operation will rely on three actions: releases preparation; releases execution; and creation of branches for maintaining and stabilize releases. Each of these actions will map out into a specific *plugin* goal: *prepare*, *perform* and *branch*, respectively.

- Database migrations:

From the available tools to perform database migrations, such as DbDeploy[112], migrate4j[113] and autopatch[114], Liquibase[115] was the favored to conduct that kind of operations in the project RTS.

A core feature of Liquibase is to enable DBs migrations using script files containing plain old SQL code, exempting the need to use *changesets* files with specific implementation formats. Such files enable easier migrations in a portable fashion since the SQL code is directly executable over PostgreSQL instances, used on runtime RTS systems. Despite these systems employ a driver that implements the JDBC API to carry out database operations, Liquibase helps manage DB

evolutions with Hibernate - an implementation of JPA (Java Persistence API) for Java environments. Liquibase binds along with Maven builds due to a *plugin* that allows automated migration on integration and testing environments. In production scenarios, migrations can be accomplished by running a command line application developed in Java that takes multiple parameters. Development teams can also use the Liquibase API for Java that allows developers to programmatically access all features from this migration tool. As well as allowing migration of multiple databases towards its evolution, Liquibase also allows *rollbacking* them to older versions. This tool supports to rollback certain database operations automatically but using custom SQL code to match each evolution with a rollback operation is supported inside *changeset* files.

- Issue Tracking System (ITS):

On RTS is intended to use just freely distributed development tools, preferably open-source ones, and ITSs are no exception. From the several choices, such as Bugzilla, Launchpad or Trac, the project had already adopted Redmine, before the beginning of the present dissertation. Redmine is a fairly popular web-based tool, freely distributed with an open source license, developed in Ruby and supports different databases. Allows managing multiple projects and has good third-party integration with tools like Subversion and Hudson, also used on the project. In addition, there are a large number of *plugins* that extend its capabilities, held by a wide community of developers. This solution provides all necessary features for proper project evolution management, so was decided to continue using it as RTS' official ITS.

In order the RTS project to be managed successfully, it is required that the selected ITS fill out some prerequisites complied by Redmine. This tool is a highly flexible ITS which helps manage users with different roles across multiple projects. It does time tracking with a calendar and an integrated Gantt chart. Besides having default configurations on different kinds of issues, Redmine supports creating custom ones by defining its full lifecycle and the selection of specific fields. Integrates well with VCSs, like Subversion by providing diff views, repository browser, *changesets* viewing and connection between SVN revisions and issues for better traceability. Also implements a documentation manager, news or other files, a discussion forum and a participative *wiki*. It also supports e-mail notifications and RSS feeds.

Redmine offers all conditions to development teams to manage tasks, plans, and releases of applications, but motivation and commitment of workers are required to carry out all actions necessary for its proper use.

5.2 The construction environment pipeline

There are many different ways of setting up an integration environment depending on what is required to perform the integration process, but there are actors who are essential and common

to all these environments. The main actors are: the Developer, the Version Control Repository, the Build and Continuous Integration Server [38], as specified on previous chapter. Next, a development pipeline with continuous integration capabilities will be proposed suitable for the RTS project.

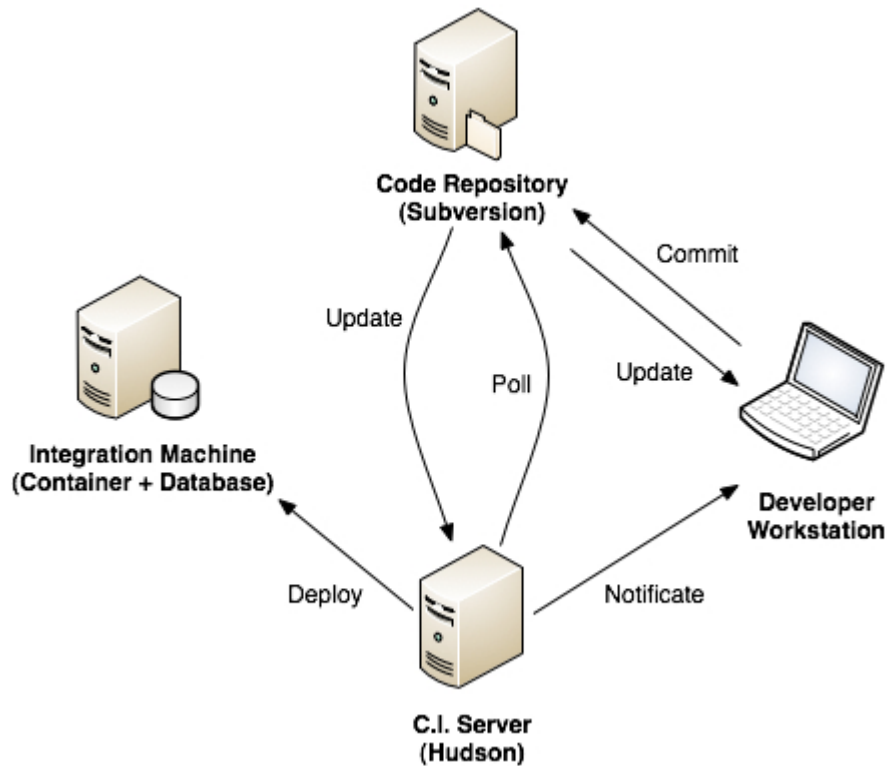


Figure 46 - Simplified RTS' development pipeline.

Figure 46 shows the integration environment set for the development of the RTS project. For representation purposes, Hudson (CI server), Subversion server and the integration machine are depicted separately though in fact they are installed on the same machine on RTS' development scenario. Before integrating new code into the pipeline, developers must update their Subversion working copy to receive new possible changes from the repository. This procedure should be done regularly for developers integrate colleagues' work into their private work. Then, the integration process is started when developers commit changes from their workstations to Subversion. At regular intervals, Hudson polls Subversion state to see whether there are new changes and update its working copy to start executing the build, which includes compilation, testing and packaging. After packaging the project's artifacts, Hudson will deploy them on the integration machine where integration tests are carried out. The database migration is included in this routine. The integration machine runs a Tomcat J2EE web container and PostgreSQL instances. After Hudson finishes the deployment of packages in the integration container, tests will be performed in order to guarantee that the integration of all executables and databases changes of that build work as expected.

5.2.1 The project's files repository

At the beginning of this work, the RTS' Subversion directory layout was not arranged to perform organized operations over multiple developing versions without mutual interference, or even in a scalable way. Therefore, the elected strategy to dispose the different development lines takes the default structure recommended for all Subversion repositories [11]. Such a structure consists in dividing the repository into different directories, each having its function. Some of those directories will have direct match with the different developmental lines the project may take in accordance with the active development versions and maintenance versions, as explained on the propose (chapter 4). This arrangement delivers improved organization and maintenance for project's files repository to handle the continuing evolvement and growing of the RTS project. The chosen layout consists of three main directories in the root of the project repository: *trunk*, *branches* and *tags*.

Trunk will hold the most highly evolved version of the development project and be the focus of continuous integration, which will be also the most unstable development line during the construction process. With few exceptions, *trunk* will be the main evolution line, reflecting the development of every new feature, even with the existence of parallel development lines. Hudson will use this directory as the primary target of its action, it is consequently important when adding new features to reflect them as soon as possible in the *trunk*.

The *branches* directory will have different roles according to existent maintenance versions and the amount of highly complex requirements under development. Will also be the most complicated directory to maintain and rules must be followed to prevent unnecessary files being placed in the repository. The exact use of branching within version control context of RTS project was discussed on the previous chapter. *Branches* are used to maintain releases and to stabilize release candidates (RC).

The *tags* directory contains copies of final and released versions therefore this directory must be regarded as read-only after generating a release. This directory enables to easily reproduce all production versions of the RTS application.

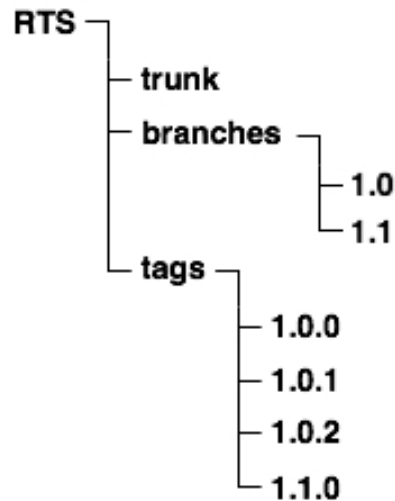


Figure 47 - Subversion's directory structure.

Figure 47 shows an example of the directory structure of RTS project's Subversion repository. The *trunk* contains the most actual developments of the project (mainline); the *branches* directory, in this case, provides two releases subject to maintenance, which correspond to 1.0.x and 1.1.x releases; the picture shows that the *tags* directory contains four final application's releases, where 1.0.1 and 1.0.2 account corrections to 1.0.0 release and have been generated from the "1.0" branch. Version 1.1.0 has not yet been liable to fixes, while the "1.1" branch has been created stabilizing the corresponding release candidate (RC).

On RTS' development scenario, quality assurance relies mostly on Hudson by using automated builds that include source-code quality analysis, unit testing or even system testing. Hudson will use most of its processing on building the *trunk*, which represents the main development line. Always committing into the *trunk* is a very effective way of development [12] since it is the only way to take full advantage of CI process. Developers should reflect all their work into *trunk* - with a few exceptions - because it ensures that all team's work is continuously integrated in the same place with instantaneous feedback, as stated previously. According to this philosophy, RTS development team must follow the established conventions when creating new branches and to use them only when different lines of development are not going to converge at the same point, as branches are considered independent development lines. However, atomic merges between *trunk* and *branches* may be performed during the stabilization period of RCs or when one performs bug corrections found on production machines, for example. Whenever a change is performed in a branch, the *trunk* must be evaluated to check whether makes sense to merge this fix since it might have already been done during the development process or neither be compatible among the different development threads.

5.2.2 Managing the software artifacts repository

The Maven dependency mechanism will allow RTS to have repository managers, in this case Nexus. Nexus allow to proxy public repositories to avoid downloading the same information numerous times each time a Maven build is executed. RTS developers' workstations, instead of downloading directly from the public repositories, may do so using Nexus that stores all the information that passes through it. Thus, the second time a given dependency is required within the development team, is loaded from the cache and is spared an unnecessary download. Another reason to use this proxies is the use of external snapshot dependencies. Maven will continuously check for new updated snapshot (development) versions and this mechanism will literally save seconds to RTS project's builds when external snapshot dependencies are used.

Nexus will be used as artifact deployment destination for all produced binaries from the RTS development team. This increases collaboration by automating module sharing for internal use rather than distributing them manually or building them to get the required artifact. Other external teams within IEETA may also use the RTS' binaries as external dependencies. In seconds, RTS final products (release) or developing products (snapshot) can be shared without the need of a formal request, with just the declaration of the required dependency within the POM, or by downloading it directly from Nexus. However, in order for this process to be effective, an automated scenario for sharing artifacts must be settled, along with the best practices on software development, including the continuous code integration.

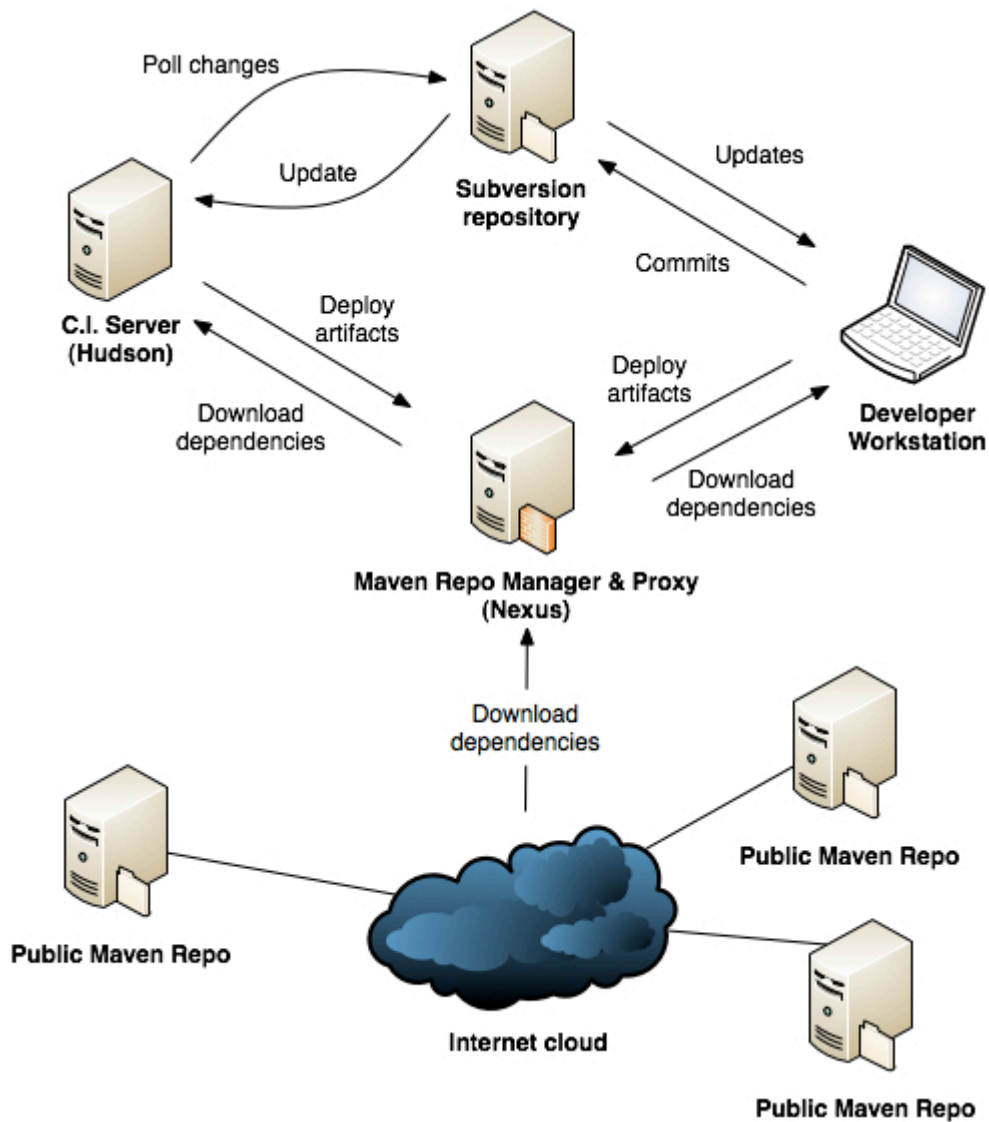


Figure 48 - Repository Management scenario on RTS project.

Figure 48 shows a representation of the adopted solution to implement the behavior described above. In spite of being installed on the same machine with Hudson and SVN server, Nexus is represented as being housed in a dedicated machine just for the sake of representation. In the depicted scenario, Nexus presents itself as the core of the whole process, as it will store all the generated artifacts from the organization, in this case, the RTS project. As described in the previous section, Nexus can prevent machines that rely on it to download dependencies directly from external public repositories, optimizing the network load by not downloading the same files repeatedly. Nexus will get the dependencies from the Internet, whenever a client requests the download, as illustrated. The picture shows that developers' workstations and Hudson will get the project dependencies when necessary, both after running a project build with Maven. Maven resolves dependencies and will do its best to find them on accessible Maven repositories.

The releases and snapshots repositories management is done with the help of Hudson. This process starts when the developer do a commit on Subversion, then Hudson, after a periodical update operation, gets the newly code developments, which will trigger a new build. If everything goes well on this build, Hudson will generate all the project artifacts and deploy them to Nexus. Hudson will automatically deploy the artifacts that are still under development (snapshots), making them immediately accessible for internal sharing. The released versions, in turn, are triggered in the process of launching a new final version. The release generation and its deployment to Nexus can be prepared on any workstation or using Hudson with a dedicated parameterized build on CI server, but always with human initiative. After each successful automatic build, Hudson will deploy all the recently generated artifacts on Nexus's snapshots repository. This repository may contain several development builds of the same artifact version, ordered by upload date. These files may be declared as dependencies on POMs by using versions numbers that denote the deploy date or a "-SNAPSHOT" suffix, which will get the most updated development version of the artifact (Figure 49). The use of different builds of the same development version might be helpful when the most recent snapshot version is broken due to possible programming bugs, and the RTS modules using it as dependency can specify the last build that worked on the dependencies list, just while the problem is being solved by the RTS' development team.

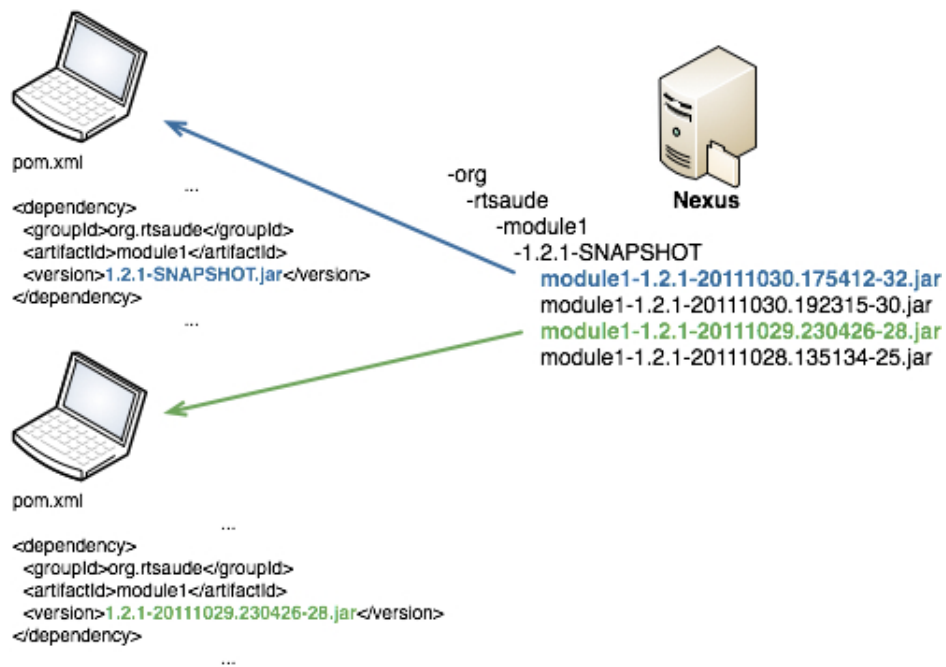


Figure 49 - Snapshot dependencies download.

There is a catch in this mechanism: The continuous build will generate several files that occupy a lot of space. Multiplying several times the size of all compiled files from the RTS project it will become rapidly unmanageable and fill the storage device in just a few days. Figure 49 shows an example of Nexus' snapshots repositories with multiple files of the same artifact. Therefore, it is necessary to control the amount of builds that must be kept in the snapshots repository. Nexus

allows scheduling the repository cleanup with just providing the minimum snapshot count for each artifact and number of days that should be retained. Within RTS project, the decided policy for the Snapshot Repository is keeping only the most updated file per snapshot version, because, for the moment, only one team uses the installed repository manager and the build process is fairly simple. When a snapshot version is broken, a member of the team should solve the problem immediately. On Figure 50 is depicted the Nexus' Scheduled Tasks configuration window where a task for daily removing old snapshot versions is programmed to maintain the repository clean.

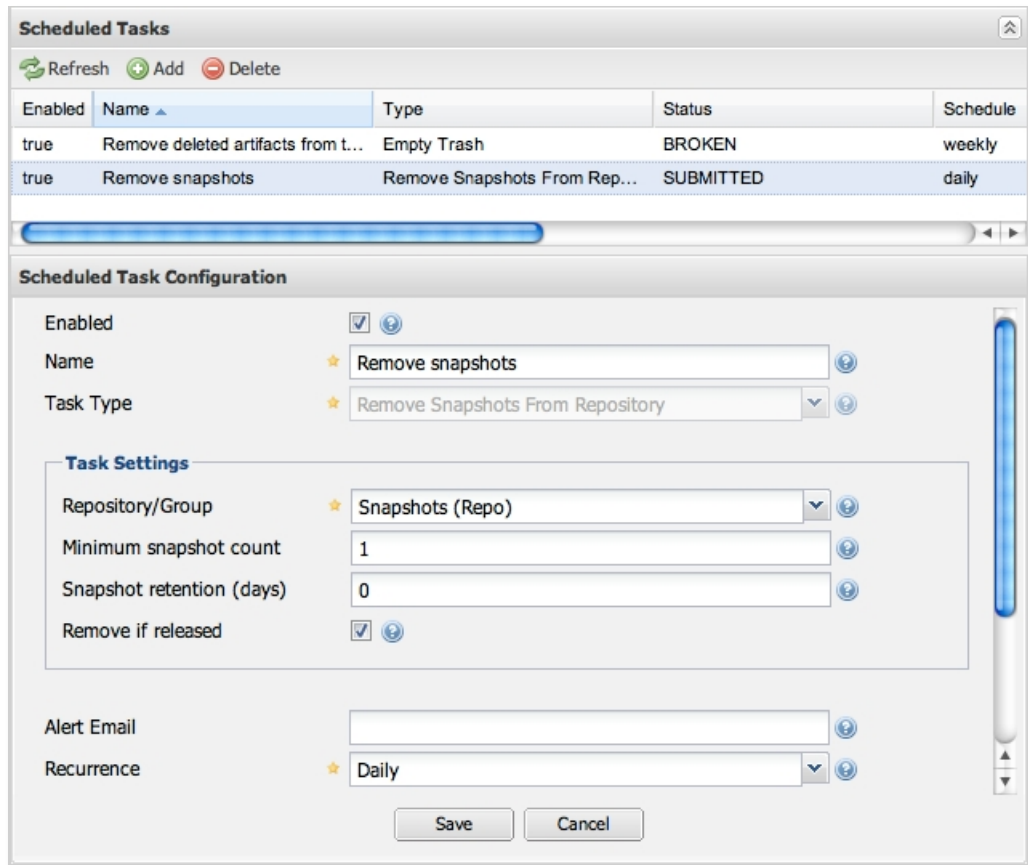


Figure 50 - Snapshots repository maintenance.

