



# VISUALIZATION OF SOFTWARE DEVELOPMENT METRICS ON A DASHBOARD

Hector Miquel Vidal i Dura



Manresa's School of Engineering – EPSEM  
Technical University of Catalonia  
2016



# VISUALIZATION OF SOFTWARE DEVELOPMENT METRICS ON A DASHBOARD

Hector Miquel Vidal i Dura

24 ECTS thesis submitted in partial fulfillment of a  
*Magister Scientiarum* degree in ICT Systems Engineering

Advisor

Sebastia Vila Marta

Klaus Zesar

Faculty Representative

Sebastia Vila Marta

Committee

---

---

---

Manresa's School of Engineering – EPSEM

Technical University of Catalonia

Barcelona, June 2016

## VISUALIZATION OF SOFTWARE DEVELOPMENT METRICS ON A DASHBOARD

THE RESEARCH DESCRIBED IN THIS THESIS IS AN ATTEMPT TO ANALYSE MULTIPLE KEY PERFORMANCE INDICATORS IN SOFTWARE DEVELOPMENT PRODUCTION AND SHOW THEM AT GLANCE IN A DASHBOARD. SO IT PROVIDES A BRIEF VIEW ON SOFTWARE ENGINEERING PROJECTS, SOFTWARE QUALITY AND SOFTWARE METRICS IN A REAL SOFTWARE COMPANY (REVAL HOLDINGS INC.).

24 ECTS thesis submitted in partial fulfillment of a Degree degree in ICT Systems Engineering

Copyright © 2016 Hector Miquel Vidal i Dura  
All rights reserved

Manresa's School of Engineering – EPSEM  
Technical University of Catalonia  
Av. de les Bases de Manresa, 61  
08242 Manresa, Barcelona, Spain

Telephone: +34 938 77 72 00

### Bibliographic information:

Hector Miquel Vidal i Dura, 2016, VISUALIZATION OF SOFTWARE DEVELOPMENT METRICS ON A DASHBOARD, Degree thesis, Manresa's School of Engineering – EPSEM, Technical University of Catalonia.

Printing: 8010 Graz, Austria  
June 2016

*There's a tremendous bias against taking risks. Everyone is trying to optimize their ass-covering. –ElonMusk*



# Abstract

It is difficult to understand, let alone improve, the quality of software without the knowledge of its software development process and software products. There must be some measurement process to predict the life-cycle of software products. The research described in this thesis is an attempt to analyse multiple metrics in Software Development production and show them in a Dashboard. So it provides a brief view on Software Engineering, Software Quality and Software Metrics in a real software company (Reval).

The major results of this metrics, will be monitored in an easy to read, single page, real-time user interface, showing a graphical presentation of the current status (snapshot) and historical trends of the SW production key performance indicators to enable instantaneous and informed decisions to be made at a glance.

Metrics is not an interesting talk, but it a necessary thing. It is part of the necessary overhead for delivering the product. If we can detect emerging delivering risks early, then we can deal with them. If we don't discover which their weight is, then we are just blindsided and projects can fail. So it's important to measure the right things and ensure they are on track (To steer (drive) work in progress). Secondly, it is important to measure improvement efforts because otherwise we know that we are changing things and we know whether feel good but we don't know the real improvements, we can't qualify them. (To support continuous improvements efforts).

# Resum

Resulta difícil entendre, i més encara millorar, la qualitat del programari sense conèixer el procés de desenvolupament del mateix i dels seus productes. És necessari dur a terme algun tipus de procés d'avaluació per tal de determinar la qualitat dels productes (software). La investigació que es descriu en el present estudi és un intent d'analitzar diversos paràmetres en la producció del desenvolupament del programari i mostrar-los en una "dashboard". Es realitza, per tant, un breu recorregut per l'enginyeria, la qualitat i els paràmetres del programari en l'entorn d'una empresa informàtica real (Reval).

Els resultats principals d'aquests paràmetres apareixeran monitoritzats en una única pàgina, de fàcil lectura, en una interfície a temps real per a l'usuari, la qual mostrarà una gràfica de l'estat actual (snapshot) i un historial dels indicadors clau de la producció software que permetran realitzar decisions instantànies i informades d'una ullada.