5.2.3 The continuous integration server

One of the most significant elements to combine all processes and tools described in this document is Maven, which is used in the RTS project. Its flexibility allows it to manage build executions, quality checks, the dependency management (internal and external) and more. Has also a key role in the continuous integration process because it enables the automation of the entire assemblage lifecycle, testing runs and distribution management, interacting with Hudson to supply all required feedback to ensure the validation and integration of all project's changes. Maven offers the ability to execute different build profiles of the same project that can be adjusted to different

environments. For example, a build carried out on a developer working station may behave differently than a build performed on the CI server. Such flexibility allows organizing different integration builds to render efficiency and interactivity to the entire process. The chosen CI server – Hudson - interconnects seamlessly with Maven and can activate profiles to help split and prioritize the continuous integration process because the execution of an entire build can be a cumbersome activity, and may prevent feedback to instantly be triggered rendering useless the CI. The combination between Maven and Hudson is ideal for tear down the continuous integration process in different stages with different priorities regarding the feedback on the integration result. It is also possible in certain circumstances to integrate more than a development line, each one corresponding to a different version. The development pattern of the RTS' development team focuses on integrating all new available features on the main development line (*trunk*), implying that these parallel lines, subject to the CI server's action, correspond to different maintenance versions of the project.

On the integration environment of the RTS project, the triggering process that enables Hudson to initiate a new build vary from the manual execution using the Hudson's GUI to the automated flow which is started if there are any changes to the project files in Subversion. If the build fails, a notification is sent to the development team containing the details why the build cannot be completed. A timer is scheduled to check regularly if there are changes in Subversion's repository, so if no changes were made, the build is not executed. Otherwise, Hudson will carry out its working copy update and execute a build for integration purposes.

5.2.3.1 Using multiple integration lines

A Maven build has a well-defined lifecycle, which comprises various stages ranging from cleaning the output files from prior builds to the deployment of binary files on Nexus. If the whole lifecycle is performed, it demands a lot of CPU consumption of the machine that runs Hudson. There are multiple lifecycle's phases that may take long minutes to run, in case the project is made up of many modules and files. A Maven build performed by Hudson, traversing its complete lifecycle may take longer than would be expected for the continuous integration scenario that aims to implement rapid cycles. In that case, code integration would not be agile, feedback would become slower and retrieving a working version of the application within minutes would be at risk. The solution to this problem is to split the build by using multiple inter-dependent *jobs* on Hudson, tuned for with different frequencies (Figure 52).

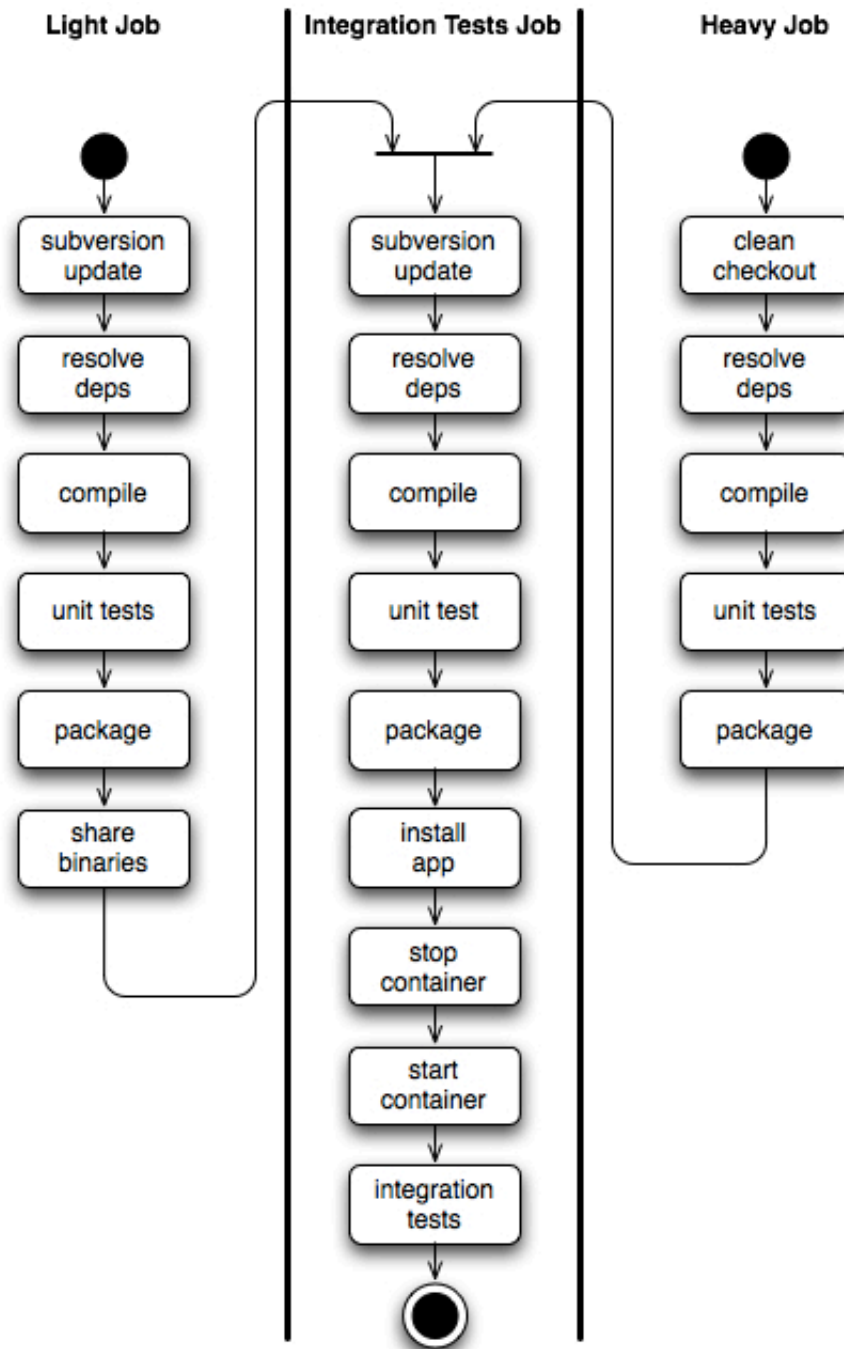


Figure 51 - Hudson integration jobs.

For purposes of rapid feedback when integrating new code, a *job* in Hudson was created which cleans up files resulting from earlier builds, compiles the source files, carries out automated unit tests and, finally, performs artifacts deployment to Nexus for sharing generated files. If the build fails at some point, the developer that broke down the code stability is notified to immediately fix the error. This *job* provides functionality in order to ensure minimal integration of new changes therefore its execution frequency is high. Thus, the Subversion repository is checked every 5 minutes by Hudson to spot new modifications on the project and, if any, the *job* is executed. An

update operation from code repository is performed prior to each run which implies that the build workspace is not pristine and earlier generated artifacts from previous builds remains intact until Hudson starts a new build on the *job*, which can lead to some build failures in occasional situations. This job is represented on the left on Figure 51. At the end of building this *job*, integrations tests will be conducted.

To supplement the previous *job*, a new one was set up to fulfill two main functions: run a base build on a pristine working copy and trigger integration tests, depicted on the right on Figure 51. This *job* is executed once daily at night but removes all files from earlier builds before performing a clean checkout from the Subversion repository before each build execution, which guarantees a clean but a slower build. A further difference from the previous *job* is the fact that doesn't perform artifacts deployment Nexus. By the end of execution, another *job* will be trigger to perform runtime integration tests.

The integration-testing *job* is always executed in sequence with previously described *jobs*. Such tests are performed on a dedicated machine therefore the very fact of scenario configuration premises exist, with oblivious failure points to Hudson, led to insulate the integration-testing *job* from the rapid integration *jobs*, like described on previous paragraphs, to prevent breaking the whole CI cycle. The performed lifecycle includes the package installation on Tomcat application server with Cargo in the integration environment and subsequently the execution of runtime integration tests.

At last, for code quality assurance purposes, static analysis tools are used, isolated in a dedicated *job*. Static analysis of all RTS project's files can take around 20 minutes to finish. This *job* is performed just once a week. Such check produces results based on a brief historical record to state the evolution of code quality over time, reinforcing the choice of weekly scans.

All		RTS
S	W	Job ↓
		RTS-trunk-heavy
		RTS-trunk-light
		RTS-trunk-light-integration
		RTS-trunk-reports

Figure 52 - Configured jobs on Hudson.

5.2.3.2 Continuous feedback mechanism

The CI environment relies on a notification system to immediately inform the development staff about the project status when problems are detected. It is vital that the team of developers is

always prepared to deal with implementation errors detected by Hudson. Hudson’s notification is based on the principle of “no news is good news” and starts whenever it detects a problem running a *job*. Bugs are typically introduced during the construction process by a RTS developer. As soon the error is detected, Hudson notifies the responsible programmer for the instability by notifying him via email and optionally notifying other collaborators. Once the developer is notified, shall immediately fix the error to avoid disrupting the proper functioning of development cycle and prevent the problem to propagate through the code, besides being able to stop the entire development team’s activity. If additional developers check in code changes to Subversion during this instability period, they will also receive notifications from Hudson about the existing problems. Following fixing the error and subsequent build execution, all parties will receive an email regarding the successful integration and team’s activity will return to normal.

Besides the built in methods, Hudson has the flexibility of supporting many other forms of notifications. In addition to the notification types mentioned above, there are *plugins* that span this functionality: Notification Plugin allows sending JSON messages with well-defined format of the integration result to a given machine and that information can be processed in many ways, since it is an API; Instant Messaging Plugin integrates with most available messaging applications and acts as a notification *bot* by that means; Status Monitor Plugin enables to visually notify the integration state to be presented on large screens fixed on the development team’s physical location; among many other developed *plugins* by the Hudson’s programmers community.

5.3 Software project changes management

5.3.1 The versioning strategy

For the RTS project, the chosen version-numbering schema is based on a structured triplet of integers: MAJOR.MINOR.PATCH, each one with a specific meaning described on Table 10.

Segment	Description
Major	When breaks up compatibility
Minor	When enough requirements are implemented and the changes are externally visible
Patch	Bug fixes without new requirements

Table 10 - Version segments.

Each segment of the project’s version number increment a single unit, depending on the changes that were introduced. The patch segment will be incremented only when bug fixes are

added to a specific version. This must not comprise changes on data structures and API definitions neither include new requirements. When enough new features are produced and possibly some bug corrections, the minor segment must be incremented. API compatibility should not be broken but new methods can be added and existent ones can be marked as deprecated. When large modifications occur, like breaking up compatibility with older versions of the application, the major segment should be incremented.

There are two kinds of versions to bear in when generating distributable files using Maven: release versions and snapshot versions. The first kind refers to final and published versions of the application. The second kind concerns development versions that overlap a released major, minor or patch version, depending on what's planned by the RTS project responsible. The snapshot strategy supports multiple versions being developed at the same time. When this scenario occurs and a release date is reaching the deadline, a release candidate (RC) version can be created to isolate those developments from the rest of the development pipeline, ensuring the maximum stability of the final version by not crossing unrelated work targeted to other releases.

On the ITS level of perspective, when a new release version is about to be closed, all issues targeted to that version, if possible, should be closed, tested and validated. If there is the impossibility of solving all the issues aimed for the version to be generated, their target should be shifted to a future RTS release.

5.3.2 Applying database changes

Typically, managing code files' changes is based on using management tools such as Subversion. Such management is accomplished statically, which enables the compilation of those code files to generate always the same result: binary files which always match to the same version. Installing a particular version of the system in a runtime environment, consist on copying all binaries to the machine, replacing the files from previous versions. A similar procedure does not apply on updating databases because the information contained herein differs on every scenario and just replacing the respective schemas is not enough. In order to upgrade the DB consistently with the application's version being installed requires migrating only new changes matching the new version, such as inserting new tables, or adding new constraints. Changes to be migrated must be properly marked so that the DB integrity not becomes corrupted. The procedure described above applies to the pretension of evolving an application version on a machine. The opposite, i.e. the regression of a version, should also be possible. Databases' versions control refers to the capacity, as in managing static files' versions on Subversion, to accomplish roll forward and rollback migrations, to record information about every atomic change, who implemented it and why. This feature extends the ability to trace all changes on databases, as intended for regular source-code files.

The most common strategy for migrating databases versions is using a table that indicates the installed version in each environment. Therefore, each set of changes in the database corresponding to the main solution's version, one must create a script that evolve the database to a newer version (roll forward) and a script that allows to recover from those same developments (roll-back) [12]. Thus, for each application version deployed in any runtime environment, it is always possible to know which scripts have already been applied in the database so that migration of newer versions apply only correspondent changes. Liquibase uses *changesets* files that are compatible with a variety of formats such as XML or plain old SQL, enabling straightforward DBs migration on the RTS project. It was decided to use SQL code on this project to conduct migrations: a *changeset* file for each of the four PostgreSQL schemas used in this project, because using SQL code is agnostic to every DB migration tool and also dispense new developers to learn a new lexicon to conduct such operations. Like the rest of files that make up the overall solution, *changeset* files are now versioned in Subversion. This means that changes in the DB schemas will be directly matched with versions of the overall solution. In the event of using PostgreSQL stored procedures on the project, the files containing them should also be included inside the SVN repository. However, its deployment on runtime environments will not origin the same difficulties like schemas migration because it is not necessary to record which store procedure is installed in a particular environment. Scripts containing the stored procedures can be executed every time a DB is migrated, as these new changes will override the existing ones. Liquibase implements this strategy and logs, on the occasion of migrating the DB, what new changes were held therein. During the development of new increments to database schemas, the RTS project's developers must uniquely identify, within each *changeset* file, such changes in SQL formatted comments with specific references. Thus, when applying the evolved *changesets* in a specific instance, these references will be recorded in the Liquibase table and, in a subsequent migration of the DB, they will be ignored applying just the further changes (Figure 53).

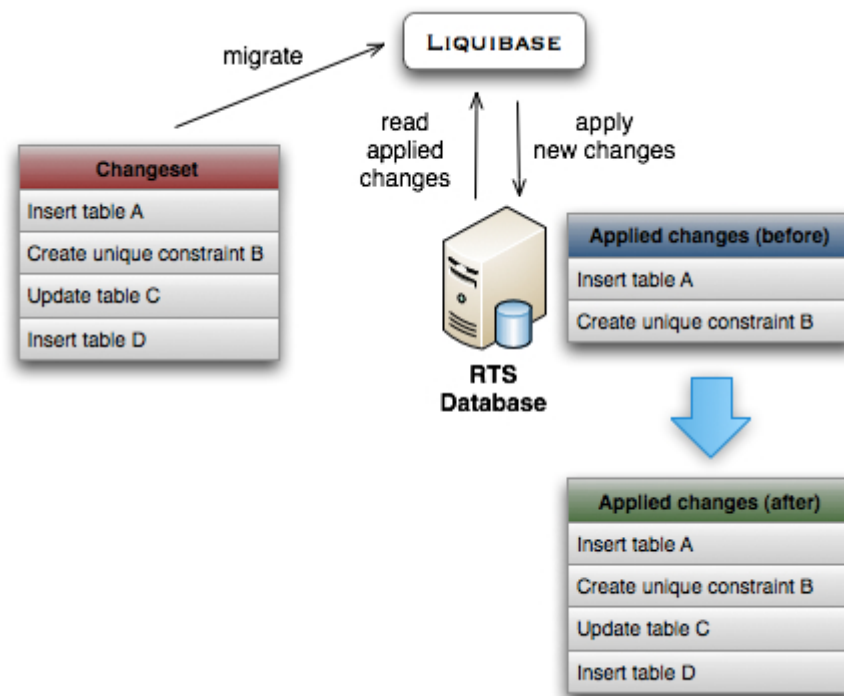


Figure 53 – Using Liquibase for database migrations.

On *changeset* files, for each rolling increment of schemas, a block of code should be created to rollback the correspondent change in cases where is necessary to reverse versions of the whole application. Liquibase supports the implicit rollback of certain operations, such as inserting tables and adding columns, but filling tables with data and removal of existent tables requires the manual specification of SQL code that allows reverting such changes because its automatic rollback is not supported. It is therefore advisable to include indiscriminately on every atomic change in schemas the corresponding SQL code to rollback it.

The engine used by Liquibase on each runtime environment tags the database with the version of the deployed application and the tag operation is performed in the process of deploying a new version of the RTS project. On production machines, executing the Liquibase standalone command-line application applies the migration of DBs and respective tag operation, while on the dedicated environment for integrating and testing the RTS system, Hudson server accomplishes such operations in an automated manner shortly after the deployment of the application packages on the runtime servers.

The produced code by developers for the RTS project will be regularly and automatically checked, built, subject to low-level unit testing, packaged and then deployed in the integration environment. During this process, the DB version, which will be migrated by Liquibase, is always consistent with the installed SVN revision (or version) of the application that is installed on the integration machine - in this case, that version matches a Subversion revision in a development

branch. Once the proper deployment of the application is done and correspondent migration of DBs is executed, the system is subject to automatic integration tests started by Hudson.

As mentioned in previous sections, there's a Liquibase plugin that allows migrate DBs with Maven builds, which was adopted by the RTS project to perform those operations in an automated way. The Hudson CI server uses Cargo Maven Plugin and performs automated deployments of the application in this scenario.

5.3.3 Releasing versions of the project

As mentioned on previous chapter, before launching a RTS application release, all tasks, bugs and requirements for that release should be validated and closed. Releasing a new version of the application may rely on those issues' statuses but is not always feasible to finish them as scheduled, which may impose the development team to shift them to further releases by ensuring that no one remains unsolved. Figure 54 shows issues' statuses as used on Redmine by the RTS project. When an issue is created and assigned to a particular developer, it has the "new" status but when the related work is finished, he should change the issue status to "resolved" and immediately assign it to another developer for validation purposes. Then, the new assignee must ensure that the original developer's work was properly carried out, by validating the issue. If it is the case, the issue must be set to "closed", otherwise, the second assignee must send the issue back to the implementer to complete or fix the work. When an issue is closed by proper validation, it is deemed as successfully resolved and ready to be included on final RTS' releases.

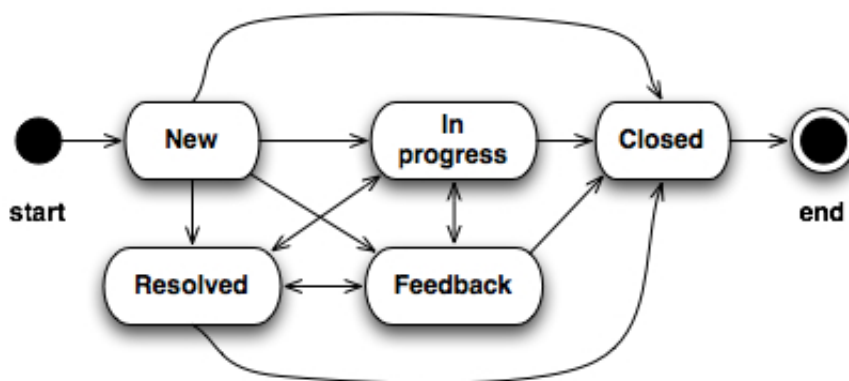


Figure 54 - Typical issue statuses.

RTS project responsible must continuously check all corresponding issues statuses during the development process. Closed issues were validated and successfully resolved by the RTS development team, otherwise they are not ready to be included on the final release. Redmine allows checking those statuses and to easily move unfinished issues to further releases, even when a release is about to be completed. Figure 55 shows how to bulk shift a set of issues to another versions, when they are not ready to be released.

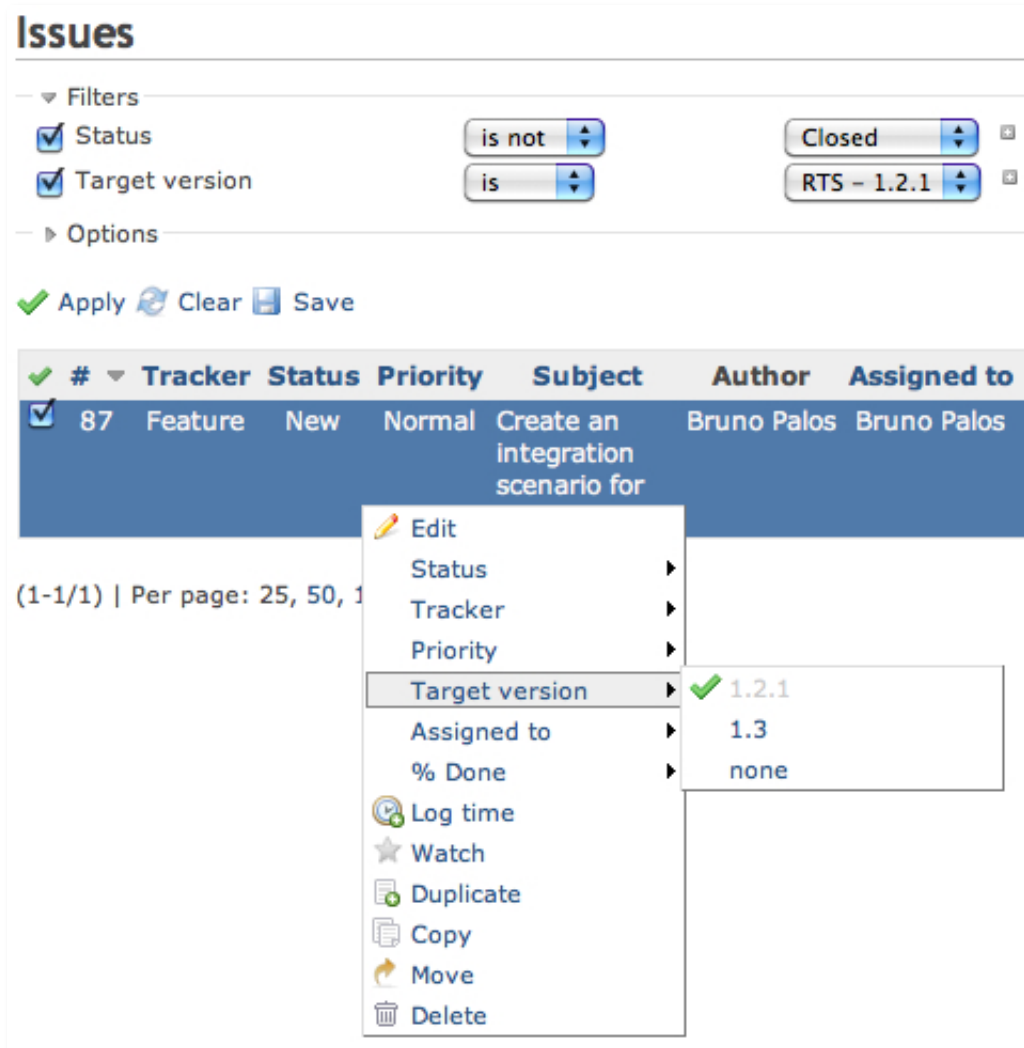


Figure 55 - Shifting issue to another version on Redmine.

Following the guarantee that all release issues were validated, even if that entails shifting unfinished and invalidated issues to future releases, the development team can perform the release of the RTS application. There are two distinct kinds of release, as explained in earlier chapters: releases with features (major and minor) and maintenance releases (patches). Next, it will be explained how to easily perform major, minor and patch releases using Maven.

As pointed out in above chapter, prior every version release, a release candidate (RC) is created for stabilization purposes. The RC may be established upon completion of all features aimed at the final release, even if they were not yet validated. Stabilization requires solving bugs found during validation phase of new developed features, aimed to that release, or solving older bugs. For RTS project, stabilization branches will always be used in a view to separate mainline (*trunk*) developments from the version about to be launched. Procedures to generate release candidates and final releases will be conducted in a semi-automatic way because it requires

executing a fixed sequence of operations, which become repetitive and clinging to errors when run in a fully manual manner.

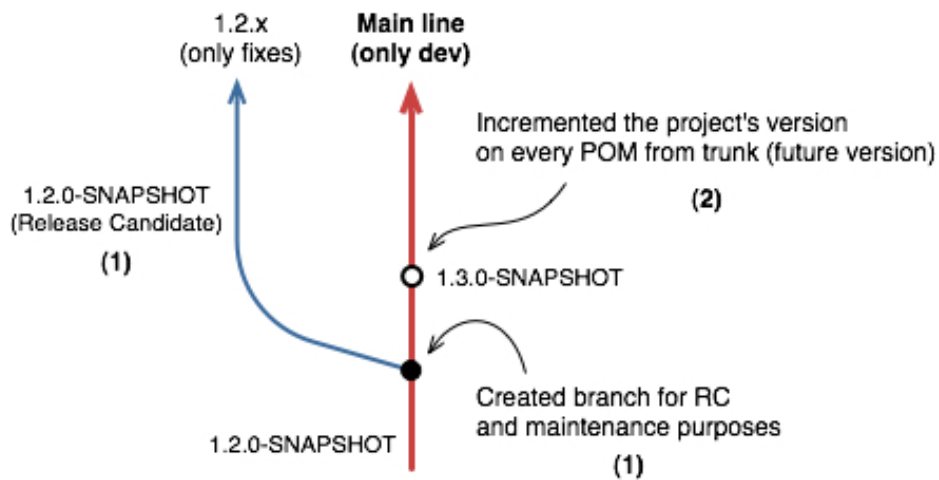


Figure 56 - Creating a RC branch.

Figure 24 shows what is required to create a branch to a release candidate (RC). If performed manually, the sequence of needed steps are:

- Check for changes to be submitted to the Subversion repository;
- Create a version branch (1);
- Increase the versions of all POM files in the *trunk* to the next development version (2);
- Submit changes to POM files.

As seen on figure, at a given instant, it was decided a RC to be produced by creating a new branch (1). In this illustration, the version that is being developed is the 1.2.0 and will be transposed into the new branch, where it will be raised later as a final release. The *trunk*, following the establishment of the branch, will bear the development of the next release (2), which in this case will be 1.3.0. Thus, the development of that release may optionally start while the period of the 1.2.0 release stabilization follows in the new branch. The semi-automatic process of generating RCs in the RTS project will be done using the Maven Release Plugin. This *plugin* allows developers to perform all above steps automatically, except the definition of the next development version, which will be prompted during the execution of the *plugin*. When the “branch” goal of this *plugin* is run, it will prompt the user about the next development version (in the figure example: 1.3.0-SNAPSHOT) and then automatically execute all steps outlined above, reflecting what is illustrated in Figure 56. All RC stabilization related issues should be submitted to the new branch directly and, if pertinent, those changes should be merged to *trunk*, so that these problems do not recur in the next version.

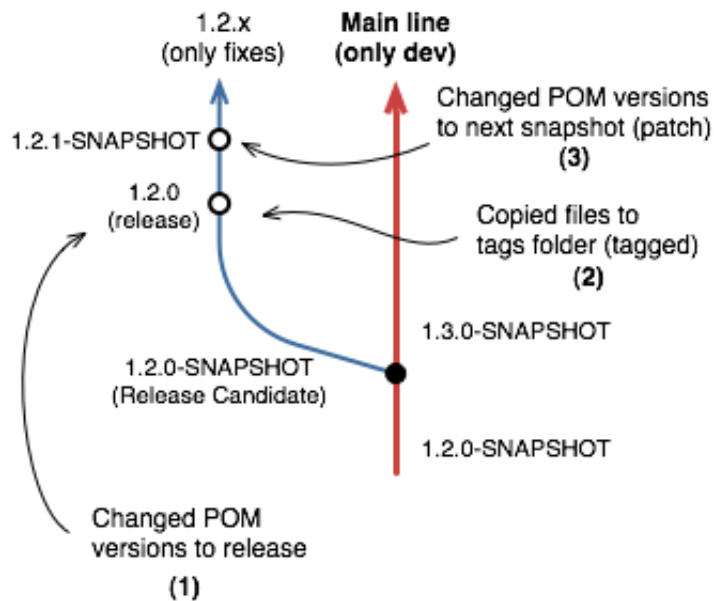


Figure 57 - Releasing process steps.

Once the RC is stabilized, with every issue closed and validated, the next action to take is to perform the final release. Figure 57 shows the steps that must be followed to release versions from the branch. Those versions could be the RC itself (for major and minor releases) or, in maintenance situations, patch releases. The sequence of steps to prepare the final release is:

- Check for changes to be submitted to the Subversion repository;
- Check if there are dependencies in the project still under development (snapshots);
- Change POM files' versions of the branch to final release (1);
- Perform automated tests with the new version set on POM files;
- Commits changes to POM files;
- Create a version tag, copying files to *tags* directory (2);
- Change back version POM files' version of the branch to next patch version, for maintaining the major/minor release (3);
- Commits again to Subversion the changes to POM files.

The extensive list of steps that must be taken to prepare the release may lead to oblivion or errors if performed manually. Therefore, as like creating branches, releasing RTS final versions is done semi-automatically using again the Maven Release Plugin. The execution of this *plugin* using the "prepare" goal runs all the steps described above, after questioning the user about which versions are involved in the process: the final release literal and next development patch literal on the branch, as the ultimate objective of branches is maintenance. However, the release is not complete without publishing binaries to Nexus repository. The Maven Release Plugin enables also holding such task, using the goal "perform" and perpetuate the release as complete. Figure 58 resumes the Maven Releases Plugin goals used by the RTS development team to perform releases of the application.

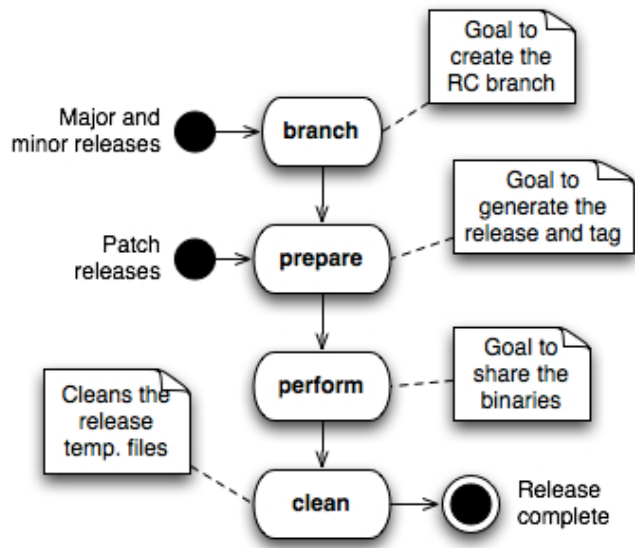


Figure 58 - Maven Release Plugin mapping with RTS' releases strategy.

A release is only finished when it is taken for closed on Redmine. Figure 59 shows how to close an RTS application release.

RTS

Overview Activity **Roadmap** Issues New Issue News Document

Version

Name *

Description

Status

Wiki page

Date

Sharing

Figure 59 - Closing a release on Redmine.

5.4 Automated tests on every build

5.4.1 Automated integration tests

In the RTS continuous integration (CI) environment, a dedicated scenario for integration testing was created. The responsible entity for performing such tests is Hudson, which are frequently run during integration cycles of the developed work by RTS team. Integration tests, as previously explained, are executed following each change on project's files and Hudson conducts them independently from the rapid integration *jobs*. This prevents the main CI *jobs* to stop running rapid integration builds when integration tests detect some issue with the system components. Yet, the job that runs integration tests will notify RTS developers if such problems arise on the project. Every build started by Hudson will produce the executable files of the most updated version of the trunk and install all of them in the live environment, right after the migration of all RTS databases in that scenario, making them coherent with the remaining application and mimicking a real environment. Not until the end of the deployment on Tomcat and the migration of all databases, the integration tests are performed. Hudson performs the complete installation of the application on the integration scenario, including all database changes, in a completely automatic manner. Figure 60 shows a representation of that runtime environment. Hudson uses Maven builds on every integration cycle with resource to three Maven *plugins*: Liquibase[115], [116], Cargo[117], [118] and soapUI[46]. Liquibase is a database (DB) migration tool; Cargo remote controls J2EE containers like Tomcat - used by the RTS project - and supports deployments of java applications and to stop and start the container. soapUI is a web services (WS) testing framework that allows creating tests suites with multiple WS parameters, including WS security.

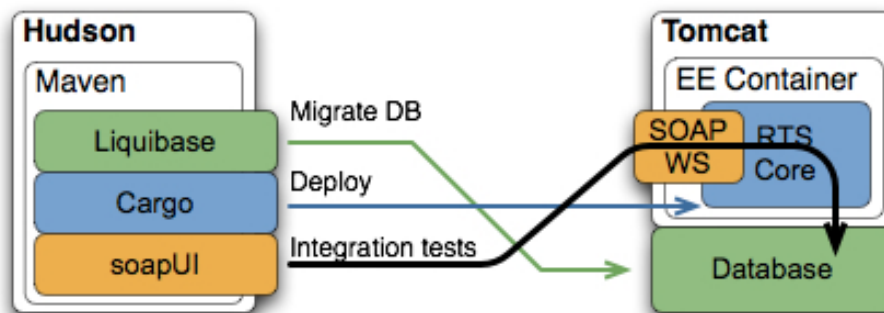


Figure 60 - Integration testing.

An integration tests suite was created using the soapUI graphical tool (Figure 61) to access the web services interface of the RTS project, the *RTS.WS* module. The suite tests the integration between DBs, the RTS application core and the *RTS.WS*. Using the WSDL (Web Service Definition Language) file, the test suite was generated with multiple interface binds and configured to simulate remote requests for each remote access points of the *RTS.WS* interface with

authentication support. Figure 28 shows a test execution over the WS using the soapUI graphical tool.

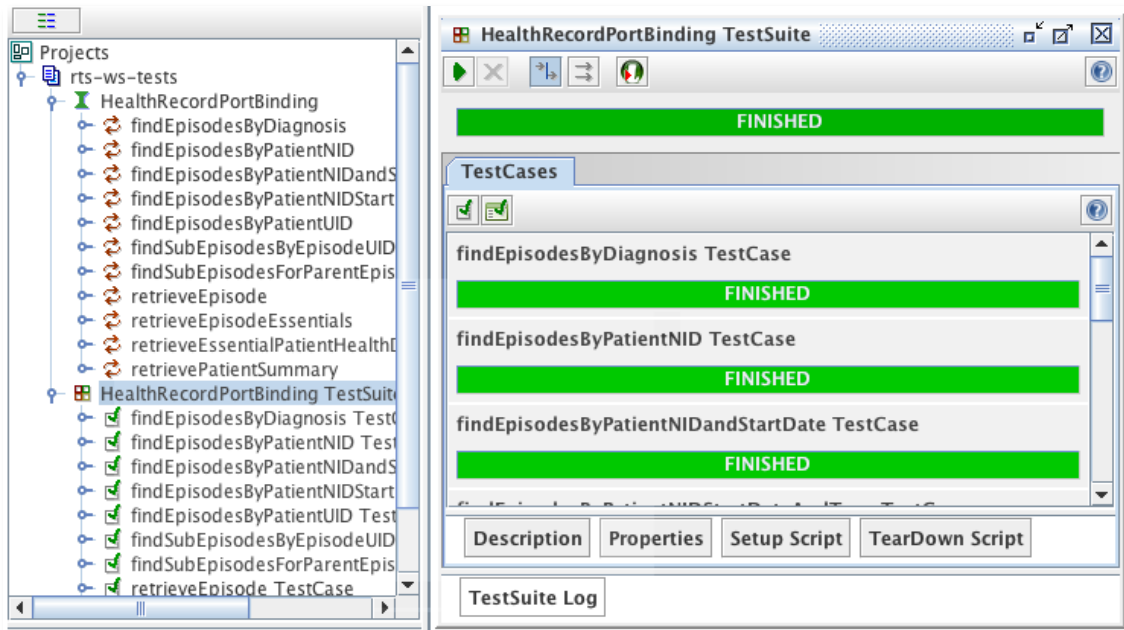


Figure 61 - Using soapUI to create a test suite.

The soapUI framework saves the test suites in XML format that can be used by the soapUI Maven Plugin. The created soapUI test suite was settled on a Maven build for use by Hudson on the integration tests job so as to be executed after migrating the database and deploying the application on Tomcat (Figure 60). Figure 62 shows the Hudson's integration tests job updating one of the RTS' databases using Liquibase Maven Plugin. This *plugin* will directly access databases using a JDBC driver for PostgreSQL and perform the schemas migration to be coherent with the rest of the RTS application modules on the integration runtime environment inside Tomcat (Figure 60). Deployment of the RTS application no Tomcat application server will be done next.

```
[INFO] Building RTS Databases
[INFO]   task-segment: [clean, install]
[INFO] -----
[INFO] [clean:clean {execution: default-clean}]
[INFO] [liquibase:update {execution: rtsmail-integration}]
[INFO] -----
[INFO] Executing on Database: jdbc:postgresql://heartbeat.ieeta.pt:543
INFO 6/2/12 12:09 AM:liquibase: Successfully acquired change log lock
INFO 6/2/12 12:09 AM:liquibase: Reading from databasechangelog
INFO 6/2/12 12:09 AM:liquibase: Reading from databasechangelog
INFO 6/2/12 12:09 AM:liquibase: Successfully released change log lock
INFO 6/2/12 12:09 AM:liquibase: Successfully released change log lock
[INFO] -----
```

Figure 62 - Database automatic migration on RTS dev. environment.

During the Maven build, Cargo will update the Tomcat J2EE Web container with the RTS application. As shown in Figure 63, Cargo will first *undeploy* the previous version of the installed application before deploying the newly built version by Hudson. After this process, the soapUI test suite will be run and invoke the WS interfaces of the RTS application to check if the integration of the supporting modules were done successfully. On error, Hudson will fail the integration tests job and notify the RTS development team about the integration problem.

```
[INFO] [cargo:deployer-undeploy {execution: deploy-cargo}]
[INFO] [mcat6xRemoteDeployer] Undeploying [/home/hudson/.hudson/jobs/RTS-t
integration/workspace/rts/RTS.WS/target/RTS.WS.war]
[INFO] [cargo:deployer-deploy {execution: deploy-cargo}]
[INFO] [mcat6xRemoteDeployer] Deploying [/home/hudson/.hudson/jobs/RTS-tru
integration/workspace/rts/RTS.WS/target/RTS.WS.war]
[INFO] [soapui:test {execution: test-ws}]
soapUI 4.0.1 Maven2 TestCase Runner
00:14:30,202 WARN [SoapUI] Missing folder [/home/hudson/.hudson/jobs/RTS-
for external libraries
00:14:30,349 INFO [DefaultSoapUICore] Creating new settings at [/root/soa
00:14:39,249 INFO [WsdProject] Loaded project from [file:/home/hudson/.h
integration/workspace/rts/RTS.WS/src/test/resources/rts-ws-tests-soapui-pr
00:14:40,569 INFO [SoapUITestCaseRunner] Running soapUI tests in project
00:14:40,571 INFO [SoapUITestCaseRunner] Running Project [rts-ws-tests]
00:14:40,584 INFO [SoapUITestCaseRunner] Running soapUI testcase [findEpi
00:14:40,587 INFO [SoapUITestCaseRunner] running step [findEpisodesByDiag
00:14:41,601 INFO [SoapUITestCaseRunner] Finished running soapUI testcase
time taken: 976ms, status: FINISHED
```

Figure 63 - Running integration tests with Hudson.

This procedure is repeated whenever a change is made to RTS project files in order to ensure fast integration of the components involved on tests, preventing regression issues to be detected hours, days or weeks later.

5.4.2 Automated unit tests

As an example for the RTS development team, a unit test case was created using JUnit to test a helper class of the RTS project. The example below shows the concept and the benefit of using automated unit testing prolifically during development phase on the project. Using the default Maven configuration, a test case was placed on the “src/test/java” directory of the same java package of the class being tested (Figure 63). The tested class is “CitizenCardParser.java” and the unit test class is the file “CitizenCardParserTest.java”, both shown with an arrow in Figure 64.

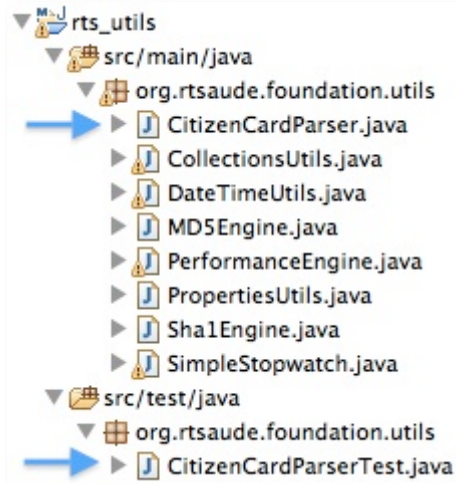


Figure 64 - Unit tests directory structure.

Using Maven default configuration, the only steps to automate unit tests on Maven builds were the JUnit dependency configuration on the project's parent POM file and to place the unit test class on the default tests directory. In the unit test file, JUnit annotations were used to specify which methods should be tested by the framework and the code were developed to test the methods logic using JUnit's assertions. So, every Maven build will execute the test case on the tests phase of the Maven lifecycle, whether locally, on RTS developers' workstations, or by Hudson during integration builds. This automated testing process is advantageous for continuous integration because it verifies eventual regression problems when the unit tests coverage is significant. For the case of RTS project, the shown example is the first developed automated unit test but it is expected the RTS development team to create automated unit tests during every feature implementation to increase the code test coverage.

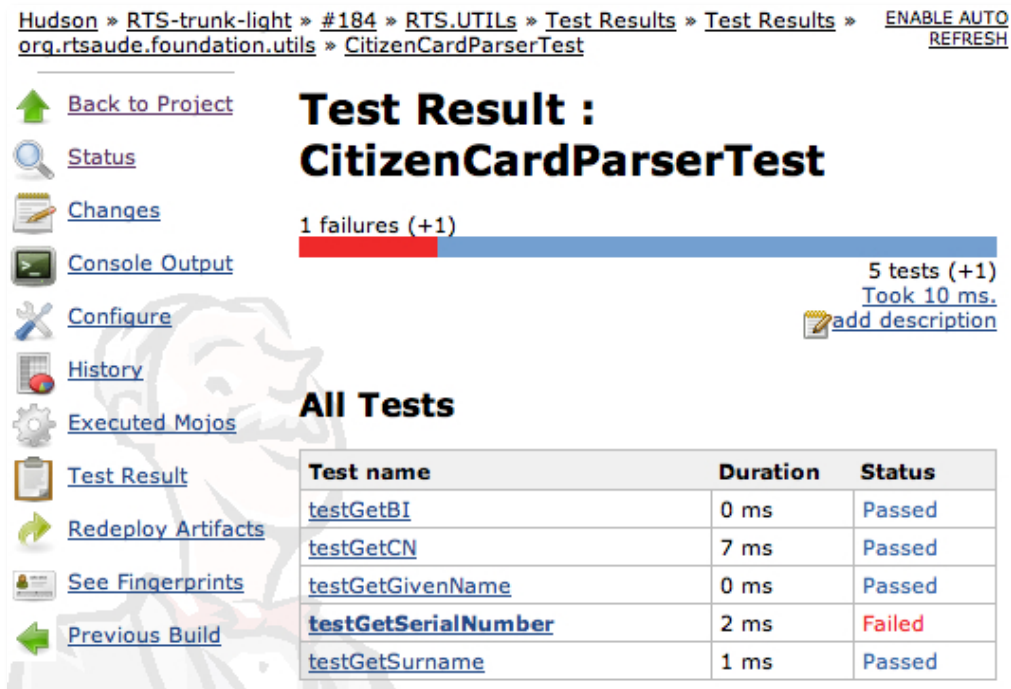


Figure 65 - Test result with fails.

Figure 65 shows the previous test case execution by Hudson on an integration build. As shown, the “CitizenCardParserTest” test case analyses five methods and one failed. The error was simulated to exemplify the negative feedback performed by Hudson upon a unit testing error, which generates an e-mail notification to the developer that introduced the problem on the code. Using the Hudson’s web GUI, it is possible to see details about the testing results, including the cause and a graphic with previous tests runs. This information includes a simple test coverage analysis about the number of unit tests used during recent integration builds. The static analysis of automated unit tests coverage will be addresses in the next section.

5.5 Automated code verifications

Hudson will carry out code static analysis every week on the RTS’ project files, although RTS’ developers should use static analysis *plugins* on their IDEs to check for bugs in real time. Hudson will focus code analysis on controlling the code quality by analyzing and checking the quality advancement and placing the results into graphics and tables. Various static analysis tools were installed on Hudson in order to carry out constant code checks. The most relevant are: FindBugs, PMD, Checkstyle and Cobertura. FindBugs performs analysis on Java code and detects bugs patterns, divided by categories and severity. PMD can detect unused code, overcomplicated expressions ad other bugs, besides detecting duplicated code – “copy paste detector”. Checkstyle examines the use of Java code conventions by the RTS’ development team, such as the number of

characters per line or proper code documentation. Lastly, Cobertura is a tool for tests coverage, or code coverage, which calculates the proportion of code that is acceded by tests.