L'anàlisi mètric, tot i no ser un tema interessant, és un aspecte necessari. És part de la sobrecàrrega necessària per a realitzar l'entrega del producte. Si podem detectar els possibles riscos de l'entrega a temps, els podrem solucionar. Si no descobrim quina és la seva magnitud, estem anant a les palpentes i els projectes poden fracassar. És important mesurar els aspectes adequats i assegurar-nos que van pel bon camí (per dirigir un treball progressivament). En segon lloc, és important mesurar els esforços per poder millorar ja que per molt que canviem coses i ens sembli que ho fem bé, no coneixerem les millores reals i no les podrem qualificar.



# Abstrakt

Ohne Kenntnis über Softwareentwicklungsprozesse oder –produkte die Qualität einer bestimmten Software zu evaluieren oder gar zu verbessern, ist sehr schwierig. Mit verschiedenen Messgrößen lassen sich Parameter wie Qualität und Funktionalität einer Software gut berechnen. Die in dieser Arbeit beschriebenen Forschungsergebnisse sind ein Versuch, die verschiedenen Messgrößen in der Softwareentwicklung zu analysieren und sie in einem Dashboard anzuzeigen, um einen schnellen Überblick über Qualität und Relevanz und Interdependenzen der Metriken der Software zu erhalten.

Die Ergebnisse der metrischen Berechnungen sollen auf einer einzelnen Seite in Echtzeit und leicht verständlich innerhalb eines Interfaces visualisiert werden, sodass sowohl der Status Quo als auch historische Trends der Softwareproduktions-KPIs auf einen Blick ersichtlich sind und wichtige Entscheidungen schnell und informiert getroffen werden können.

Metriken über die Software zu erstellen mag kein sehr spannendes Thema sein, es ist jedoch ein unbedingt nötiges, da das Verständnis über die Metriken für die Betriebsfähigkeit des Produkts ausschlaggebend sind. Wenn zukünftige Lieferprobleme schnell erkannt werden können, kann damit sachgerecht umgegangen werden. Wenn die Tragweite der Risiken nicht eingeschätzt werden kann, kann nicht proaktiv gehandelt werden und Projekte können darunter leiden. Aus diesem Grund ist es wichtig, die richtigen Parameter zu messen und zu kontrollieren, um Qualitätssicherung zu garantieren. Zusätzlich ist es natürlich auch wichtig, die Optimierungsprozesse zu dokumentieren um zu sehen, in welche Richtung sie die Produkte führen und um zukünftige Verbesserungsvorschläge zu evaluieren.



# Preface

Dear reader, the document you are reading right now is my final thesis with which I conclude four years of studying ICT Engineering degree program at the Polytechnic University of Catalonia.

I signed myself up for this degree out of pure curiosity for the topic, the ambition to challenge myself and acquire another perspective of thinking. I came to the conclusion that there was more to creating a good software product than just writing its code. This conclusion made me realize that in order to increase my skills and professionalism I had to increase my awareness of the aspects that together form the complete software engineering.

While I'm writing this as the last part of my final thesis document I truly believe that my skills, passion and appreciation of the software engineering process has increased. I hope that my final thesis document proves this learning process to you, the reader.

Working on my thesis research was a true learning experience for me. It has given me a better understanding of what scientific research actually is. It especially has given me a lot of respect for people that dedicate themselves to collecting facts about any research subject and share these with other people in the world making it possible to learn from.



# Contents

<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvii</b>
<b>Abbreviations</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Objective . . . . .	2
1.3 Manuscript organization . . . . .	3
<b>2 Software Engineering Basics</b>	<b>4</b>
2.1 What is Software Engineering? . . . . .	4
2.2 Why Software Fails or Succeeds? . . . . .	5
2.3 Software Engineering Activities . . . . .	6
2.3.1 PEOPLE . . . . .	7
2.3.2 PRODUCT . . . . .	8
2.3.3 PROJECT . . . . .	9
2.3.4 PROCESS . . . . .	10
<b>3 Software Configuration Management</b>	<b>11</b>
3.1 What is Software Configuration Management? . . . . .	12
3.2 Configuration Management Processes . . . . .	12
3.3 SCM Terminology . . . . .	13
3.4 Version Management . . . . .	16
3.5 SCM Best Practices . . . . .	17
<b>4 Software Design</b>	<b>18</b>
4.1 What is Software Design? . . . . .	18
4.2 Goals/Characteristics of Software Design . . . . .	18
4.3 Software Design Terminology . . . . .	19
4.3.1 Component . . . . .	19
4.3.2 Coupling . . . . .	19
4.3.3 Cohesion . . . . .	20