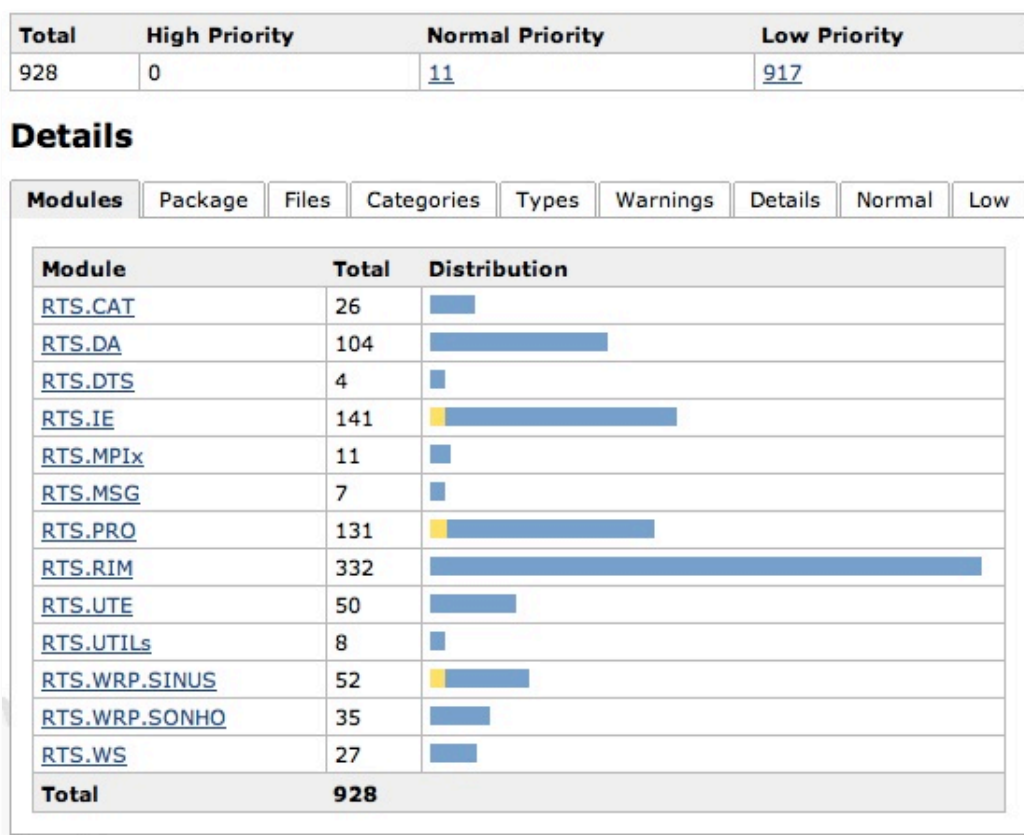


Figure 66 - FindBugs summary on Hudson.

A weekly *job* was set up on Hudson to perform code static analysis. The results are brought on formatted tables and graphics, organized by categories. The results are available by kind, per project module, per package, per file and per code area inside each class file of the RTS project. Figure 66 shows a FindBugs results table provided by Hudson, with bug distribution by RTS modules and showing the amount of bug patterns found, divided by severity. Hudson enables to analyze every bug in detail, namely the exact line of code where was found. FindBugs even suggests a way to fix the bug, as shown on Figure 67. The example presents a bug on “DischargeLetter.java” file, on line 24 with a low severity but programmers should have critical thinking with FindBugs or other tool’s hints because some of the found patterns are false positives.

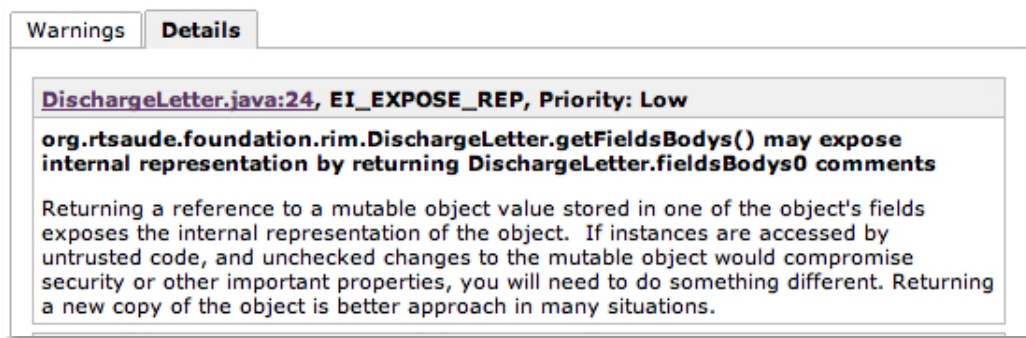


Figure 67 - Problem detail on Hudson found by FindBugs.

The most effective means to enhance the quality of code is prevention, not detection. Detection assumes the bugs were introduced and detected by verification, like by static analysis tools. Each RTS developer workstation should have static analysis *plugins* installed on their IDE, like FindBugs, PMD and Checkstyle *plugins*, so that bugs are displayed in real time on source-code editors. Figure 68 shows FindBugs *plugin* on Eclipse showing a line with a bug pattern. As on Hudson, it is possible to get the pattern details and correction hint.

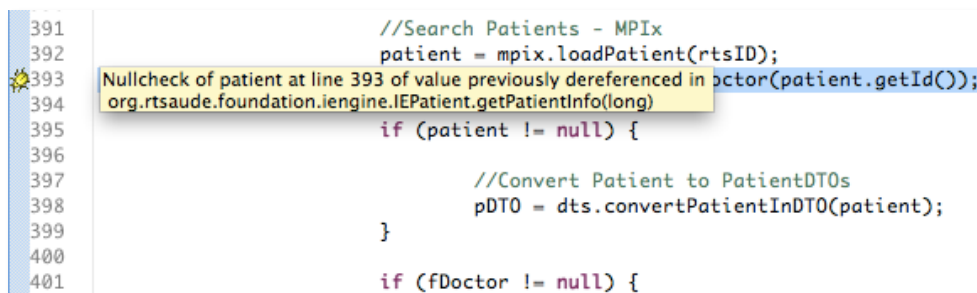


Figure 68 - Bug found by FindBugs on Eclipse.

The PMD tool as well as detecting different types of bugs like FindBugs, detects duplicated blocks of code (Figure 69). Hudson allows searching all occurrences and checks exactly where are the duplicated blocks on the project's files and on which lines. These events are divided by severity depending on the number of duplicate lines in each instance. That is, on each occurrence of duplicate blocks, the more lines have been copied, the greater the severity. Code blocks duplicates can be easily fixed using code *refactors*, which improve the code quality and maintainability.

```

---
601         IClinicalDataWs iclinical = new IEClinicalData();
602         IPatientWs rtsiepat = new IEPatient();
603         IEpisodeWs iepisodes = new IEEpisodes();
604         PatientSummaryResult result = new PatientSummaryRe
605         try {
606             PatientDemographics pd = rtsiepat.retrieve
607             PatientEHR pehr = new PatientEHR();
608             pehr.setActivePrescription(iclinical.getAc
609             pehr.setDevice(iclinical.getDevice(pUniver
610             pehr.setEpisodes(iepisodes.getEpisodesRoot
611             pehr.setEssentialClinicalData(new Essentia
612             pehr.setInvalidity(iclinical.getInvalidity
613             pehr.setPregnancy(iclinical.getPregnancySt
614             ons(iclinical.getSc
615             cal.getVaccinesChar
616             aphics(pd);
617         hr);
618         ex) {
619         ryResult(StatusCode
620         }
621

```

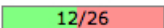
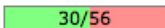
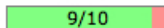
10 lines of duplicate code.

Duplicated in:

- [org.rtsaude.webservices.HealthRecord \(644\)](#)

Figure 69 - Duplicate code detection example.

As noted in the previous section, automated unit tests proliferation on RTS project will help detect regression issues immediately during continuous integration cycles performed by Hudson. So it is important to identify which areas of code are being subject to automated tests. Cobertura tool was installed on Hudson to detect which lines of code of the RTS project are covered by tests. The obtained percentage allows giving development team the awareness whether critical modules are being verified by continuous integration (CI). Figure 70 illustrates a code coverage analysis to the “CitizenCardParser” class file, referred on the preceding section and subject to automated testing with JUnit (Figure 64). This class has 10 methods and, according to coverage analysis, only 9 are being subject to the action of the unit test file “CitizenCardParserTest.java” (Figure 64). For each of the class’ methods, the coverage analysis examines which lines are being covered by tests and, in this case, only 53% of all class’ lines are being tested. This figure also demonstrates that 12 of the 26 possible conditions are run by tests. Analyzing test coverage results allows improving the effectiveness of developed unit test cases and to check which areas of code needs to be tested, aiming the objective of almost full coverage, while not being possible in many cases.

Name	Conditionals	Lines	Methods
CitizenCardParser	46% 	54% 	90% 

Coverage Breakdown by Method

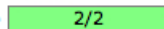

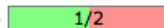
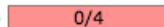

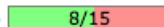
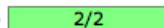
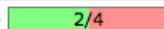

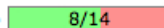
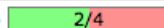

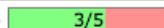
Name	Conditionals	Lines
java.lang.String getCN(java.lang.String)	N/A	100% 
void <init>()	N/A	50% 
void <clinit>()	N/A	50% 
java.lang.String getSerialNumber(java.lang.String)	N/A	0% 
java.lang.String getFieldBI(java.lang.String,java.lang.String[])	42% 	53% 
java.lang.String getSurname(java.lang.String)	N/A	100% 
java.lang.String getBI(java.lang.String)	N/A	50% 
java.lang.String getField(java.lang.String,java.lang.String[])	50% 	57% 
java.lang.String getGivenName(java.lang.String)	N/A	50% 
org.rtsaude.foundation.utils.CitizenCardParser getInstance()	50% 	60% 

Figure 70 - Test coverage of CitizenCardParser class.

6 Results

We carried out a survey about software development practices and processes in the Portuguese industry of small and medium-sized companies (see Annex A). From the survey outcome, organizations have expressed interest in adopting construction practices aimed to improve the quality of their software products. Nevertheless, there is still great room for improvement in some key points of the used processes and practices. During the request for participation, some companies revealed strong interest to contribute and highlighted the significance of the survey in the software development field. Such interest was expressed in the high participation rate: 25%. The results were compiled in a report and sent to the participating companies (see Annex B), which some of them replied with positive feedback.

To properly deploy the proposed construction process on the RTS[3], [4] development environment, all team must comply the defined steps and practices. For such changes to be effectively embraced, the project's collaborators have to learn and perceive the core procedures and follow them with discipline. In addition, the relative complexity of the proposed construction process propelled the establishment of a best practices document for use by the RTS project's developers (Annex C). This document explains the proposed development practices without mention particular details for easier assimilation after reading. It also contains guidelines for developers, specific design patterns and strict Java code conventions. Figure 71 presents the covered topics on the best practices document and their correlation. As seen in the illustration, the document focuses more on programming subjects and less on management matters.

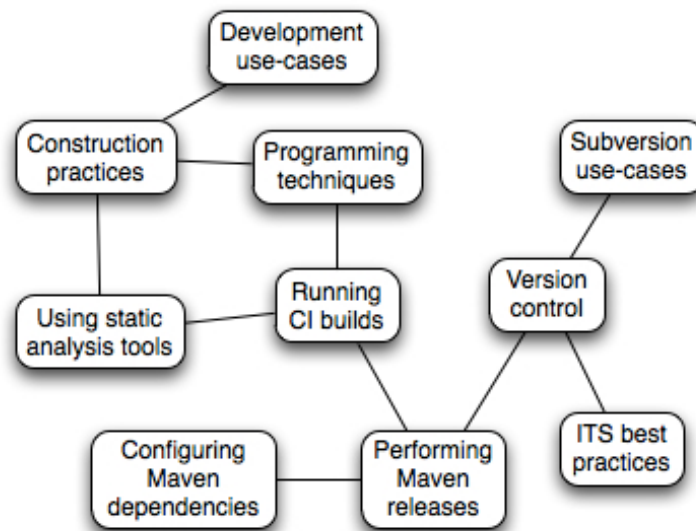


Figure 71 - Best practices document's structure.

The implementation of the proposal submitted by this dissertation changed the way the software construction takes place in the RTS project. The new introduced practices altered the

programmers' habits and have brought improvements in the way the project is built. One of the most important features of the new construction process is continuous integration, which led to switch the code production paradigm on the RTS project. Releasing versions of the application also became more effective: Maven[72] releases and Subversion's directory structure now supports the defined versioning strategy. Releasing versions of the RTS application is now a more agile process, which helps to efficiently uphold handling over the developmental environment. Project's dependencies management and database change management were also successfully achieved by equipping the project greater control over the different RTS' components. Seamless databases migrations are now possible and allowed to purge the risk of hazards on sensitive data during deployments on production environments.

We could not confirm the benefits of automating unit tests using continuous integration. Due to low development pace and the large project size, the growth of automated unit tests in the RTS project was not feasible. Thus, the quality of produced code did not notably improved during the course of this work. The use of integration tests has also not evolved, in spite of the establishment of a runtime integration scenario with support for automated deployments of databases changes and binaries of the RTS application.

The progress management and adaptive development did not suffer relevant changes that allow this dissertation to create a pertinent conclusion.

7 Conclusions and future work

This dissertation features a proposal for improving the software construction processes and practices on the RTS project [3], [4] that allows being compared with the practices used on the Portuguese software industry, based on the outcome of a survey (Annex A and Annex B), carried out in the scope of this work. The survey results showed that there are differences about the given meaning on the different construction practices between the industry and the proposal presented in this work. According to those results, the Portuguese software companies attach great significance to functional tests and most of them claimed that a specialized team tests their products. Given the nature of the RTS project, functional tests are conducted by the RTS development team itself, despite the proposed construction process suggesting the achievement of such validations by elements that did not developed the features being tested. Placing emphasis on proper use of tasks management platforms is greatly valued by the Portuguese industry like the proposal we presented on this work.

Besides the divergences in the use of functional testing practices between the Portuguese companies and the proposed construction process, we found other major differences: continuous integration (CI) and automated testing are substantially more valued by the proposal we present than by the present industry. Unlike the generalized belief of software companies, this dissertation supports that CI is vital in the process of developing software. The adoption of such technique outweighs the lack of testing resources on the RTS project. CI enables tight change control using automated tests, which can virtually cover all source-code lines and thereby improve the final product.

The proposed construction process is endowed with some characteristics that define agile development. This work is more focused on code construction practices than on management methods, like agile methodologies. However, conclusions can be drawn about the proximity of this proposal to such methodologies and agree on whether the agile philosophy is close to be met or not. The proposed construction process is explicit in some of the items specified by the agile manifesto[20], such as adaptation to changes, evolutionary development and valuing communication between individuals, however it is ambiguous on the definition of highly collaborative interactions and does not specify the fixed time iterations during development cycles. It also does not require frequently delivering value to costumers. However, the proposed work does not hinder the adoption of a particular agile methodology since there are no mismatches between the agile philosophy and the presented practices. Characteristics of academic software projects may inhibit the adoption of a specific agile methodology because most of them require experience, discipline and closeness between individuals and these traits may be insufficient in the academic scope.

A major obstacle to implement the new construction process in the RTS project is the relative complexity of the proposal, which encompasses several disciplines of software development. For every software project it takes time for teams to absorb and perceive every practices defined by new ways of working, combined with the lack of training and coaching about the new construction process[119]. We have not provided initial training to the RTS development team about the practices described in this dissertation and that may be the key to properly implement them on the project.

The implementation of the proposed work on the RTS development environment shows that organized processes enhance efficiency in the production of products and services. Despite the inconclusive results about the final quality of the RTS application in the eyes of customers, the new construction process helped to ensure greater control over the project. Like on other industries, process improvement adds value to the end product [120].

The assembled platform supports all sorts of automated tests but there are only two automated test cases, created in the context of this dissertation: a single unit test and an integration test, which checks the operability between Web Services interfaces, RTS' internal components and the databases. It is necessary to produce massive amount of automated unit tests for taking the best out of potential of this kind of validation. The analysis of code coverage may be vital to check what critical regions should be checked by automated tests. It is also necessary that automated integration tests cover more project modules and not only those referred previously.

Continuous integration may also support additional kinds of automated tests: functional tests, which can easily be carried out using tools like Selenium[45] (automates Web browsers), and load/performance tests using tools like JMeter[32], which should help to study the application's behavior on different environments.

The definition of the construction process was incomplete because it did not cover certain important matters of software configuration management: automatic packages installation and application configurations management. To speed up the delivery process, installation packages should be used with support for packages versions management. The application configurations management is typically concerned with portable configuration files and is responsible for managing how the application behaves on different situations: during builds, during deployment and during runtime.

Depending on the prospect to be given to the RTS project, it could be of interest to adopt a particular agile methodology that best suits the project's characteristics. The adaptation of the proposed practices to an agile methodology should not be incompatible, as previously mentioned.

8 References

- [1] J. Highsmith, *Agile Project Management: Creating Innovative Products*, Second Edi. Addison-Wesley Professional, 2009.
- [2] C. Larman, *Agile and Iterative Development: A Manager's Guide*. Addison Wesley, 2003.
- [3] I. C. Oliveira and J. P. S. Cunha, "Integration Services to Enable Regional Shared Electronic Health Records," in *User Centred Networked Health Care - Proceedings of MIE, Oslo, Norway*, 2011, pp. 310-314.
- [4] "Rede Telemática Saúde." [Online]. Available: <http://www.rtsaude.pt>. [Accessed: 13-May-2012].
- [5] Oxford University Press (Corporate Author), "New Oxford American Dictionary." .
- [6] W. Lewis, *Software Testing and Continuous Quality Improvement*, Third Edit. CRC Press, 2009.
- [7] C. Borrer, Ed., *The Certified Quality Engineer Handbook*, Third Edit. ASQ Quality Press, 2009.
- [8] G. Gordon Schulmeyer, Ed., *Handbook of Software Quality Assurance*, Fourth Edi. Artech House, 2008.
- [9] M. McDonald, M. Robert, and R. Smith, *The Practical Guide to Defect Prevention*. Microsoft Press, 2008.
- [10] D. Huizinga and A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management*. John Wiley & Sons, Inc, 2007.
- [11] B. Collins-sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion: For Subversion 1.5*, vol. 5. TBA, 2008.
- [12] J. Humble and D. Farley, *Continuous Delivery*. Addison-Wesley, 2011.
- [13] M. Moreira, *Adapting Configuration Management for Agile Teams: Balancing Sustainability and Speed*. John Wiley & Sons, Inc, 2009.
- [14] W. S. Humphrey, *Managing the Software Process*. Addison-Wesley Professional, 1989.
- [15] M. Andrews and J. A. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley Professional, 2006.
- [16] International Organization for Standardization, "ISO 9000 - Quality management." [Online]. Available: http://www.iso.org/iso/iso_catalogue/management_and_leadership_standards/quality_management.htm. [Accessed: 28-May-2012].
- [17] I. O. for Standardization, "ISO 15504 - Information technology: Process assessment." [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38932. [Accessed: 30-May-2012].
- [18] E. Whitworth, "Agile Experience: Communication and Collaboration in Agile Software Development Teams," Carleton University, 2006.
- [19] B. C. Ferreira Dias Alves, "Construção de módulos para o Ddesenvolvimento ágil de sites Web na Dreamlab," Universidade de Aveiro, 2010.
- [20] "Agile Alliance." [Online]. Available: <http://www.agilealliance.org>. [Accessed: 30-May-2012].
- [21] P. Kroll, P. Kruchten, and G. Booch, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley Professional, 2003.
- [22] "Scrum Alliance." [Online]. Available: <http://www.scrumalliance.org>. [Accessed: 30-May-2012].
- [23] J. Shore and Chromatic, *The Art of Agile Development*. O'Reilly Media, 2007.
- [24] A. Cockburn, *Agile Software Development: The Cooperative Game*, Second Edi. Addison Wesley Professional, 2006.
- [25] C. Larson and F. LaFasto, *Teamwork: What Must Go Right/What Can Go Wrong*. Sage Publications, 1989.
- [26] G. Chin, *Agile Project Management: How to Succeed in the Face of Changing Project Requirements*. AMACOM, 2004.
- [27] S. C. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.
- [28] P. Caldwell, *Code Leader: Using People, Tools, and Processes to Build Successful Software*. Wiley Publishing, Inc., 2008.
- [29] P. Aston, C. Fitzgerald, and V. Mitov, "The Grinder, a Java Load Testing Framework." [Online]. Available: <http://grinder.sourceforge.net/>. [Accessed: 10-Jun-2012].

- [30] B. Damian, “FWPTT - Web Load Testing Framework.” [Online]. Available: <http://fwptt.sourceforge.net/>. [Accessed: 10-Jun-2012].
- [31] C. Goldberg, “Multi-Mechanize - Performance Testing Framework.” [Online]. Available: <http://testutils.org/multi-mechanize/>. [Accessed: 10-Jun-2012].
- [32] The Apache Software Foundation, “Apache JMeter.” [Online]. Available: <http://jmeter.apache.org/>. [Accessed: 10-Jun-2012].
- [33] C. Goldberg, “Pylot - Web Performance Tool.” [Online]. Available: <http://www.pybot.org/>. [Accessed: 10-Jun-2012].
- [34] J. Fulmer, “Siege.” [Online]. Available: <http://www.joedog.org/siege-home/>. [Accessed: 10-Jun-2012].
- [35] A. Hunt and D. Thomas, *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.
- [36] T. Riley and A. Goucher, Eds., *Beautiful Testing: Leading Professionals Reveal How They Improve Software*. O’Reilly, 2010.
- [37] P. Tahchiev, V. Massol, and G. Gregory, *JUnit in Action*, Second Edi. Manning Publications, 2010.
- [38] P. M. Duvall, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley, 2007.
- [39] A. Malec, C. Pickett, F. Hugosson, and R. Lemmen, “Check: A unit testing framework for C.” .
- [40] “CMockery.” [Online]. Available: <http://code.google.com/p/cmocker/>. [Accessed: 10-Jun-2012].
- [41] R. S. Consulting, “CppUTest.” [Online]. Available: <http://www.cpputest.org/>. [Accessed: 10-Jun-2012].
- [42] B. Lepilleur, “CppUnit - C++ port of JUnit.” [Online]. Available: <http://sourceforge.net/projects/cppunit/>. [Accessed: 10-Jun-2012].
- [43] “googlemock - Google C++ Mocking Framework.” [Online]. Available: <http://code.google.com/p/googlemock/>. [Accessed: 10-Jun-2012].
- [44] Gargoyle Software Inc., “HtmlUnit.” [Online]. Available: <http://htmlunit.sourceforge.net/>. [Accessed: 10-Jun-2012].
- [45] “Selenium - Web Browser Automation.” [Online]. Available: <http://www.seleniumhq.org/>. [Accessed: 10-Jun-2012].
- [46] SmartBear Software., “soapUI.” [Online]. Available: <http://www.soapui.org/>. [Accessed: 10-Jun-2012].
- [47] “JUnit.” [Online]. Available: <http://www.junit.org/>. [Accessed: 10-Jun-2012].
- [48] “TestNG.” [Online]. Available: <http://www.testng.org/>. [Accessed: 10-Jun-2012].
- [49] “Mockito.” [Online]. Available: <http://code.google.com/p/mockito/>. [Accessed: 10-Jun-2012].
- [50] “JSUnit.” [Online]. Available: <http://www.jsunit.net/>. [Accessed: 10-Jun-2012].
- [51] The jQuery Project., “QUnit.” [Online]. Available: <http://docs.jquery.com/QUnit/>. [Accessed: 10-Jun-2012].
- [52] Pivotal Labs., “Jasmine.” [Online]. Available: <http://pivotal.github.com/jasmine/>. [Accessed: 10-Jun-2012].
- [53] S. Bergmann, “PHPUnit.” [Online]. Available: <https://github.com/sebastianbergmann/phpunit/>. [Accessed: 10-Jun-2012].
- [54] “NUnit.” [Online]. Available: <http://www.nunit.org/>. [Accessed: 10-Jun-2012].
- [55] Microsoft Corporation., “MSTest.” [Online]. Available: <http://www.microsoft.com/visualstudio/>. [Accessed: 10-Jun-2012].
- [56] K. Li and M. Wu, *Effective Software Test Automation: Developing an Automated Software Testing Tool*. Sybex, 2004.
- [57] RedHat, “Arquillian.” [Online]. Available: <http://www.jboss.org/arquillian.html>. [Accessed: 10-Jun-2012].
- [58] B. Swift, “GINT - Groovy Integration Test Framework.” [Online]. Available: <https://bobswift.atlassian.net/wiki/display/GINT>. [Accessed: 10-Jun-2012].
- [59] “Citrus.” [Online]. Available: <http://www.citrusframework.org/>. [Accessed: 10-Jun-2012].
- [60] A. Zeller, *Why Programs Fail: A Guide To Systematic Debugging*, Second Edi. Morgan Kaufmann, 2009.
- [61] A. Scott and Ž. Filipin, “Watir - Web Application Testing in Ruby.” .
- [62] I. Actimind, “actiWATE - Web application testing environment.” [Online]. Available: <http://www.actiwate.com/>. [Accessed: 07-Jul-2012].

- [63] "JUnit - Unit test framework for web pages." [Online]. Available: <http://code.google.com/p/junit>. [Accessed: 07-Jul-2012].
- [64] A. Oram and G. Wilson, *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, 2010.
- [65] The Apache Software Foundation, "Subversion." [Online]. Available: <http://subversion.apache.org>. [Accessed: 07-Jul-2012].
- [66] "Git." [Online]. Available: <http://www.git-scm.com>. [Accessed: 07-Jul-2012].
- [67] "CVS." [Online]. Available: <http://www.nongnu.org/cvs>. [Accessed: 07-Jul-2012].
- [68] Canonical Ltd., "Bazaar." [Online]. Available: <http://bazaar.canonical.com>. [Accessed: 07-Jul-2012].
- [69] "Mercurial." [Online]. Available: <http://mercurial.selenic.com>. [Accessed: 07-Jul-2012].
- [70] M. Mason, *Pragmatic Version Control using Subversion*. The Pragmatic Programmers, 2006.
- [71] T. Preston-Werner, "Semantic Versioning." [Online]. Available: <http://www.semver.org>. [Accessed: 25-Nov-2011].
- [72] "Apache Maven Project." [Online]. Available: <http://maven.apache.org>. [Accessed: 13-Sep-2011].
- [73] Sonatype, "The Benefits of a Repository Manager." 2011.
- [74] The Apache Software Foundation, "Ivy." [Online]. Available: <http://ant.apache.org/ivy>. [Accessed: 07-Jul-2012].
- [75] "Bugzilla." [Online]. Available: <http://www.bugzilla.org>. [Accessed: 07-Jul-2012].
- [76] Canonical Ltd., "Launchpad." [Online]. Available: <https://www.launchpad.net>. [Accessed: 07-Jul-2012].
- [77] "Trac - Integrated SCM & Project Management." [Online]. Available: <http://trac.edgewall.org>. [Accessed: 07-Jul-2012].
- [78] J.-P. Lang, "Redmine." [Online]. Available: <http://www.redmine.org>. [Accessed: 07-Jul-2012].
- [79] A. Mukherjee, *Challenges and Solution: A Web Based Issue Tracking System for detecting Bugs*. VDM Verlag Dr. Müller, 2010.
- [80] E. Dustin, *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Professional, 2002.
- [81] J. Cleland-Huang, O. Gotel, and A. Zisman, Eds., *Software and Systems Traceability*. 2012.
- [82] J. F. Smart, *Java Power Tools*, vol. 33, no. 1. O'Reilly, 2008.
- [83] B. Rex, *Critical Testing Processes: Plan, Prepare, Perform, Perfect*. Addison-Wesley Professional, 2003.
- [84] "FindBugs." [Online]. Available: <http://findbugs.sourceforge.net>. [Accessed: 07-Jul-2012].
- [85] InfoEther, "PMD." [Online]. Available: <http://pmd.sourceforge.net>. [Accessed: 07-Jul-2012].
- [86] "Checkstyle." [Online]. Available: <http://checkstyle.sourceforge.net>. [Accessed: 07-Jul-2012].
- [87] M. Doliner, "Cobertura." [Online]. Available: <http://cobertura.sourceforge.net>. [Accessed: 07-Jul-2012].
- [88] Oracle, "Hudson." [Online]. Available: <http://www.hudson-ci.org>. [Accessed: 07-Jul-2012].
- [89] "Jenkins." [Online]. Available: <http://www.jenkins-ci.org>. [Accessed: 07-Jul-2012].
- [90] "CruiseControl." [Online]. Available: <http://cruisecontrol.sourceforge.net>. [Accessed: 07-Jul-2012].
- [91] "Lunbuild." [Online]. Available: <http://lunbuild.javaforge.com>. [Accessed: 07-Jul-2012].
- [92] The Apache Software Foundation, "Apache Continuum." [Online]. Available: <http://continuum.apache.org>. [Accessed: 07-Jul-2012].
- [93] C. Hibbs, S. Jewett, and M. Sullivan, *The Art of Lean Software Development*. O'Reilly Media, 2009.
- [94] S. Guckenheimer and N. Loje, *Agile Software Engineering with Visual Studio: From Concept to Continuous Feedback*, Second Edi. Addison-Wesley Professional, 2011.
- [95] M. Clark, *Pragmatic Project Automation: How To Build, Deploy and Monitor Java Applications*. The Pragmatic Programmers, 2004.
- [96] T. O'Brian, J. Casey, B. Fox, B. Snyder, J. Van Zyl, and E. Redmond, *Maven: The Definitive Guide*. Sonatype inc., 2008.
- [97] S. D. Ritchie, *Pro .NET Best Practices*. Apress, 2011.
- [98] J. Weirich, "Rake - Ruby Make." [Online]. Available: <http://rake.rubyforge.org>. [Accessed: 07-Jul-2012].
- [99] The Apache Software Foundation, "Apache Ant." [Online]. Available: <http://ant.apache.org>. [Accessed: 07-Jul-2012].
- [100] S. Desikan and G. Ramesh, *Software Testing: Principles and Practices*. Addison-Wesley Professional, 2007.
- [101] J. F. Smart, *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

- [102] J. Ward and J. Peppard, *Strategic Planning for Information Systems*. John Wiley & Sons, Inc, 2002.
- [103] Instituto Nacional de Estatística, “Estudos sobre Estatísticas Estruturais das Empresas (2008),” 2010.
- [104] M. Riou, *Raven: Scripting Java Builds With Ruby*. firstPress, 2007.
- [105] J. Morris, *Practical Data Migration*. British Computer Society, 2006.
- [106] K. Schwaber, *Agile Project Management with Scrum*. Microsoft Press, 2004.
- [107] A. Kelly, *Changing Software Development: Learning to Be Agile*. John Wiley & Sons, Inc, 2008.
- [108] JFrog Ltd., “Artifactory.” [Online]. Available: http://www.jfrog.com/home/v_artifactory_opensource_overview. [Accessed: 07-Jul-2012].
- [109] Sonatype Inc., “Nexus.” [Online]. Available: <http://www.sonatype.org/nexus>. [Accessed: 07-Jul-2012].
- [110] The Apache Software Foundation, “Apache Archiva.” [Online]. Available: <http://archiva.apache.org>. [Accessed: 07-Jul-2012].
- [111] The Apache Software Foundation, “Maven Release Plugin.” [Online]. Available: <http://maven.apache.org/plugins/maven-release-plugin>. [Accessed: 07-Jul-2012].
- [112] “DbDeploy.” [Online]. Available: <http://www.dbdeploy.com>. [Accessed: 07-Jul-2012].
- [113] “migrate4j.” [Online]. Available: <http://migrate4j.sourceforge.net>. [Accessed: 07-Jul-2012].
- [114] “autopatch.” [Online]. Available: <https://github.com/tacitknowledge/autopatch>. [Accessed: 07-Jul-2012].
- [115] “Liquibase.” [Online]. Available: <http://www.liquibase.org>. [Accessed: 07-Jul-2012].
- [116] “Liquibase Maven Plugin.” [Online]. Available: <http://www.liquibase.org/manual/maven>. [Accessed: 07-Jul-2012].
- [117] “Cargo Maven Plugin.” [Online]. Available: <http://cargo.codehaus.org/Maven2+plugin>. [Accessed: 07-Jul-2012].
- [118] V. Massol, M. Wringe, and S. A. Tokmen, “Cargo.” [Online]. Available: <http://cargo.codehaus.org>. [Accessed: 07-Jul-2012].
- [119] M. Cohn, *Succeeding with Agile - Software development using Scrum*. Addison Wesley, 2010.
- [120] W. McIntyre, *Lean and Mean Process Improvement*. BookSurge Publishing, 2009.

9 Annexes

9.1 Annex A - Questionnaire form

Questionário de diagnóstico às práticas de construção de software nas pequenas e médias empresas portuguesas

0% 100%

I. Caracterização do Projeto

1. Qual a dimensão da equipa envolvida?

Considere a equipa com os vários perfis associados, com pelos menos 10% de tempo alocado ao projecto.

Escolha uma das seguintes respostas

- 1-3 pessoas
- 4-10 pessoas
- 11-25 pessoas
- 26-50 pessoas
- > 50 pessoas

2. A equipa envolvida no projecto está disposta de forma distribuída?

Os projectos de software podem ter elementos da equipa presentes em diferentes localizações físicas que comunicam usando plataformas apropriadas.

Escolha uma das seguintes respostas

- Não, está toda disposta no mesmo local
- Sim, está disposta em diferentes localidades a nível nacional
- Sim, está disposta em diferentes pontos, com diferentes fusos horários.

3. Que categoria ou categorias descrevem melhor o projecto?

Selecione todas as que se apliquem

- | | | | |
|--|--|---|--|
| <input type="checkbox"/> Desktop (Standalone) | <input type="checkbox"/> Enterprise (Multi-tier) | <input type="checkbox"/> Embedded | <input type="checkbox"/> Outra(s), por favor especifique: <input type="text"/> |
| <input type="checkbox"/> Mobile (smartphone/devices) | <input type="checkbox"/> Multi-module | <input type="checkbox"/> Critical | |
| <input type="checkbox"/> Web | <input type="checkbox"/> Enterprise | <input type="checkbox"/> System (OS, drivers, etc...) | |

4. Indique as linguagens de programação usadas:

Selecione todas as que se apliquem

- | | | | |
|-----------------------------------|--------------------------------------|---------------------------------|--|
| <input type="checkbox"/> Assembly | <input type="checkbox"/> COBOL | <input type="checkbox"/> Perl | <input type="checkbox"/> Tcl |
| <input type="checkbox"/> C | <input type="checkbox"/> Java | <input type="checkbox"/> PHP | <input type="checkbox"/> FoxPro |
| <input type="checkbox"/> C++ | <input type="checkbox"/> JavaScript | <input type="checkbox"/> Python | <input type="checkbox"/> Outra(s), por favor especifique: <input type="text"/> |
| <input type="checkbox"/> C# | <input type="checkbox"/> Lisp | <input type="checkbox"/> Ruby | |
| <input type="checkbox"/> Caml | <input type="checkbox"/> Objective-C | <input type="checkbox"/> Scala | |

5. Indique há quanto tempo decorre o projecto:

Escolha uma das seguintes respostas

- < 6 meses
- 6 meses – 1 ano
- 1 ano – 3 anos
- 3 anos – 6 anos
- > 6 anos

6. Esta solução encontra-se em produção (parcialmente ou na totalidade) e sujeita a manutenção?

Escolha uma das seguintes respostas

- Não está instalada em produção.
- Em produção < 6 meses
- Em produção 6 meses – 1 ano
- Em produção 1 ano – 3 anos
- Em produção > 3 anos

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

eLearning
universidade de aveiro

0% 100%

II. Controlo de versões e gestão de projectos

1. O projecto usa controlo de versões do código?

O código fonte é consolidado num servidor; é possível desenvolver sobre cópias locais e, depois, integrá-las no servidor.

Sim Não

a. Se respondeu sim, indique qual o software de controlo de versões que o projecto usa:

Qual o pacote de software ou serviço utilizado que permite controlar as mudanças dos ficheiros e resolver os potenciais conflitos entre edições concorrentes.

Escolha uma das seguintes respostas

- Bazaar
 CVS
 Git
 Mercurial
 Subversion (SVN)
 Synergy
 Team Foundation Server
 Outro(s), por favor especifique:

2. É possível reproduzir versões específicas do projecto?

É possível restaurar o estado do projecto referido a um determinado momento anterior (inclui os ficheiros de código e de configurações), através de comandos/opções?

Sim Não

a. É possível incrementar uma versão específica?

É possível acrescentar modificações aos ficheiros do projecto referidos a uma versão, num determinado momento anterior, sem comprometer outras versões do projecto de desenvolvimento?

Sim Não

b. É possível obter informações das mudanças de uma versão específica?

É possível determinar quais as mudanças do projecto entre instantes do tempo (identificar as diferenças no código e nas configurações)?

Sim Não

c. Se o projeto usa bases de dados, a sua estrutura é sujeita a controlo de mudanças?

As mudanças na estrutura das bases de dados são controladas de modo a que estejam coerentes com o restante código ao longo das versões do projeto?

Sim Não

3. Indique as formas como são controladas as versões das dependências do projeto:

Cada versão do projeto foi preparada para funcionar com versões específicas de módulos complementares e bibliotecas (internos ou externos). Como é que é feito o controlo das versões dos módulos/bibliotecas em relação à evolução das versões do projeto?

Escolha uma das seguintes respostas

- Sem controlo
- Apenas com software de controlo de versões (de ficheiros).
- Com software habilitado para controlo de dependências (e gestão de repositório de dependências).

a. Se escolheu a última opção, indique qual o software de gestão de dependências que o projeto usa:

Se respondeu escolheu a última opção, indique qual a aplicação que o projeto usa para gerir as versões das dependências associadas às versões do projeto:

Selecione todas as que se apliquem

- PAR (Perl)
- PEAR (PHP)
- RubyGems
- Maven Repositories
- Ivy
- Outro(s), por favor especifique:

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

eLearning
universidade de aveiro

0% 100%

III. Integração contínua

1. O projecto usa integração contínua automatizada?

O projeto utiliza a prática de integração de código recentemente desenvolvido, incluindo procedimentos frequentes de consolidação dos incrementos na solução global e verificação automatizada de qualidade do estado assim obtido?

Sim Não

a. A frequência da integração contínua:

Indique a frequência com que o novo código é consolidado automaticamente na solução global:

Escolha uma das seguintes respostas

- Sempre que possível (sem frequência imposta)
 Diariamente
 Semanalmente
 Outra. Qual?

b. As fases abrangidas pela integração contínua:

Indique que fases da construção que são executadas no processo de integração automática de novo código na solução global:

Selecione todas as que se apliquem

- Integração de código fonte e garantia de boa compilação.
 Execução de testes automáticos de verificação.
 Obtenção automática de métricas de qualidade do código (quality review).
 (Re-)Instalação da solução (Deployment) em ambiente de testes/desenvolvimento.

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

0% 100%

IV. Testes de Software

1. O projecto é sujeito a testes funcionais por uma equipa dedicada?

O perfil de "tester" existe na equipa e o projecto é sujeito a testes para verificar que está de acordo com as especificações funcionais.

Sim Não

2. Indique os tipos de testes que o projecto está sujeito:

Especificar quais os testes que estão incluídos na prática de construção; Destes, indicar também os que estão abrangidos pela integração contínua (I.C.)

		I.C.
Testes unitários (1)	<input type="checkbox"/>	<input type="checkbox"/>
Testes de integração (2)	<input type="checkbox"/>	<input type="checkbox"/>
Testes funcionais (3)	<input type="checkbox"/>	<input type="checkbox"/>
Testes de desempenho (4)	<input type="checkbox"/>	<input type="checkbox"/>
Testes de regressão (5)	<input type="checkbox"/>	<input type="checkbox"/>

- ?** 1) Testes unitários: testes individuais a unidades de código, normalmente para verificar cada método ou função, escritos pelo próprio programador em tempo de desenvolvimento.
2) Testes de integração: testes com o objectivo de determinar se a inserção de novas funcionalidades numa solução abrangente foi bem sucedida.
3) Testes funcionais: Testes à funcionalidade do projecto de modo a verificar se efectua as tarefas pretendidas.
4) Testes de desempenho: Testes sobre a aplicação estudando o seu comportamento simulando diferentes cenários e condições, com objectivo de identificar possíveis deficiências internas (de implementação).
5) Testes de regressão: Testes que permitem verificar se a mudança do código gera novos erros em funcionalidades já existentes.

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

Questionário de diagnóstico às práticas de construção de software nas pequenas e médias empresas portuguesas

0% 100%

V. Práticas de Qualidade

1. No projeto foi utilizado um guia com orientações de boas práticas, convenções ou regras de produção de código?

Existe algum documento com linhas orientadoras para os programadores com o objectivo de melhorar a qualidade do código e disseminar práticas comuns para tornar o desenvolvimento em equipa mais fácil e documentado?

Sim Não

2. No projecto foi utilizada alguma metodologia/processo de desenvolvimento de software descrito na literatura?

O projecto segue algum processo específico de desenvolvimento de software com o objectivo de melhorar a eficácia, rapidez e qualidade da solução produzida?

Escolha uma das seguintes respostas

Sim
 Sim, mas substancialmente adaptado pela empresa
 Não

a. Se respondeu sim, indique qual/quais:

Seleccione todas as que se apliquem

- Extreme Programming (1)
- Feature Driven Development (2)
- Rapid Application Development (3)
- Scrum (4)
- Test Driven Development (5)
- Unified Process ou derivados (6)
- Velocity (7)
- Waterfall (8)
- Outro(s), por favor especifique:

- ?** (1) Extreme Programming: Este tipo de metodologia está associada a ciclos de desenvolvimento muito curtos e permite integrar novos requisitos frequentemente.
- (2) Feature Driven Development: Processo de desenvolvimento adequado a equipas de maior dimensão e menos especializadas dando ênfase à monitorização do processo regularmente.
- (3) Rapid Application Development: Desenvolvimento com planeamento mínimo mas com recurso extenso à prototipagem. Permite maior rapidez de desenvolvimento.
- (4) Scrum: Metodologia de desenvolvimento e gestão de projectos que tem como princípio chave a aceitação da mudança constante e tornar eficaz a resposta a essas mudanças.
- (5) Test Driven Development: Processo de desenvolvimento que consiste em pequenos ciclos de desenvolvimentos iniciados com a implementação de test cases.
- (6) Unified Process ou derivados: Metodologia iterativa, incremental e customizável de desenvolvimento de software. Centrada na arquitectura e gestão de riscos.
- (7) Velocity: Método que permite a medição do valor entregue pelas equipas de software à solução.

(8) Waterfall: Modelo de desenvolvimento de software sequencial, sem iterações ou incrementos.

3. O projeto é sujeito a revisões de código regulares?

O projecto é sujeito a inspeções de código (code review) regulares com o objectivo de detectar erros de semântica e de sintaxe e padrões de programação sub-ótimos?

Sim Não

a. Se respondeu sim, indique o tipo de inspecção a que o código está sujeito:

Especificar quais as inspeções abrangidas pela integração contínua (I.C.):

	Revisão de código pontual	Revisão de código inserida na I.C.
Humanas (1)	<input type="checkbox"/>	<input type="checkbox"/>
Análise estática automática (2)	<input type="checkbox"/>	<input type="checkbox"/>
Testes de cobertura automáticos (3)	<input type="checkbox"/>	<input type="checkbox"/>
Código duplicado (4)	<input type="checkbox"/>	<input type="checkbox"/>

? (1) Inspeções de código humanas: Inspeções ao código levadas a cabo por elementos da equipa de desenvolvimento.

(2) Análise estática: Inspeções levadas a cabo por software que analisa o código por padrões de erros comuns, más práticas no código, etc.

(3) Cobertura de testes: Análise efectuada por software que determina qual a quantidade de código que é sujeita a testes unitários.

(4) Código duplicado: Análise ao código por software que permite encontrar blocos de código duplicado.

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

eLearning
universidade de aveiro

0% 100%

VI. Controlo de defeitos

1. Qual o sistema de controlo de defeitos (e tarefas) em utilização no projecto?

Indique a aplicação que permite registar e gerir defeitos identificados no código (bugs) e, eventualmente, novos requisitos.

Selecione todas as que se apliquem

- Bugzilla
- codeBeamer
- GitHub
- Google Code Hosting
- HP Quality Center
- JIRA
- Lauchpad
- Redmine
- Starteam
- Team Foundation
- Trac
- Não é usado
- Outro, por favor especifique:

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

Questionário de diagnóstico às práticas de construção de software nas pequenas e médias empresas portuguesas

0% 100%

VII. Conclusão

1. Impacto dos diferentes aspectos no sucesso da construção do projeto de software:

Por favor, indique o grau de importância dos diferentes processos da construção da solução no que respeita à qualidade e sucesso do projeto:

	Nenhum			Muito	
Processo de Build	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Controlo de versões	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Integração contínua	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Testes unitários	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Testes integração	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Testes funcionais	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Testes de performance	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Métricas de qualidade	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Gestão de tarefas	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Práticas e metodologias	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

2. Das várias ferramentas em uso no seu projecto de desenvolvimento, qual a que elege como mais valiosa no ciclo de construção de software?

<< Anterior

Seguinte >>

Sair e limpar inquérito

Continuar mais tarde

eLearning
universidade de aveiro

0% 100%

VIII: Perfil e contacto

1. Qual a dimensão da empresa (número de colaboradores)?

Neste campo só se aceitam números

2. Qual o peso dos projectos de desenvolvimento e integração de software (em percentagem) na atividade da empresa (estimativa)?

Neste campo só se aceitam números

3. Deseja receber os resultados consolidados deste questionário, sob a forma de um Sumário Executivo?

Sim Não

4. Deseja receber uma cópia digital da dissertação de Mestrado Integrado referida?

Sim Não

5. Deseja que o nome da empresa seja publicada para efeitos de agradecimento no texto da dissertação?

Sim Não

a. Se respondeu sim à alínea 3), 4) ou 5), por favor indique:

Nome:

Cargo:

Empresa:

Email:

6) Se desejar, deixe alguns comentários ao questionário:

9.2 Annex B – Public appraisal of contributing respondents

In the setup of the survey, we have asked respondents whether they would agree us to include the company name in a public appraisal of their collaboration. The following is the list of the companies that agreed to it. We are thankful for their contribution, as for the others that preferred not to be disclosure.