<b>5</b>	<b>Software Testing</b>	<b>21</b>
5.1	IEEE - Definition of Testing . . . . .	22
5.2	Testing Goals . . . . .	23
5.3	Verification vs. Validation . . . . .	23
5.4	Testing methods . . . . .	23
5.4.1	Static testing . . . . .	24
5.4.2	Dynamic testing . . . . .	24
5.5	Testing levels . . . . .	26
5.5.1	Unit Testing . . . . .	26
5.5.2	Integration Testing . . . . .	26
5.5.3	System Testing . . . . .	26
5.5.4	Operational Acceptance Testing . . . . .	27
5.6	Testing types . . . . .	27
5.6.1	Functional testing vs. non-Functional testing . . . . .	27
5.6.2	Functional testing . . . . .	27
5.6.3	Non-Functional testing . . . . .	28
5.7	Automated testing . . . . .	29
<b>6</b>	<b>Software Metrics</b>	<b>29</b>
6.1	What is a Metric? . . . . .	30
6.1.1	Metric . . . . .	30
6.2	Purpose of the Metrics . . . . .	31
6.3	Types of Metrics & Common Software Measurements . . . . .	31
6.4	Goal-Question-Metric (GQM) Paradigm . . . . .	32
<b>7</b>	<b>Dashboarding</b>	<b>35</b>
7.1	Benefits of using a dashboard . . . . .	35
7.2	Dashing . . . . .	36
<b>8</b>	<b>Project Overview</b>	<b>39</b>
<b>9</b>	<b>Architecture</b>	<b>41</b>
9.1	Data Sources . . . . .	42
9.2	Data Layer . . . . .	42
9.3	Data Base . . . . .	42
9.4	Business Logic . . . . .	43
9.5	Dashboard - User Interface . . . . .	43
<b>10</b>	<b>Systems Overview</b>	<b>43</b>
10.1	Sonar Qube . . . . .	44
10.2	JIRA . . . . .	46
10.3	SVN . . . . .	47
10.4	Jenkins . . . . .	48
10.5	Test Report . . . . .	49

<b>11 Data Layer</b>	<b>51</b>
<b>12 Dashboard</b>	<b>55</b>
12.1 Architecture . . . . .	55
12.2 Dashboard . . . . .	56
12.3 Anatomy of a widget . . . . .	56
12.4 Job . . . . .	58
<b>13 Final Implementation: Metrics - Widgets</b>	<b>60</b>
13.1 JIRA Progress . . . . .	60
13.2 Code Changes . . . . .	61
13.3 Test for Builds rate . . . . .	62
13.4 Daily Builds . . . . .	63
13.5 Jenkins Builds History . . . . .	64
13.6 NLOC . . . . .	65
13.7 JUnit Tests Rate . . . . .	66
13.8 Timeline . . . . .	67
13.9 Ticker Bar . . . . .	68
<b>14 Conclusions</b>	<b>69</b>
<b>15 Future prospect</b>	<b>69</b>





# List of Figures

- 2.1 Iron Triangle for Project Management. . . . . 6
- 2.2 Four P's of Software Eng. Activities. . . . . 7
- 2.3 Common Life-Cycle in Software Projects. . . . . 10
  
- 3.1 Codeline: sequence of versions. . . . . 14
- 3.2 Baseline. . . . . 15
- 3.3 Mainline. . . . . 15
  
- 4.1 Coupling. Tight/loose coupling. . . . . 20
  
- 5.1 The Relative Cost of Fixing Defects. . . . . 22
- 5.2 Box Testing Approach. . . . . 25
  
- 6.1 THE GOAL QUESTION METRIC APPROACH - Victor R. Basili. . 33
  
- 7.1 Dashing Dashboard Sample. . . . . 36
  
- 8.1 Software production work-flow diagram. . . . . 40
  
- 9.1 Application architecture diagram. . . . . 41
- 9.2 Data Base Example. . . . . 43