Company name	City
9Tree	Cartaxo
Be.Ubi	Aveiro
Bullet Solutions	Oporto
Conexus World	Espinho
Digidelta Software, Lda.	Torreres Vedras
Edgelabs	Aveiro
FeedZai	Coimbra
Grupo@work	Matosinhos
Humansoft	Leiria
Log	Lisbon
Moving2u, Lda.	Coimbra
NDrive	Oporto
PHC Software, S.A.	Oeiras
Pictonio, Lda.	Aveiro
Pontual - Soluções Tecnológicas	Santa Maria da Feira
S2L - Software e Sistemas, Lda.	Viseu
SAGE Portugal, S.A.	Oporto
Seegno	Braga
Segilac - Sistemas de Gestão de Saúde, Lda.	Vila Nova de Gaia
Sensebloom	Coimbra
Sonatrix, Lda.	Covilhã
Sysactum - Sistemas e Software, S.A.	Oporto
Sysnovare	Oporto
Tecla Infinita	Fundão
Vendder	Oporto
White Road Software	Lisbon

9.3 Annex C - Best practices document

Grupo de Sistemas e Informação na área da Saúde
RTS – Rede Telemática de Saúde

TECHNICAL REPORT

Coding Style and Developer Best Practices

Version 0.9, not revised

Contributors: Ilídio Oliveira, Bruno Palos

Abstract: Presents the normative coding style and developing practices to be used in the RTS Project.

Circulation: RTS Project developers and the SIAS Group.

Contents

1	Introduction	4
1.1	Why do we need this guide?	4
1.2	Background concepts and values	4
1.2.1	The development scenario	4
1.2.2	Terms definition	5
2	Developer use cases in practice.....	7
2.1	Set-up Developer Environment	7
2.2	Checkout a project and start developing.....	7
2.3	Contribute with new code	7
2.4	Adding new files to the Project.....	8
2.5	Refactor classes and packages.....	8
2.6	Rollback changes from previous releases	8
2.7	Start a new version (and about builds?)	8
2.8	Generate a corrective patch to a existent version	9
2.9	Keep a development branch synchronized with the main development line (trunk)	9
2.10	Updating the databases schemasError! Bookmark not defined.	
3	Shared software construction practices	10
3.1	Java coding style	10
3.1.1	What conventions to use?	10
3.1.2	Javadoc	10
3.1.3	Check style tools	12
3.1.4	Bug patterns tools.....	12
3.2	Version control good practices	13
3.3	Best practices with the Issues Tracking System (ITS) ..	14
3.4	Continuous Integration good practices	14
4	Using construction processes and tools.....	15
4.1	Java projects with Maven	15
4.1.1	Maven releases and snapshots.....	15
4.1.2	Maven commands	15
4.1.3	Run the unit tests	16
4.1.4	Verify if there are no SNAPSHOT dependencies.....	16
4.1.5	Changing the version of the module	16
4.1.6	Generating a release	17
4.1.7	Remote repository manager	17

4.2	Continuous Integration Server	18
4.3	Your project's code repository (SVN)	20
4.3.1	Configuring a graphical SVN client.....	20
4.3.2	How repository is organized	21
4.3.3	Operations using command line.....	21
4.3.4	View SVN log	22
4.3.5	Update the working copy with the latest's developments.....	23
4.3.6	Commit the changes to repository	23
4.3.7	Create new versioned file.....	24
4.3.8	Move/remove versioned files.....	24
4.3.9	View local modifications	25
4.3.10	Revert local modifications	25
4.3.11	View file modifications.....	26
4.3.12	Compare changes from different revisions	26
4.3.13	Conflict after an update.....	27
4.4	Versioning	28
4.4.1	Overview	28
4.4.2	Version-numbering strategy.....	28
4.4.3	Releasing a version	29
5	Programming patterns	35
5.1	Generic programming techniques	35
5.2	Patterns for logging	35
5.2.1	Introduction.....	35
5.2.2	Logging efficiency	35
5.2.3	Guarded logging.....	37
5.2.4	Logging framework configuration	39
5.2.5	Guidelines for using logging levels	40
5.3	Patterns for Exceptions handling	41
5.3.1	Introduction.....	41
5.3.2	Common errors on exceptions	41
5.3.3	Basic principles of exception handling	45
5.3.4	Custom exception classes.....	45
5.3.5	The use of unchecked exceptions:	46
5.3.6	Exceptions on multi-module systems	46
6	References	47

1 Introduction

1.1 Why do we need this guide?

Because:

- We have different brains, values and attitudes (difference is good).
- We value results (↑productivity).
- We code better when coding for the team (↑quality).
- Shared rules make developing easier (↑mental sanity).
- I will benefit from shared practices later on, even with respect to my own work.

1.2 Background concepts and values

1.2.1 The development scenario

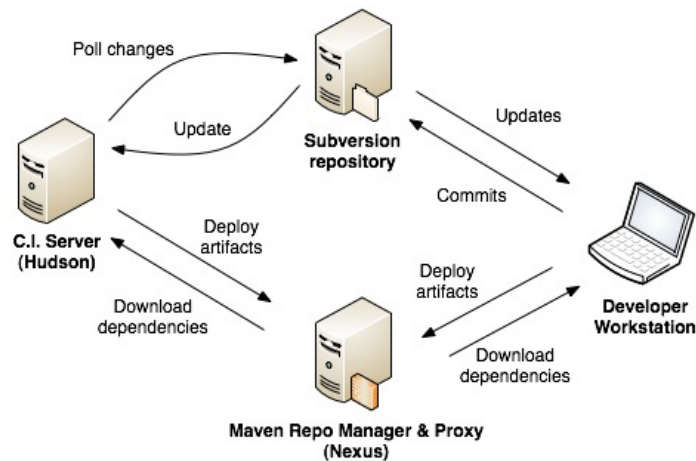


Figure 1 - RTS' development scenario.

Figure 1 shows how the RTS's development pipeline works. This process starts when the developer do a commit on Subversion, then Hudson, after a periodical update operation, gets the newly code developments, which will trigger a new build. If everything goes well on this build, Hudson will generate all the project artifacts and deploy them to Nexus. Hudson will automatically deploy the artifacts that are still under development (snapshots), making them immediately accessible for internal sharing. After each successful automatic build, Hudson will deploy all the recently generated artifacts on Nexus's snapshots repository.

1.2.2 Terms definition

- Build

The build process is a series of steps that transforms the various project components in an application ready to be deployed. In other words, a build follows a sequence of statements previously prepared in a specific order resulting in packaged files, prepared to be installed into production or into other types of runtime environments.

- Branch

Branches are development lines that will evolve in parallel with each other and with trunk (the main development line). They are mapped as directories on the code repository (Subversion) to separate developments of different application's versions. Branches are used to maintain releases and to stabilize release candidates (RC), just before a final release.

- Code repository

Code repositories are typically Version Control Systems (VCS). Such tools offer teams the possibility to keep versions of the project files and people to collaborate without interfering in each other's work. These systems allow to go back in history's files or to perpetuate releases. One of the most useful features in a VCS is the ability to reproduce earlier states of software projects.

- Continuous Integration

Continuous integration (CI) is a technique that makes the development process smoother, more predictable and less risky, even on advanced stages in the lifecycle of software applications. Additionally, bugs can be traced soon after they are introduced into the project code and, after notification, developers can solve them quickly. This avoids the period called "Integration Hell", which is used to integrate software components in the final stage of development.

- Database migrations

Typically, managing code files' changes is based on using management tools such as Subversion. Such tools are not enough when evolving databases because the information contained herein differs on every runtime scenario and just replacing the respective schemas is not enough. In order to upgrade the DB consistently with the application's version being installed requires migrating only new changes matching the new version, such as inserting new tables, or adding new constraints. Changes to be migrated must be properly marked so that the DB integrity not becomes corrupted.

- Dependencies management

The continuous progress of the project frequently implies changing the way the various modules of the system components communicate. Such modules or APIs may be internal or external and are also coupled with different releases as they are subjected to evolution as most software systems. The connection between the project and its dependencies shall be carried out

Coding Style and Developer Best Practices

through their compatibility, therefore different project releases may rely on different versions of external libraries without ensuring backward compatibility.

- Issue

Are defects, tasks or requirements that allow development teams to organize work. They are characterized by severity, target version and assignee. Requirements represent new application functionalities and may be imposed by customers or internally suggested by other project stakeholders. Defects are unforeseen problems or bugs found by testing teams, customers or even by the development team. Tasks represent other generic activities to be performed by the project's staff.

- Releases

Releases are closed versions of the application. A release is performed after the end of development phase. There are three kinds of releases: major, minor and patches. Major releases are versions who breakup compatibility with previous versions; Minor releases are evolutions of the application with new features but without compatibility breakup; patches are releases with the objective of fixing bugs.

- Trunk

Opposed to branches, the trunk is the main development line. All branches born from trunk and contains the most updated version of the project. For continuous integration purposes, all feature development should be committed to the trunk.

- Snapshot version

Snapshot versions are versions of the application, which are still under development. All releases are preceded by a snapshot version. This term were introduced with the adoption of Maven versions.

2 Developer use cases in practice

2.1 Set-up Developer Environment

- Synchronize your workstation's time with an NTP server (ex: ntp.ua.pt);
- Use an IDE and O.S. with full UTF-8 support;
- Install the necessary plug-ins on your IDE:
 - Install a SVN integration plug-in;
 - Install a plug-in for the IST (Issue Tracking System);
 - Install all necessary plug-ins for code static analysis. The FindBugs plug-in is the most important;
- You can complement your IDE SVN plug-in with a standalone SVN client. You can use the command-line client or a graphical tool;
- Install the correct Java SDK version and the selected Application Server for local development;
- Install Maven version 2 in your computer for building, dependency management, Unit testing, etc.:
 - After install, update the `~/.m2/settings.xml` file with the correct parameters.

2.2 Checkout a project and start developing

Method 1:

- Choose a location in your workstation to place the new working copy of the project;
- Checkout the project from the SVN repository;
- Using your IDE, create a new project importing the new created working copy;
- Start developing.

Method 2:

- Using your IDE, create a new project from SVN repository. The IDE will checkout the project for you;
- Start developing.

2.3 Contribute with new code

- Update your working copy (go to 4.3.5 section);
- Develop code;
- Check for SVN repository changes (go to 4.3.4 section);
 - If there are new SVN transactions (go to 4.3.4), update your working copy;

Coding Style and Developer Best Practices

- If the code is not compiling, correct the errors;
- If the static analysis detects bugs, correct the code;
- If the Unit testing fails (go to 4.1.3 section), correct the code;
- After the code is compiling, the detected bugs fixed (if possible) and the all the Unit tests passing, commit the new changes to the SVN repository (go to 4.3.6 section);
- If applicable, update the status of the associated issue in the project's ITS;
- If the Continuous Integration server detects problems, stop immediately what you're doing and fix those problems.

2.4 Adding new files to the Project

- Update your working copy (go to 4.3.5 section);
- Create the new file (go to 4.3.7);
- Add the file to the SVN working copy (go to 4.3.7 section);
- Go to the "Contribute with new code" use-case.

2.5 Refactor classes and packages

- Update your working copy (go to 4.3.5 section);
- Refactor using IDE tools (go to 4.3.8);
- Confirm the SVN operations done by the IDE;
 - If the new files weren't added to the working copy, add the files (go to 4.3.7 section);
 - If the removed/moved files weren't removed from the working copy, remove or move the files (go to 4.3.8 section);
- Go to the "Contribute with new code" use-case.

2.6 Rollback changes from previous releases

- Identify the SVN revision to rollback (go to 4.3.4);
- If the rollback applies only to one file, identify it;
- SVN merge the change of the specific revision and file (if applicable);
- Verify the changes affected by the merge;
- Go to the "Contribute with new code" use-case.

2.7 Start a new version (and about builds?)

- Verify the issues in the ITS targeted to the new version;
 - If the issues are not closed, move them to a future version;
- Choose the target build or SVN revision (?) that complies with the version identified in the ITS;
- If the revision is not the HEAD of the correspondent SVN branch, create a new branch for this version;
- Verify that are no dependencies with SNAPSHOT version;

Coding Style and Developer Best Practices

- Change the SNAPSHOT version of the Project's POMs to its corresponding release version;
- Commit the changes to the corresponding branch (go to 4.3.6 section);
- Tag the last revision with the desired version;
- If applicable, increment the POMs version in the correct branch to the new development version (SNAPSHOT) and commit the changes.

2.8 Generate a corrective patch to a existent version

- Create a new branch from the desired tag, if not already created;
- Change the release version of the Project POMs to the corrective SNAPSHOT version and commit the changes, if applicable;
- Go to "Contribute with new code" use-case;
- Change the SNAPSHOT version to its correspondent release version;
- Commit the changes (go to 4.3.6 section);
- Tag the files with the corrective version.

2.9 Keep a development branch synchronized with the main development line (trunk)

- Make sure your working copy has no local modifications (go to 4.3.9);
- From the development branch, merge the trunk;
 - If resulted in conflict, resolve the problems;
 - If the conflicts are impossible to resolve, revert the code and try to understand what seems to be the problem with your co-workers;
- Verify if the merge didn't break the code;
- If unit testing fails, fix the problems;
- After every thing is ok, commit the merge result;
- Repeat the previous process frequently when developing to the branch;
- If all developments to the branch are finished, switch to the trunk and merge with reintegrate option;
- Verify if the merge didn't break the code;
- Run all necessary unit tests;
- Commit the reintegration (go to 4.3.6 section);
- Delete the development branch.

3 Shared software construction practices

3.1 Java coding style

3.1.1 What conventions to use?

Apply the following conventions, in this order:

1. The following tried-and-true conventions:
 - a. Always specify UTF-8 for encoding (unless otherwise required).
 - b. Preferred indentation is 8 spaces (yes, eight).
 - c. Use an IDE with code formatting capabilities and apply them.
2. Always use English names for the programming entities: classes, methods, fields, etc.;
3. Read and apply the guidelines from [Code Conventions for the Java Programming Language](#), especially the Naming Conventions chapter. (“Attempt to make your names so clear that comments are (almost) unnecessary”).

Complementary, apply also the following best practices;

4. Try to match the style and naming that you can observe in Java standard API.
5. Use the “get”, “set” and “is” naming convention for methods that change and read private fields.
6. Avoid implementing long methods; usually it signals a design problem.
7. Try to maintain your fields as private as possible, specially with multithreading;
8. Don’t hard-code numbers in you implementation. Don’t hard-code any literals.
9. Remember that other people will read the code. And you, some weeks later.

3.1.2 Javadoc

You must use Javadoc syntax to produce documentation. You are expected to include meaningful comments in the code too. Note that:

- Documentation-style is for API, pre and post conditions reference.
- Comments are for other programmers to understand your rationale and options when writing the code.

At the very least, you need to provide a class purpose description in Javadoc. As a standard practice, also document public methods.

Coding Style and Developer Best Practices

Entity level:	Javadoc to provide:
Source file header (not in javadoc)	Full project name. Copyright notice. Optional: author name.
Class	Purpose description. Optional: author name (@author tag). Optional: version tag. Optional: link to related files/classes (@see tag).
Method (public):	Purpose description. Pre-conditions (if applicable). Parameters (if applicable). Description of return (if applicable).

```

/*
 * RTS - Rede Telematica da Saude
 * Copyright (c) 2005-2011 IEETA - SIAS
 *
 * This software is the confidential and proprietary information of IEETA
 * You shall not disclose such Confidential Information and shall use it
 * only in accordance with the terms agreed with IEETA.
 */

/**
 * Core scheduler to initiate the distributed integration processes.
 * @author IOliveira
 * @see org.rtsaude.foundation.rim.Episode;
 */

```

Listing 1: Sample header (to be used as a template) and class javadoc.

Be also aware that Java comments can be used to tag unfinished work and problems (Table 1). Modern IDEs will pick up these tags and call the programmer attention.

Tag	Use to:
// FIXME Need to control limits	Signal a known problem in the code that prevents it to work as expected.
// TODD Implement the iterator	Signal some task, usually new code.
// XXX Unlikely to support concurrency	Signal code portions that work, but the implementation is poor and needs to be revised.

Table 1: Tags embedded in comments to signal issues requiring future attention.

3.1.3 Check style tools

Checkstyle tools allow developers to maintain the code with a specific style, as defined on code conventions. There are several ways to check and maintain code conventions while developing code:

- Using IDE' code formatters;
- Using automatic static analysis tools;
- Using plugins on IDEs to warn the developer when the conventions are not being respected.

Figure 2 shows the Checkstyle plugin embedded on Java code editor from the Eclipse IDE. This feature enables real-time style checking while developing code. Figure 3 shows the code style formatter configuration for Eclipse. This allows to auto-format source-code files with the desired code conventions and style. Other IDEs, like Netbeans or IntelliJ IDEA, can also auto-format code files.

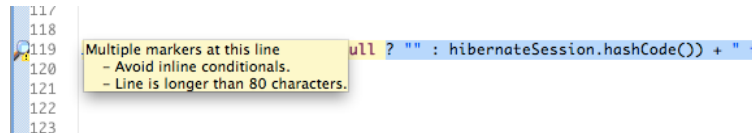


Figure 2 - Checkstyle plugin on Eclipse.

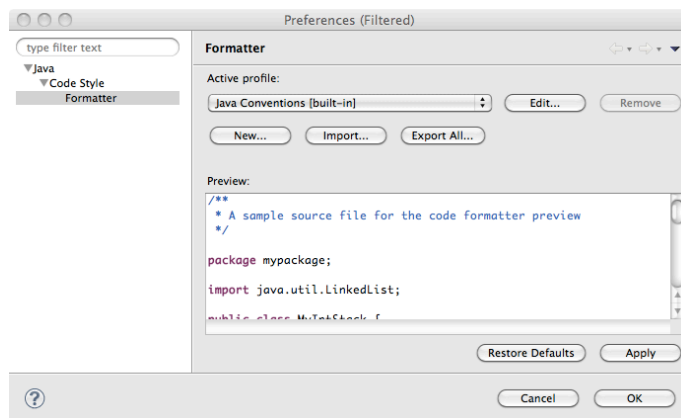


Figure 3 - Code style formatter configuration on Eclipse.

3.1.4 Bug patterns tools

There are several static analysis tools to find bug patterns on source-code files. One of the most popular is FindBugs. This tool can be used integrated on IDE code editors, on Continuous Integration servers, or even standalone. On the RTS project, every developer should install the FindBugs plugin for the IDE installed on his workstation (there are plugins for the most popular IDEs, like Netbeans, IntelliJ IDEA and Eclipse). Figure 4 shows real-time static analysis using FindBugs on Eclipse.

Coding Style and Developer Best Practices

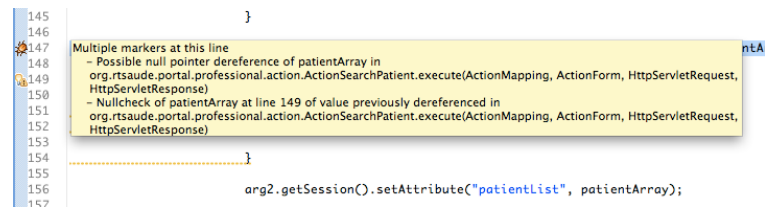


Figure 4 - FindBugs plugin installed on Eclipse IDE.

Other useful static analysis tool is PMD. It finds bug patterns similarly to FindBugs, but it has the ability to search for suspect copy-pasted code blocks. Figure 5 shows the found suspect copy-paste using the PMD plugin for Eclipse. PMD plugins for other IDEs are also available.

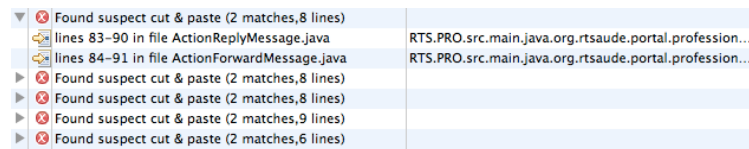


Figure 5 - Using PMD to find suspect copy-pasted code.

3.2 Version control good practices

1. Before committing, be sure to check for possible changes in the repository! (Usually: do an update!)
2. When you commit a change to the repository, make sure your change reflects a single propose (Ex: Fixing a bug, adding a new feature, etc);
3. If possible, try to create changesets linked to the issue tracker. Use the issue ID in the commit message;
4. Commit to trunk only tested and compiling code;
5. After merging, run the unit tests to ensure that the merge was successful;
6. After creating a tag, don't commit to it any more. Visualize the tag as read-only. If you need to resolve an issue in that specific version, create a branch from that tag and commit the changes to it;
7. If you are using a branch for version maintaining purposes, when you commit a fix to this branch and this fix is not resolved in the trunk, merge it to the trunk after you finished it;
8. Try not to merge a large quantity of changes between trunk and the branches. Use atomic commits;
9. Make, at least, one commit a day with all your day's work.

3.3 Best practices with the Issues Tracking System (ITS)

1. You must use an ITS in a daily basis. In RTS, we're using Redmine.
2. Always reference an ITS issue on commits to ensure traceability;
3. Resolve issues by priority;
4. Use a tool to trace the time spent in every task and register it in ITS. This helps the team to estimate the time spent in future issues and not to justify your work;
5. Use the ITS as a central repository of information for the project. If you find a solution to a specific and recurrent problem, use it to spread the information thought the team;
6. Document your work when working on the issue. Remember that **you will probably work on it again**;
7. Try to use the sub-tasking capabilities of the ITS to maintain an atomic structure of the issue. Remember the divide and conquer concept.

3.4 Continuous Integration good practices

1. Don't check in on a broken build;
2. Always run all commit tests locally before committing;
3. Commit your changes frequently (at least once a day);
4. Never go home with changes to commit;
5. Never go home on a broken build;
6. Always be prepared to revert the previous revision;
7. Take responsibility for all breakages that result from your changes;
8. Fix broken builds immediately.

4 Using construction processes and tools

4.1 Java projects with Maven

4.1.1 Maven releases and snapshots

Maven uses releases and snapshots to distinguish development versions from released versions of an artifact.

This is specified in the POM. If the version 1.2.1-SNAPSHOT is declared in the pom.xml file, the 1.2.1 version of the module is in development. By deploying the artifacts as a snapshot to the snapshots repository it is guaranteed that the last version of the development version is accessible to the rest of the development team and to yourself.

Ideally, a non-snapshot version shouldn't be used to make developments. A typical use-case to correct a released version is to create a branch from the tag directory and then change the version of the POMs to snapshot.

In the project's repository, an artifact in the releases repository should be considered has a final version.

4.1.2 Maven commands

Use Maven commands in the folder where a pom.xml file is present. Typically, a pom.xml file is placed in the root file structure of a project module.

Maven Command	Purpose
<i>\$mvn compile</i>	To compile. The first time you execute this command, Maven will download the necessary dependencies.
<i>\$mvn test</i>	To compile and run the test sources.
<i>\$mvn test compile</i>	To compile the test sources, but not run them.
<i>\$mvn clean</i>	To remove the target directory (compiled files).
<i>\$mvn package</i>	To generate java archive files.
<i>\$mvn install</i>	To install the generated artefacts in your local repository.
<i>\$mvn deploy</i>	To install the generated artefacts in your project's repository:
<i>\$mvn clean install</i>	You can use multiple goals in one command. For example: This will clean the target directory, compile the sources, package and install the artefacts in your local repository.

Table 2: Useful Maven commands

4.1.3 Run the unit tests

Maven runs the unit tests of a suitable framework (JUnit, TestNG, etc.) present in the project's default directory: `src/test/java`.

All the unit tests of the project can be run at once or just the ones of individual modules by executing the command in the same directory of the module POM file.

The unit test goal is integrated with Maven build lifecycle but you can run that goal alone, ignoring other lifecycle goals. To run the tests use the command `'mvn test'` in the same directory of the desired POM file. All the tests must pass otherwise the build fails.

To compile the tests without run them use the command `'mvn test-compile'`. Or to skip the tests on a regular build cycle use the command `'mvn install -Dmaven.test.skip=true'`, for example. Skip the tests only in punctual situations because the objective is to produce high quality software. The C.I. server never skips the tests and fails the builds with failed unit tests.

4.1.4 Verify if there are no SNAPSHOT dependencies

The Maven Dependency Plugin can be used to list a dependency tree of a project or module.

To list the SNAPSHOT dependencies of a project or module use the following command: `'mvn dependency:tree -Dincludes=::*-SNAPSHOT'`.

4.1.5 Changing the version of the module

This can be done manually, POM-by-POM or you can do it with the Versions Maven Plugin. The Table 3 shows how to do it.

Maven command	Description
<code>\$mvn versions:set \</code> <code>-DnewVersion=1.2.1-SNAPSHOT</code>	Change the module to a snapshot version. Used in the same directory level of the parent POM.
<code>\$mvn versions:set -DnewVersion=1.2.1</code>	Change the POMs to a release version. Used in the same directory level of the parent POM.
<code>\$mvn versions:use-releases</code>	This command can override the previous by removing the <code>"-SNAPSHOT"</code> from the POM version when a release version of the module is about to be generated.

Table 3: Version plugin goals.

This plugin is used just to change the version numbering in the POM file. The next section shows how to automate a release generation, including changing the POM's version, dispensing the use of this plugin.

4.1.6 Generating a release

The Releases Maven Plugin is useful to automate the process of generating a release. There are two phases to accomplish when generating a release using this plugin:

- Prepare
 - Check if there are no uncommitted changes;
 - Check if there are no SNAPSHOT dependencies;
 - Change the POMs version removing the “-SNAPSHOT”;
 - Change the SVN information in the POM to be associated with the tag;
 - Run the project tests with the modified POMs;
 - Commit the POMs;
 - Tag the code in SVN with the version name;
 - Change the version of the POMs to a new snapshot version;
 - Commit the modified POMs.
- Perform
 - Checkout from the SVN the code from the new tag;
 - Run Maven goals to release the project.

The Table 4 shows how to use the release plugin goals.

Maven command	Description
<i>\$mvn release:prepare</i>	<i>Prepares the process and generate the release.properties file to be used by the perform goal.</i>
<i>\$mvn release:perform</i>	<i>Perform the release has described previously.</i>
<i>\$mvn release:clean</i>	<i>Cleans the backup POM files and remove the release.properties file.</i>
<i>\$mvn release:branch</i>	<i>Performs branches and updates both the POM versions on trunk and on the newly created branch.</i>
<i>\$mvn release:rollback</i>	<i>If the clean goal haven't been executed, this rollbacks the previous generated release.</i>

Table 4: Releases plugin goals.

4.1.7 Remote repository manager

There is a server that will manage the artifacts generated by the development team. This server is available at: <http://heartbeat.ieeta.ot:8180/nexus>.

When you use the command “*mvn deploy*”, the Maven will upload all generated artifacts by the build to the repository manager. And if your module depends on a specific version of an artifact generated by your team members, you can use the repository manager to download it. Maven will do it automatically has needed.

There is no need to e-mail the dependencies between developers and to move them to the SVN. The dependencies are available on the fly in the project's artifacts repository.

4.2 Continuous Integration Server

The RTS project base the development paradigm by using a Continuous Integration server. The project uses Hudson is available at: <http://heartbeat.ieeta.pt:8180/hudson>

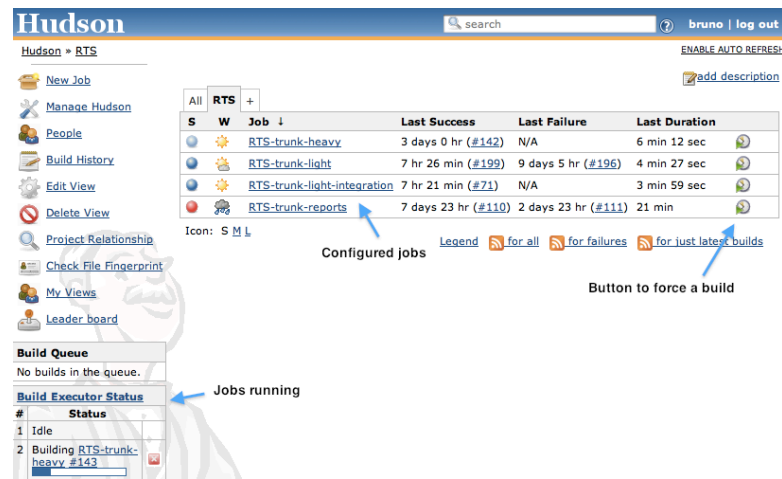


Figure 6 - General view on Hudson.

Within Hudson's main web page, the user can view all configured integration Jobs, the currently Jobs performing builds and other general options (Figure 6).

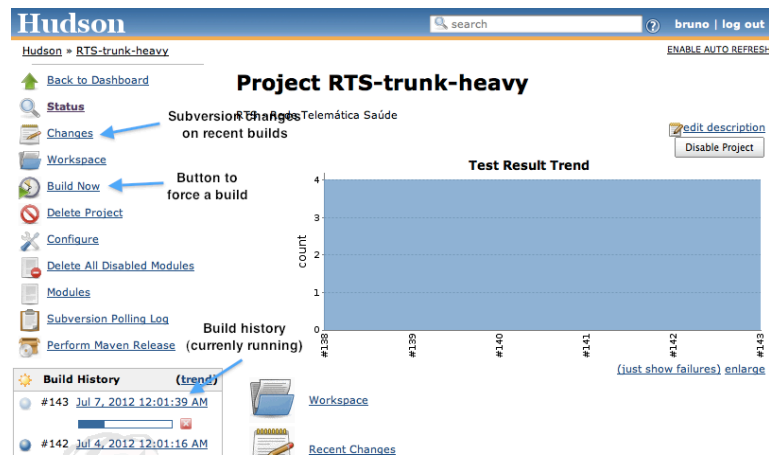


Figure 7 - Job view on Hudson.

On each Job, Hudson has several options. Users can force the Job to execute a build, view the recent Subversion changes, configure the Jobs, etc. This view shows the build history and, if applicable, the recent unit test cases results (Figure 7).

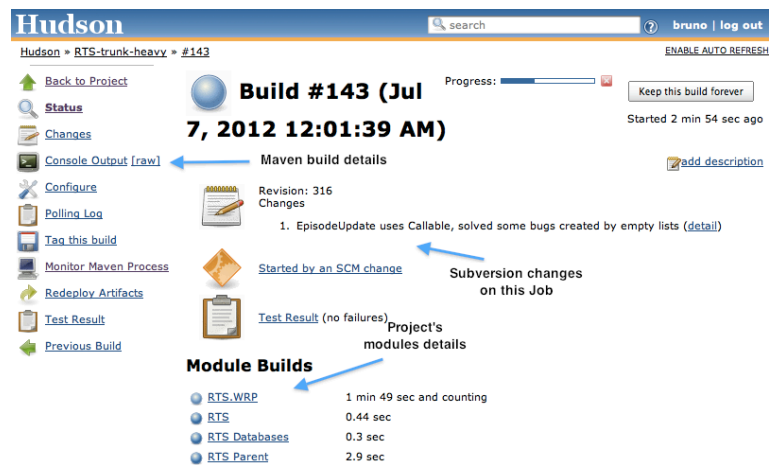


Figure 8 - Build details on Hudson.

Each build within a Job can be viewed in detail. The user can view the console output to see the details of the Maven build. This view enables the user to view details of the build on the project's module (Figure 8).

4.3 Your project's code repository (SVN)

4.3.1 Configuring a graphical SVN client

The repository path is located at:

<https://heartbeat.ieeta.pt/svn/rts>

If you use, for example, SmartSVN has an SVN client, you can create a new repository profile as showed in Figure 7.

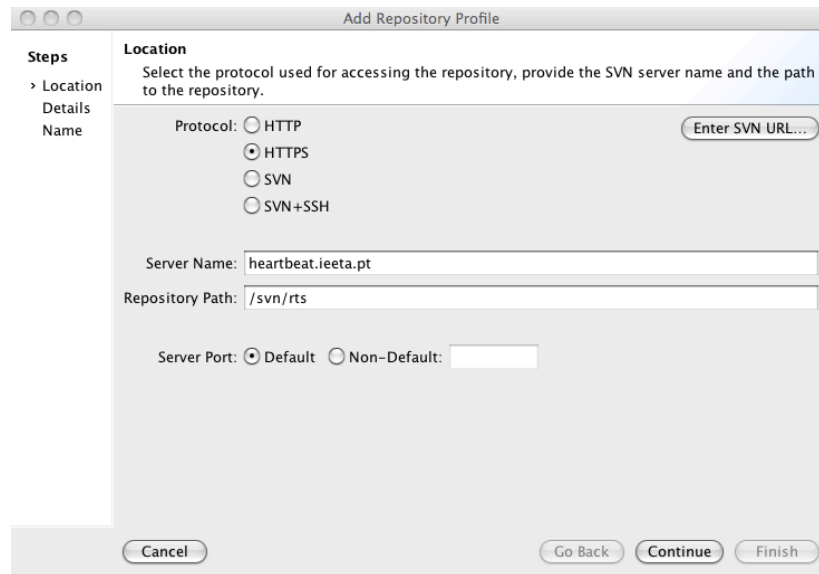


Figure 9 - SmartSVN repository profile.

4.3.2 How repository is organized

The project's repository is organized in three directories. Each one has a specific function:

- “branches”: A branch can be used for personal purposes, development purposes and versioning purposes;
- “tags”: For versions only. A specific version must be mapped with a tag in this directory. Is mandatory to never commit to a tag;
- “trunk”: This directory is where the most update code must be. All developers commit to this directory.

4.3.3 Operations using command line

Command	Result
<i>\$svn checkout https://heartbeat.ieeta.pt/svn/rts</i>	Get a working copy of the project (CHECKOUT)
<i>\$svn commit file1.java -m "Fixed a bug"</i>	Committing after changing files (COMMIT)
<i>\$svn update</i>	Update you working copy with remote changes (UPDATE)

Table 5 - Basic SVN commands

Commands	Description
<i>\$svn checkout https://heartbeat.ieeta.pt/svn/rts/trunk</i>	Checking out the trunk (TRUNK)
<i>\$svn checkout \</i> <i>https://heartbeat.ieeta.pt/svn/rts/branches/1.2</i>	Checking out a branch (BRANCH)
<i>\$svn add file2.java</i>	Adding a new file to the repository (ADD)
<i>\$svn delete file2.java</i>	Deleting a file from the repository (DELETE)
<i>\$svn status file1.java</i>	Overview (STATUS) - (A)dded; - (C)onflict; - (D)eleted; - (M)odified
<i>\$svn diff</i>	View details of your local modified

Coding Style and Developer Best Practices

	files (DIFF) Created lines preceded by "+" Removed lines preceded by "-"
<pre>\$svn copy https://heartbeat.ieeta.pt/svn/rts/trunk \ https://heartbeat.ieeta.pt/svn/rts/branches/1.9 \ -m "Create new branch"</pre>	Creating a branch from the trunk
<pre>\$svn copy https://heartbeat.ieeta.pt/svn/rts/trunk \ https://heartbeat.ieeta.pt/svn/rts/tags/2.1</pre>	Creating a tag from the trunk (TAG)
<pre>\$svn copy -r 123 \ https://heartbeat.ieeta.pt/svn/rts/branches/2.5 \ -m "Create branch from revision 123"</pre>	Creating a tag from a specific revision
<pre>\$svn delete \ https://heartbeat.ieeta.pt/svn/rts/branches/my_branch</pre>	Deleting a branch (this command can be used to delete other elements, like files and directories)

Table 6 – Common SVN commands.

4.3.4 View SVN log

This can be very useful to view last commits, view revisions, modified files, detect repository activity, etc.

- Subversion command-line client:

The command 'svn log -l 5' will show the last 5 log messages from the repository. Used with -v option (verbose), will show the evolved files too.

- Eclipse IDE:

The user must follow the menus 'Window->Show View->Team->History' and hit the 'refresh' button.

- Netbeans IDE

Doesn't support this feature, must use an external tool.

- SmartSVN tool:

The log is presented in the 'Transactions' section with help of a selection tree, a refresh button and transaction filters.

4.3.5 Update the working copy with the latest's developments

Don't forget to maintain your working copy synchronized with the project's repository every time. Occasionally, you may need to update just parts of the project.

- Subversion command-line client:
Use the command 'svn update' at the desired location. The use of this command updates all elements bellow in the current directory. This command can be used to update files passed in the argument.
- Eclipse IDE:
With an appropriate view, select a project, package or file from the tree, right click over-it and select 'Team->Update to HEAD'. This works right clicking over the code editor to update the opened file.
- Netbeans IDE:
Select the desired element (project, module, package or file) and use the application menu 'Team->Update'.
- SmartSVN tool:
Select the file or directory in the tree and click the "Update" button.

4.3.6 Commit the changes to repository

This action must be used with careful because it can cause the repository code not to compile. Incomplete commits, unversioned new files, etc. can cause this. With unversioned files extra careful must be taken because the code compiles locally.

Don't forget to write an explicit commit message explaining the task and, if possible, refer an ITS issue key in the message.

- Subversion command-line client:
Use the following command: 'svn commit -m "message" PATH. Where 'PATH' can be added, modified or deleted files or directories. This parameter can be used with wildcards.
- Eclipse IDE:
Select the desired element (project, package or file) and using the right click menu 'Team->Commit', the IDE will commit the related modified, added or deleted files or directories.
- NetBEans IDE:
After selecting the element (project, module, package or file) select the application menu 'Team->Commit' and the IDE will commit related modified, added or deleted files or directories.

- SmartSVN application:
Press the 'Refresh' button. If you wish to commit directories or files under it, select the desired directory in the tree. If you wish to commit individual files, select the added, modified or deleted files to commit.

4.3.7 Create new versioned file

This operation will add new files to your working copy under management of SVN. Before committing check if you will send only the intended files. Its very frequent to see IDE generated files, temporary files and other unwanted files committed to SVN.

- Subversion command-line client:
The command 'svn add PATH' will add to version control the specified file or directory to the working copy. The 'PATH' parameter can be a file or directory and wildcard can be used.
- Eclipse IDE
Choose the unversioned project, module, package or file in the appropriate view, right-click the item, select the menu 'Team->Add to version control'. This action can be done over the file editor.
- Netbeans IDE
Just create a new project, module, package or file in Netbeans. Note the new created items are not versioned. The operation to add to working copy will be done before the commit operation by the IDE.
- SmartSVN tool
Use the application menu and check the 'View->Unversioned' menu, find the target files or directories and click the "Add" button.

4.3.8 Move/remove versioned files

This action is often used in refactoring and the IDE will do this automatically. Use this operation to move and rename versioned files or directories and to maintain the SVN history tree consistent. Don't use the 'svn add' and 'svn remove' to move or rename files.

- Subversion command-line client:
Use the command 'svn move SRC DST' on versioned files to move or rename them. The 'SRC' and 'DST' parameters can be files or directories. To move a item, the source path must be different from the destination path.
- Eclipse or Netbeans IDEs:
In the appropriate view, drag the desired item to the destination location to move. If you wish to rename a project, module, package or file right-click over the menu 'Refactor->Rename'.
- SmartSVN tool:

Select the target file or directory and drag and drop the resource to the desired location.

4.3.9 View local modifications

It's imperative to the programmer to track and to be aware of the status of the working copy. The developer must have control over its work. Mixing multiple development issues can be confusing and lead to problems and conflicts. Despite that, always maintain absolute control over your local modifications.

Use the available tools to view all modifications all the time, don't forget to be aware of any file, versioned or not.

- Subversion command-line client:
The command 'svn status' will show either the unversioned and local changed files or directories. The unversioned files are identified by a '?'. To view only the modified files, use the 'svn status -q' command.
- Eclipse IDE:
Doesn't support totally, the closest answer to this operation is the 'Synchronize' view but it's not possible to distinguish the unversioned files and newly added files. Must use an external tool.
- Netbeans IDE:
Select the project, module, package or file, right-click over it and choose the menu 'Subversion->Show changes'.
- SmartSVN tool:
In the application menu 'View' uncheck the 'Unchanged files', the 'Unversioned files' and the 'Ignored files'. The coloured files and marked directories are the modified items. Use the tree to view the changes under the selected directory.

4.3.10 Revert local modifications

Reverting local modifications is an operation that can't be undone. It could be the lost of hours and days of work. Unless if you want to revert all your work, use this operation atomically with small and controlled reverts, accomplished with file inspection.

- Subversion command-line client:
To revert all the modifications under the current directory, use the command 'svn revert -R .'. To revert specific files or directories use the command 'svn revert PATH', where 'PATH' parameter can be a file or directory with wildcards.

- Eclipse IDE:
Select the desired item (project, module, file, etc.), right-click over it and select the menu 'Team->Revert'.
- Netbeans IDE:
Select the specific project, module, package or file, right-click it and choose the menu 'Subversion->Revert'.
- SmartSVN tool:
Select the file or directory, right-click over it and choose the menu 'Revert'.

4.3.11 View file modifications

It's very useful to know what code was changed in the working copy relative to the pristine copy. Knowing immediately and easily the local file modifications is a feature that every modern IDE offers to the developers. This "quick diff" gives the programmer the possibility to revert a single line or blocks in a single click.

- Subversion command-line client:
Use the command 'svn diff PATH'. The 'PATH' parameter can be a file or directory and the output is the changed, added and removed lines of code.
- Eclipse IDE:
This feature must be enabled first. To accomplish this, go to the general preferences menu, select 'General->Editors->Text editors->Quick diff' and change the reference source to 'Pristine SVN copy'. Then, the changes will appear in the left side of the file editor with colour representation. Mouse-over the coloured area to view the original code.
- Netbeans IDE:
The same quick diff method with coloured representation used on Eclipse IDE.
- SmartSVN tool:
Select the modified file, right-click over it and choose 'Show changes'.

4.3.12 Compare changes from different revisions

Comparing zones of code can help track bugs and revert changes between revisions. This feature is especially relevant to view differences between branches and to ensure that a development branch is properly synchronized.

- Subversion command-line client:
Use the command 'svn diff TARGET1@[REV1] TARGET2@[REV2]'. The 'TARGET1' and 'TARGET2' parameters are the elements to compare. These items can be in different branches. The 'REV1' and 'REV2' are the revision numbers.

- Eclipse IDE:
Over the file right-click and select the menu 'Team->Show History'. Select the two revisions from the list, right-click and select 'Compare'.
To compare a specific revision with other linked to another branch, choose the revision, right-click over it and select 'Compare', then use the graphical components to select the branch and revision to compare.
- Netbeans IDE:
Choose the file to compare, right-click and select the menu 'Subversion->Search History'. Netbeans doesn't support diff operations between branches, must be used an external tool.
- SmartSVN tool:
Select the file, use the menu 'Query->Compare with revision' and select the path and revision. It's not possible to compare two arbitrary revisions between branches. The developer only can compare the HEAD of the current revision with an arbitrary revision from another branch.

4.3.13 Conflict after an update

Make one of tree things if there is an conflict on a file:

1. `$svn revert <file>` (reverts local file);
2. Copy one of temporary files created by conflict on top of your working file;
3. Merge the file by hand, examining the conflict markers (see below for an example).

```
$vim file1.java
<<< .mine
This is an example text inside a file
===
This is an eaxmasoda asd asd ds a file
>>> .r2
$svn resolved file1.java
```

You can find yourself with a tree conflict. This is caused when other developers move or delete a file that you are working on.

For example:

Coding Style and Developer Best Practices

```
$svn status  
M file1.java  
A + C file2.java  
> local edit, incoming delete upon update  
M file3.java
```

This conflict must be resolved manually by analyzing if this deletion is supposed to happen and then remove the file if you decide it:

```
$svn remove --force file2.java  
D file2.java  
$svn resolve --accept=working file2.java  
Resolved conflicted state of 'file2.java'
```

4.4 Versioning

4.4.1 Overview

The version of an application is a reproducible state at any point in the time, including all artifacts that made possible to create that version, like the source-code, test code, database scripts, build and deployment scripts, documentation, dependencies and configuration files of a given application. It should be possible from scratch to reproduce any given version of the produced software on any environment.

4.4.2 Version-numbering strategy

In the RTS project, the version-numbering schema is structured as a triplet of integers: MAJOR.MINOR.PATCH, each one with a specific meaning described on Table 7.

Segment	Description
Major	When breaks up compatibility
Minor	When enough requirements are implemented and the changes are externally visible
Patch	Bug fixes without new requirements

Table 7 - Version segments.

Each segment of the project's version number increment a unit depending on the changes that were introduced:

- Patch increments:
 - Only when bug fixes are added;
 - No changes on data structures;

Coding Style and Developer Best Practices

- No changes on API definitions;
- Never include new features.

- Minor increments:
 - When includes new features;
 - Bug fixes can be added too;
 - API compatibility should not be broken;
 - New method can be added;
 - Existent methods can be marked as deprecated;

- Major increments:
 - Large changes that breaks up compatibility with older versions.