## LIST OF FIGURES

10.1 Sonar Qube Dashboard Sample. . . . .	45
10.2 Jira sample query . . . . .	46
10.3 Jira Open Issue workflow diagram. . . . .	47
10.4 Jenkins main page. . . . .	48
10.5 Jenkins simple work-flow. . . . .	49
10.6 Reval Test Report Interface. . . . .	50
11.1 Data Layer Architecture Diagram . . . . .	51
11.2 Crontab application file. . . . .	53
12.1 Business Logic and UI Architecture diagram. . . . .	55
12.2 Result of the sample widget explained above. . . . .	58
13.1 JIRA Progress Widget. . . . .	60
13.2 Code Changes Widget. . . . .	61
13.3 Test for Builds rate Widget. . . . .	62
13.4 Daily Builds Widget. . . . .	63
13.5 Jenkins Builds History Widget. . . . .	64
13.6 NLOC Widget. . . . .	65
13.7 JUnit Tests Rate Widget. . . . .	66
13.8 Timeline Widget. . . . .	67
13.9 Tickbar news Widget. . . . .	68

# List of Tables

5.1 Static v.s Dynamic testing approach. . . . . 24

11.1 Main module - Data Source System relation. . . . . 52



# Abbreviations

The following table describes the significance of various abbreviations and acronyms used throughout the thesis. The page on which each one is defined or first used is also given. Nonstandard acronyms that are used in some places to abbreviate the names of certain white matter structures are not in this list.

SaaS: Software as a Service

SWENG: Software Engineering

SQ: Sonar Qube

SVN: Social Venture Network - Apache Subversion

KPI: Key Performance Indicators

MOI: Motivation, Organization, Ideas or Innovation

SCM: Software Configuration Management

VC: Version Control

CI: Configuration Item

OS: Operating System

IEEE: Institute of Electrical and Electronics Engineers

ISTQB: International Software Testing Qualifications Board

CPU: Central Processing Unit

API: Application Program Interface

DSL: Domain-Specific Language GUI: Graphical User Interface

## *LIST OF TABLES*

SWEBOK: SoftWare Engineering Body Of Knowledge

QA: Quality Assurance

GQM: Goal-Question-Metric

TS: Treasury Services

UX: User Experience

SWAT G: SoftWare Action Team Graz

BL: Business Logic

UI: User Interface

XML: Extensible Markup Language

JSON: JavaScript Object Notation

ID: Identification Data

DB: Data Base

SQALE: Software Quality Assessment Based on Lifecycle Expectations

DOM: Document Object Model

NLOC: Number Lines Of Code

# 1 Introduction

Following the statement “If you can’t measure it, you can’t manage it” – *W. Edwards Deming*, companies use metrics and measurement systems to monitor and control the status of their projects and products. A successful measurement system must be designed and developed based on company policies and strategies in order to overcome the challenges with overwhelming information generated by software applications for supporting decision making, prediction of events and product tracking.

Software Measurement is playing an increasing important role in software engineering and sooner or later it will become a truly engineering discipline. Thus, this research gives an overview over Software Development. It is divided in two main sections.

On one hand, we have the theory of Software Engineering that explains the principal topics of SW production life-cycle. The purpose of this part is to give an overview of the Software Engineering to understand the development part later on. So it will cover software Basics, Configuration Management, Design, Testing and software metrics.

The second part of this dissertation will be about the development of software that is based on this theory discussed in the first part. Development of customized application implemented to analyze the company development performance. All in all, the mentioned system fetches data from different environments (such as JIRA, SVN, Test Results, Jenkins, SQ...), analyzes the KPI’s and show the results in a simple on-line dashboard. The final user will be the internal stakeholders of the company, but focus in the middle management, so they can follow software production performances of the company overall or individual squads/teams. The implemented system which analyzes the development KPI’s has a Data Sources => Data Layer => Data Base => Business Logic => User Interface architecture. For this, multiple programming languages and frame-tools have been used which will be explained further down below.

## 1.1 Motivation

This last year, during my exchange, I had Software Engineering as a subject in which I had to program, and at the same time, this opened my eyes to what surrounds software development. From stakeholders to software processes, from Scrum teams to Black-Box testing, and so on. While discovering all this basics of software engineering I was looking for an internship in Austria. So in February I joined to Reval Holdings INC, an American SaaS (Software as a Service) company which provides treasury and risk management on-demand software.

Inside the company, the Quality Assurance department suggested to keep track of the life-cycle of its product (software) with KPI's to a posteriori show them up in a on-line Dashboard. In that way, all the internal stakeholders could see how the product performs. Thus, it become as part of my research thesis and my effort for the following four months.

## 1.2 Objective

The goal of this thesis is divided into two unrelated parts. Firstly, the requirements that the application has to follow and secondly the repercussion of the metrics on the company.

In the production of the application that is developed, the target will be behind the key software goals. Those are functionality, usability, maintainability, flexibility and efficiency. One of the main requirements that the company wants to focus on is maintainability of the application. It is important since the versions, environments and some values change from time to time.

From the metrics side, the goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

What it means is that all the internal stakeholders have the possibility to track various parameters in each phase of the development cycle at glance with a dashboard. Acquiring more transparency about the progress and efforts of each team, thus finding room for improvement.



## 1.3 Manuscript organization

The aim of the first part is a general overview of Software Engineering. Also, the objective of it is to give the reader a framework to understand the practical side of this thesis. So the beginning of this paper will cover the very fundamentals of Software Engineering, Software Management, Design and Testing, and finally discuss Metrics and Data Visualization. It will give another perspective to software engineers, allowing them to think about the wider implications of their work.

The second part of the thesis is a summary of the development of this project. The architecture used to develop the application will be displayed with short descriptions. At the same time, the data sources used to obtain the information will also be explained. Then the thesis will cover two of the main parts of the application; explanation of the internal working of the application (Data Layer) and then how the user interface (the Dashboard) is built. Finally the final implementation will be explained in more detail.

## 2 Software Engineering Basics

First of all, it will be necessary to make a slight introduction to the software engineering area. It will cover the definition of SWENG, motivation and goals, activities and you will see why a project fails or succeeds, and finally the Four "P's" of Software Development.

### 2.1 What is Software Engineering?

#### Definition of Software Engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. It covers all aspects of software production, not just technical process of development of tools, methods etc. to support software production. –Not only *WHAT* but also *HOW*.

So the Software Engineering goal is the creation of software systems that are:

- Meeting the customers' needs
- Reliable
- Efficient
- Maintainable
- Produced economically
- Meeting the schedule and budgets

## Software Process Activities

The systematic approach that is used in software engineering is called software process. It is a sequence of activities that leads to the production of a software product. There are four basic activities that are shared along any software production. These activities are:

- Software specification: Where customers and engineers define the software that is produced and the constraints on its operation.
- Software development: Where the software is designed and programmed.
- Software validation: Where the software is checked to ensure that it is what the customer requires.
- Software evolution: Where the software is modified to reflect customer changes and market requirements.

## 2.2 Why Software Fails or Succeeds?

There are multiple reasons why a Software Project is unsuccessful: an unsuccessful project is one that does not meet the expectations. Failing to meet just one of the following objectives can cause a project fail:

- Over budget.
- Doesn't meet stated customer requirements.
- Lower quality than expected.
- Performance doesn't meet expectations.
- Too difficult to use.

An approach is the **Project Management Triangle** or **Iron Triangle** is a model of the limitations of Project Management: Scope, Time and Cost. It is useful to help choose project biases or analyze the goals of a project. Thus, it is used to illustrate that Project Management success is measured by the project team's ability to manage the project, so that the expected results are produced while managing time and cost. Changing one of these three key constraints will most likely affect

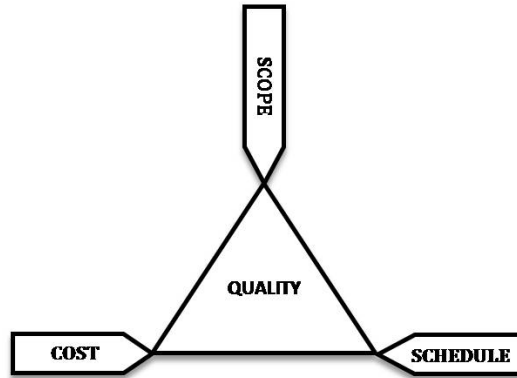


Figure 2.1: Iron Triangle for Project Management (Cost, Scope and Schedule).

the others, or impact the quality of the project. Software development project often fails because the organization sets unrealistic goals for the "Iron Triangle" of software development. By breaking the Iron triangle, you often: Cancel the project (>15%), Deliver late and/or over budget (50%), Deliver poor quality software or Underdeliver. The solution would be, depending on the case, to vary the scope (time boxing), vary the schedule or vary the resources. But, remember, nine women can't deliver a baby in one month. [Wik08]

## 2.3 Software Engineering Activities

The production of software systems can be extremely complex and has many challenges. Systems, particularly large ones, demand the coordination of many *people*, called stakeholders, who must be organized into teams or squats and whose main goal is to build a *product* that satisfies the defined requirements. They make a very good and thorough effort which has to be organized into a characterized *project*, with the purpose of success. A framework is needed within which the teams carry out the activities to build the product, this is called *process*.