On the ITS level of perspective, when a new release version is about to be closed (major, minor or patch), all issues targeted to that version, if possible, should be closed, tested and validated. If there is the impossibility of solving all the issues aimed for the version to be generated, their target should be shifted to a future version.

4.4.3 Releasing a version

The most straightforward situation for generating a new version is when a team comes across with just one development line and all newly developed issues are targeted for the next release. Therefore, when the deadline dues with all related issues committed, closed and validated, ready to be included in the new version, all the POMs that define the project must be updated by replacing the snapshot versions by their release versions. This includes the versions of the project's dependencies.

In this simple scenario, when a new version is about to be tagged and released, the team members must be alerted to not commit any change to the related development line during this transitional process. There may be occasions in which it's not possible to stop adding changes to that line of development during the releasing of a new version but that difficulty can be overcome by forking a version branch sourced in the SVN revision that should be tagged. This scenario will be explained in the next section.

Figure [?] shows an example on how to generate a new minor version and a new patch version on a single development line, commonly applied on small development teams with modest activity. When the snapshot version is stable and all correspondent issues are closed, the POM versions must be updated and

```
$mvn release:prepare -DautoVersionSubmodules=true  
$mvn release:perform
```

Figure 10 - Using Maven Release Plugin to perform a version release.

Coding Style and Developer Best Practices

Using the example exposed in Figure 11, the initial development version is “1.2-SNAPSHOT” which will be changed to the release version “1.2” after all issues are completed. After perpetuating that version with a tag, the new development version could be the “1.3.0-SNAPSHOT”, if the team intent to develop new features. But there might emerge the need to generate a different snapshot version, like the “1.2.1-SNAPSHOT” when are critical corrections to fix over the “1.2” release version. This Maven Release Plugin includes a goal to

```
$mvn release:update-versions -DautoVersionSubmodules=true
```

Figure 12 - Using Maven Release Plugin to change POM versions.

change the POMs versions and can be used as shown in Figure 12.

This command can change the “1.3.0-SNAPSHOT” to “1.2.1-SNAPSHOT”, for instance. Later, after adding the desired fixes, the “1.2.1” release version could be created by using again the “prepare” goal of the Maven Release Plugin.

Releasing with multiple development versions

Teams with active developments almost certainly will face the incompatibility of producing multiple scheduled versions on the same development line, typically the *trunk* on Subversion. The developers must bear in mind that the trunk should contain the latest developments of the project, even if there are active branches on the run. The trunk is the main development line and should be used for integrating the team’s work all the time. Therefore, development branches should integrate regularly to trunk.

One of the acceptable situations that allow the trunk to be branched is when a release version is reaching the deadline and all features of that version have been developed. This procedure ensures a way to isolate the new coming version from the instability of the trunk developments and to stabilize it with tests and validations [10]. Only bug fixes should be committed to the branch, which must be merged to the trunk immediately. There are rare exceptions when the merging of corrections will not be done but only with the project manager’s specific instructions.

A perfect example to exhibit the prior procedure is presented in Figure [?]. A release candidate (RC) is created when the last requirement targeted to the “1.2” version is implemented for the reason that the team need to implement new features for the “1.3” version. During the stability phase of the “1.2” version, tests encounters some bugs that should be fixed. Those bugs are corrected by the development team and merged to the trunk, while new unrelated requirements are implemented for the “1.3” version. When the “1.2” version is stable, it’s time to generate the final release.

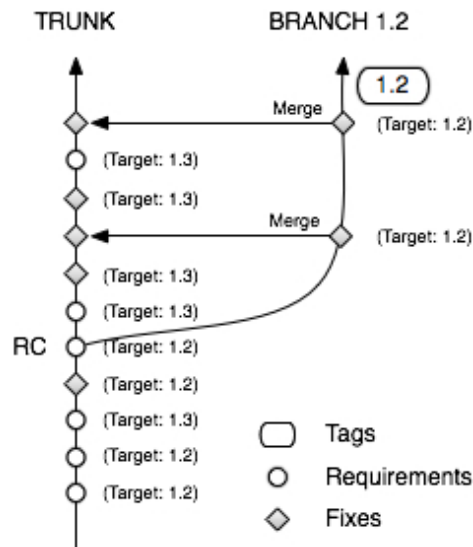


Figure 13 – Generation of the version 1.2 when next iteration (1.3) is in development.

The Maven Release Plugin offers the possibility to automatically create a version branch on Subversion (or other SCM) when the team wants to stable a release candidate. As shown in Figure 13, the “1.2” version will be created when the “1.3-SNAPSHOT” is on the run so, the POMs versions in the trunk when the RC is created corresponds to the “1.3” snapshot version. Inside the “1.2” branch, the POMs version should be “1.2-SNAPSHOT” but the *plugin* will update all necessary files automatically with that snapshot version. The Maven command presented in Figure 14 will execute the all specified steps for

```
$mvn release:branch -DbranchName=1.2 -DupdateBranchVersions=true -
DupdateWorkingCopyVersions=false -DautoVersionSubmodules=true
```

Figure 14 - Using Maven Release Plugin to create a version maintenance branch.

creating the “1.2” release candidate.

Where the “branchName” parameter will define the new branch’s name, the parameter “updateBranchVersions” will force the POMs of the new branch to be updated with the correct version, the “updateWorkingCopyVersions” parameter set to false will ignore changing the working copy versions and the “autoVersionSubmodules” parameter will allow the same version to be configured on all modules of the project.

The following phases will be executed by the “branch” goal [11]:

Coding Style and Developer Best Practices

- Verify if there are no local changes to commit;
- Create the new branch with the specified name;
- Update the POMs of the newly created branch with the prompted version;
- Commit the modified POMs.

After the development branch is created, developers can start doing all the necessary fixes, without forgetting to merge all changes to the trunk. Finally, when the release candidate is stabilized, the final version could be released by using the previously mentioned commands on Figure 10. This version branch can be used for maintaining the release version with possible bug fixes found on production environments, as will be explained in the next section.

Releasing corrective versions

Despite the effort to increase the quality control over software projects, is impossible to any development team to produce released applications without bugs. Eventually, when a bug is found, probably on a production environment, the development team must have the ability to reproduce the version where the bug was found to increment the needed corrections over it. On situations like this, branching from the correspondent tag will allow to add an increment to the exact version where the bug was found and to release a patch version. Without exception, the applied corrections to that version should be reproduced in the trunk. If the bug was found on a very old version, programmers must try to reproduce the problem to ensure that is applicable to the new version and might have to correct it on a more evolved code with different circumstances using different solutions.

The Figure 15 shows an example on how to correct bugs to an already released version. In this case, some issues were found on “1.2” version, despite the newest version of the project being superior to “1.3.1”. If the maintenance branch for this version isn’t already created, the development team must create it from the correspondent tag to increment the desired changes, such is the case shown in this example, and the POM files of the new branch must be updated with the patch version the team wants to publish, in this case, will be “1.2.1-SNAPSHOT” because the “1.2.1” version is under development until the corrections are stabilized.

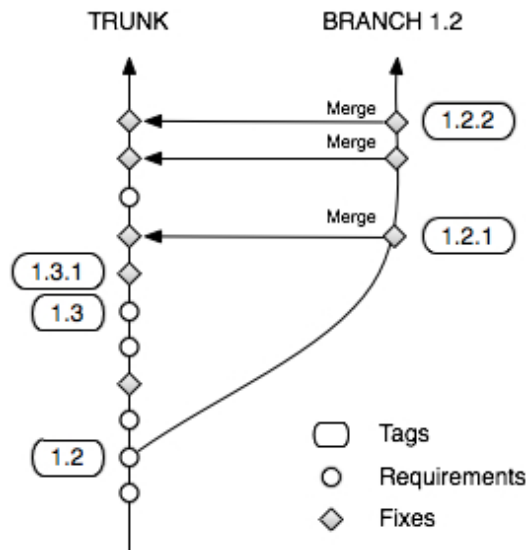


Figure 15 - Generation of the patch 1.2.1, when other versions were released.

Currently, Maven Release Plugin doesn't support creating a branch from a tag therefore the way to do that is to use Subversion commands available on every SVN client. On the example of Figure 15, after the branch is created from the "1.2" tag, the appointed personnel to solve the issues must checkout that branch and change its POM files with the new development version. As was previously used, the following command shown on Figure 12 will update those files.

After the POMs are updated with "1.2.1-SNAPSHOT", developers can start fixing the found issues, without forgetting to merge all changes to the trunk. Finally, when the patch version is stabilized, using the following commands specified on Figure 10 the release will be performed. This branch set is available for other possible fixes and is designated for patch increments, nothing else, making "1.2.2-SNAPSHOT" the next version to use on the branch's POM files. As shown in Figure 15, the 1.2.1 version can be incremented with other corrective version 1.2.2, and so on.

5 Programming patterns

5.1 Generic programming techniques

Review and study the following resources:

- Check Java [Programming Practices](#) in the official Java style guide.
- Check Bruce Eckel's [Thinking in Java](#) book guidelines.

5.2 Patterns for logging

5.2.1 Introduction

Logging should be considered as part of the application source-code, like any other regular feature and as such, its use should not be neglected, neither the impact of the bad written logging code on the system. Logging code help improve the application's quality. Thus, the logging code should be written to have the minimum impact on the performance of a running system.

5.2.2 Logging efficiency

- Logger naming conventions

For maintainability reasons, when instantiating logger objects, it should be used package base naming conventions, like the fully qualified package of the class whose being logged. Exceptionally, when different components in the same package have different logging behaviour, the loggers should be separated by the different performed tasks.

Coding Style and Developer Best Practices

```
public class ClassExample {
    public interface SpecialLoggers {
        Logger business = Logger.getLogger("business");
        Logger security = Logger.getLogger("security");
    }

    /**
     * The category of messages using this logger is the fully qualified name of
     * the class.
     */
    private static final Logger logger = Logger
        .getLogger(ClassExample.class.getName());

    public void doSomethingAgain(int parameter, String username) {
        if (logger.isDebugEnabled()) {
            logger.debug("Passed parameter: " + parameter);
        }

        if (SpecialLoggers.security.isEnabledFor(Level.WARN)) {
            SpecialLoggers.security.warn(" The user [" + username
                + "] had done something.");
        }
    }
}
```

Figure 16 - Multiple purpose Logger objects.

- Logging parameters and internationalization

When processing a log line, the first thing to be computed is the log parameter, typically a String object (Figure 17). Parameter constructions within the log message are costly and they should be wrapped with a level check. This issue will be addressed forward in the Guarded Logging section.

Using *ResourceBundle* objects is extremely costly and should be avoided when the application is implemented with only one language support. Together, parameter construction and internationalization are the most critical performance aspect to take care when using logs.

Coding Style and Developer Best Practices

```
public class StringParameterExample {
    private static final Logger logger = Logger
        .getLogger(StringParameterExample.class.getName());

    public static void main(String[] args) {
        String str = "Ok?";
        logger.debug("This" + " " + "is" + " " + "wrong!" + str);

        if (logger.isDebugEnabled()) {
            logger.debug("This is right!" + str);
        }
    }
}
```

Figure 17 - Parameter construction on logging.

5.2.3 Guarded logging

Guarded logging is a pattern to check if a given logging level is active, before executing the related log message.

```
public class GuardedLogging {
    private static final Logger logger = Logger.getLogger(GuardedLogging.class
        .getName());

    public static void main(String[] args) {
        for (int i = 0; i < 90; i++) {
            logger.debug("Value " + i + " is integer.");
        }

        if (logger.isDebugEnabled()) {
            for (int i = 0; i < 90; i++) {
                logger.debug("Value " + i + " is integer.");
            }
        }
    }
}
```

Figure 18 - Guarded logging pattern.

Avoids the unnecessary parameter construction and localization translation, because this check is verified after that construction. Almost every

Coding Style and Developer Best Practices

log messages use *String* objects and the accumulation of operations over them causes memory fragmentation and unnecessary garbage collection.

5.2.4 Logging framework configuration

- Changing log4j configuration on runtime

Restarting the framework's configuration without killing the application can be achieved using the *configureAndWatch* options at a starting point of the application. The framework will pool for configuration changes at a specified time interval and restart the logger when the configuration is changed. Figure 19 shows an example on how to use this feature.

```
public class ConfigureAndWatchExample {  
  
    public static void main(String[] args) {  
        initApp();  
    }  
  
    private static void initApp() {  
        DOMConfigurator.configureAndWatch("/opt/app/conf/log4j.xml");  
    }  
}
```

Figure 19 - Example on how to watch changes in log4j configuration files.

- Log file rollers

Using log rotation, old log messages are automatically archived so that new messages are appended to a smaller file. In Log4j, the *DailyRollingFile appender* can be used to rotate log files as shown on Figure 20.

```
...  
<log4j:configuration>  
<appender name="dest" class="org.apache.log4j.RollingFileAppender">  
<param name="file" value="${user.home}/log.out" />  
<param name="threshold" value="error" />  
<param name="immediateFlush" value="true" />  
<param name="append" value="true" />  
<param name="datePattern" value="'.! yyyy-MM-dd " />  
...  
</appender>  
...  
</log4j:configuration>
```

Figure 20 - Log file rotation configuration example.

- Formatting log messages

The Log4j framework implements multiple *Layout* objects to help format the messages, besides providing the possibility of custom layouts to be implemented extending the abstract super class *Layout*. These *Layout* implementations include XML and HTML formatters, layout suitable to multithreaded environments and date formatters.

Other main concern about log output is the date of the event that generated a given log message. When analysing logs, knowing when those messages occurred is critical but they slow down the system performance. Log4j provides various date layouts and the most efficient is the *ISO8501DateFormat*.

5.2.5 Guidelines for using logging levels

There are seven different levels defined in Log4j, respectively arranged: All, DEBUG, INFO, WARN, ERROR, FATAL and OFF, where ALL allows all information to be published and OFF doesn't allow any information to be published.

Commonly, these levels when associated to log messages are wrongly used by developers, so it should be defined clear guidelines before they start coding:

- **DEBUG:** This level of priority is used for problem diagnosis, since it provides extensive background information. It should be mostly used in the development phase. Logging information linked to this priority should be from developers to developers. On normal circumstances, this level of priority is not active on production environments, except for problem detection.
- **TRACE:** This special level will complement the DEBUG level because shows more debug information and should be used with the previous guidelines.
- **INFO:** To help trace the business behaviour of a running application, this level is the appropriated one. Developers should write these log messages for users and administrators without exposing implementation details. Log messages within this level represent the application's operations flow, including its components, thus they should be clear and should enlighten about what's the application doing.
- **WARN:** A warning log may represent a potential problem on the application. This kind of logging shows information about unexpected situations that needs immediate attention but will not prevent the majority of operations to continue flowing. This level shouldn't be used when the problem crash some function.

- **ERROR:** Indicates a severe problem in the system and is usually not recoverable and doesn't exempt manual intervention. Error messages indicate a significant or complete loss of an operation but still allowing the application to continue running.
- **FATAL:** Presents critical problems causing the application to abort. Fatal shouldn't be used if the problem is transient or any of the operations will continue.

5.3 Patterns for Exceptions handling

5.3.1 Introduction

- Checked exceptions:

These kinds of exception are used to anticipate exceptional situations and to help the application to recover from them and must make sense to the code that handles it.

Checked exceptions extends from the *Exception* class and they are always subject to the try-catch block or the throws clause, apart where was caught.

- Runtime or unchecked exceptions:

Unchecked exceptions are internal to the application and are thrown during unexpected situations - they could indicate a programming bug. It might be more correct to fix the bug that causes the exception rather than catch them.

This kind of exception extends from *RuntimeException* class.

- Errors:

Errors are external to the application that might not be able to recover from them, like the runtime exceptions. The best handling strategy to these exceptions is to print the stack trace and terminate the program. When an error is thrown, might indicate a system malfunction, an overflow in the stack or lack of memory to continue to run the application.

This kind of exception extends from *Error* class.

5.3.2 Common errors on exceptions

- Swallowing exceptions

```
public void doSomethingElse() {
    try {
        doSomething();
    } catch (CustomException e) {
        // What was i thinking?
    }
}
```

Coding Style and Developer Best Practices

This kind of exception treatment makes the information to be lost forever and might indicate that the code within the try-catch blocks is wrong. Despite not being a good practice, if the developer is facing one of the rare circumstances where using this exception handling is appropriate, at least the practice should be documented or logged. Programmers to shut up the compiler when the code throws a checked exception commonly use this bad practice.

- Log and throw

```
public void doMoreStuff() throws CustomException {
    try {
        doSomething();
    } catch (CustomException e) {
        logger.debug("Error doing some stuff!");
        throw e;
    }
}
```

Figure 22 - Log and throw anti-pattern.

As explained previously, when an exception is thrown, the runtime system will search the call stack, in inverted order of the method execution, to find the handle to deal with the exception. If the same exception is logged in multiple places, the log output will show multiple logs to the same action. This overloads the logs and denigrates the system performance. Either the exception should be logged or thrown, but never the both.

- Exceptions that are too general

```
public void processSomething(String something) throws Exception {
    if (something.length() == 0) {
        throw new Exception("There nothing to do!");
    } else {
        // Going to process something
    }
}
```

Figure 23 - Method that throws the Exception object.

Coding Style and Developer Best Practices

Such an approach, like throwing Exception, is against the all point of using checked exceptions. This violates the right level of abstraction of an interface, and throwing a too general exception just advises that something can go wrong inside a method. Instead, it should declare the checked exceptions the method can throw. This may cause the next problem to arise.

- Throwing several exceptions

```
public void analyseSentence(String sentence)
    throws InvalidSentenceException, EmptySentenceException,
    SentenceAnalysisException {
    if (sentence == null) {
        throw new InvalidSentenceException("The sentence cannot be null");
    } else if (sentence.length() == 0) {
        throw new EmptySentenceException("The sentence cannot be empty");
    } else {
        // Analyse the sentence
    }
}
```

Figure 24 – Throwing several exceptions.

If a method throws several checked exceptions, might indicate that they should be wrapped in a new custom exception. This is particularly true if basically all exceptions mean the same. But the opposite situation when the different thrown exceptions can conduce the caller to different ways of action negates that proposition.

Coding Style and Developer Best Practices

```
public void goCollectFlowers() throws EmptyFieldException, NotTodayException {
    try {
        runToField();
    } catch (BrokenLegException e) {
        // Will conserve the original stack trace
        new NotTodayException("Call an ambulance", e);
    }

    try {
        observeField();
    } catch (FlowersStolenException e) {
        // Will destroy the original stack trace
        throw new EmptyFieldException("All flowers were stolen");
    }
    pickFlowers();
}
```

Figure 25 - Destructive wrapping anti-pattern.

- Destructive wrapping

Destructive wrapping used to encapsulate other exceptions makes difficult to detect some problems that arise from the application. The original stack trace information from the wrapped exception shouldn't be lost forever within the wrapper exception.

- Breaking the encapsulation

```
public void goCollectFlowersAgain() throws IOException {
    // Collect flowers code goes here
}
```

Figure 26 – Exception breaking the method's encapsulation.

When throwing exceptions that make no sense to the caller exposes the implementation details. This causes the encapsulation to be broken, besides increasing the complexity of the code. This might indicate a wrong approach on error-handling logic inside the method that throws unrelated exceptions. Or those exceptions should be wrapped to exceptions the caller can use.

5.3.3 Basic principles of exception handling

- The exception should be caught as closest as possible from the source.

If the caller handles the exception, should take care of it immediately by wrap it, log it or take measures with appropriate code. If exceptions are a mechanism to deal with unexpected situations in a certain context of a call to a method, it should be treated according to that context, for readability reasons. Uncaught exceptions shouldn't be thrown if the error can be handled locally.

- When an exception can't be handled, it shouldn't be caught.

If the code can't handle the exception, something may be wrong in the caller's level of abstraction. Probably the code within the try block is invalid because it raises an exception for no reason or the exception is supposed to be thrown higher in the call stack. Nevertheless, if there is no alternative but to catch, at least, it should be documented or logged.

- Exception messages must include all information that led to the exception

Exception occurs in particular situations in a certain context. To realize where and why occur, the information message must include the circumstances under which occurred when they were thrown. This is another reason for them to be caught nearest to the origin as stated previously.

- When appropriate, the exception should be logged where was caught

Unless the exception is supposed to be thrown, logging is done exclusively where was caught. This practice will avoid duplicate messages on logging systems and shun using the machine in a careless manner because logging consumes resources.

- Exceptions should respect the interface's abstraction

Exceptions may be declared on methods of an interface. If the method's parameters should respect the class's level of abstraction, exceptions should to. Exceptions consistent with that level of abstraction makes possible to hide the implementation details of a class. Thus, the implementation complexity of a code where that interface is used is obviously decreased.

5.3.4 Custom exception classes

There are some guidelines that should be taken into consideration when designing a new exception. A new exception should be created when:

- There is no representation of the exception in a accessible API;
- A routine throws more than one similar exception;

- Wrap exceptions to hide implementation details.

5.3.5 The use of unchecked exceptions:

Developers with unchecked exceptions are urged to write code that just throws exceptions of that kind, because it apparently improves the code's readability and apparently fixes the problem that is trying to solve.

Unchecked exceptions should be used to represent programming bugs and unexpected exceptions. This is the reason they aren't thrown on methods of a class. Other reason not to throw unchecked exception it's because they are numerous.

Briefly, errors that the application cannot recover from are unchecked exception. Otherwise, if the application can recover from an exception, is a checked one.

5.3.6 Exceptions on multi-module systems

Generally accepted rules when dealing with exceptions on multi-module systems:

- Declared exceptions should make sense to the caller method;
- Exceptions thrown by a public method should belong to the same package of that method;
- If a checked exception is thrown by a method in one package, it should be wrapped when the exception propagates to a second package.

Heuristic when the caller receives a checked exception and don't know how to handle it:

- The exception belongs to an expected situation and should be translated (wrapped) to an exception that the caller understands;
- The exception belongs to an expected situation but it can't be handled by the caller and can't be translated, so it must be treated as an error. The method should hide it and re-throw as a *SystemException*, the main routine will handle it and log it;
- The exception doesn't belong to an expected situation, so it must be treated as an error. A *SystemException* should wrap it. The main routine will handle it.

6 References

http://wiki.eclipse.org/Version_Numbering
<http://maven.apache.org/guides/>
<http://svnbook.red-bean.com/en/1.4/svn-book.html>
<http://www.tigris.org/scdocs/SVNTips>
<http://www.redmine.org/projects/redmine/wiki/Guide>
<http://mojo.codehaus.org/versions-maven-plugin/>
<http://maven.apache.org/plugins/maven-release-plugin/>
<http://download.oracle.com/javase/tutorial>
<http://code.google.com/p/solidbase/wiki/ExceptionHandlingGuideline>
<http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html>