So, the effective project management is broken down into four P's: people, product, project and process. Communication and collaboration is all about people. The product must address the correct problem. A sound process keeps the project efficiently moving forward, and a project plan provides a road map to success.



Figure 2.2: Four P's of Software Eng. Activities (People, Product, Project and Process).

### 2.3.1 PEOPLE

People are the first and most important element of any successful project and identifying how people impact a project is crucial. The following categories of people are involved in the software process (stakeholders):

- **Executive leadership (Business managers):** Define business issues which influence the project.
- **Project managers:** Plan, motivate, organize and lead a development team.
- **Development teams:** Programmers, designers, analysts and many other specialists.
- **Customers:** Purchase the software, specify the requirements.
- **End-users:** Interact with the final product.

And between them all, it is essential to coordinate and communicate. Communication can be interpersonal or between systems that need to speak to each other to get a desired result. Documentation, meetings, phone calls, email and even Wiki's are forms of communication that can be found among project teams. Depending on the development or production model (agile, waterfall or standard) there are different ways to use any of these tactics to reach the project goal.

Project Manager – MOI model of leadership

---

Project managers are responsible for planning and tracking a project. They are involved throughout, managing the people, process, and activities. They continuously monitor progress and proactively implement necessary changes and improvements to keep the project on schedule and within budget.

According to the MOI model of Leadership, a Project Manager has to broadcast:

- **Motivation**  
The ability to encourage technical people to produce to their best potential.
- **Organization**  
The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
- **Ideas or innovation**  
The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

### 2.3.2 PRODUCT

The products of a software development effort consist of much more than the source and object codes (software to be built). It also includes the so called 'Artifacts':

- Project documentation (Requirements, Design, ...)
- Source Code
- Test Documents (Plans and Results)
- Customer Documentation (Installation- and User-guide)
- Productivity measurements (input vs. output)

Depending on the product there are two types. *Generic products type*: Stand-alone systems that are marketed and sold to any customer who wishes to buy them. *Customized products type*: Software that is commissioned by a specific customer to meet their own needs.

But there are two different products depending on the specification. *Generic products by specification*: Where the specification of what the software should do is owned by the software developer and decisions on software changes are made by the developer. *Customized products by specification*: In this case, the specification of what the software should do is owned by the customer for the software and they are the ones making decisions on software changes that are required.

### 2.3.3 PROJECT

A software project defines the activities and associated results needed to produce a software product:

- **Planning**: Plan, monitor, and control the software project. Cost estimation of human, hardware and software resources. Determining the feasibility of project and planning information often revisited → Software Project Management Plan.
- **Requirements analysis**: Define what to build. Customer needs are gathered and understood, specific product functions, customer interviews and brainstorming sessions. The requirements describe the WHAT → Software Requirements Specification.
- **Design**: Describes how to build the software. Multiple internal software structure → Software Design Document.
- **Implementation**: Program the software and integrate the software parts.
- **Testing**: Validate that software meets the requirements; code correctness is tested. Testing modules (Unit-testing), interfaces (Integration-testing) and the entire system (Acceptance-testing).
- **Maintenance**: Repair of software defects after release, customer requests for enhancements, performance or reliability improvements. Resolve problems and adapt software to meet new requirements)

### 2.3.4 PROCESS

A software process is a framework for carrying out the activities of a project in an organized and disciplined manner. It imposes structure and helps guide the many people and activities in a coherent manner. A software project progresses through different phases, each interrelated and bounded by time. A software process expresses the relationship between the phases by defining their order and frequency, as well as defining the deliverability of the project. Figure 2.3 names the major phases and indicates the order in which they are usually performed:

- **Specification** – defining what the system should do.
- **Design and Implementation** – defining the organization of the system and implementing the system.
- **Validation** – checking that it does what the customer wants.
- **Evolution** – changing the system in response to changing customer needs.

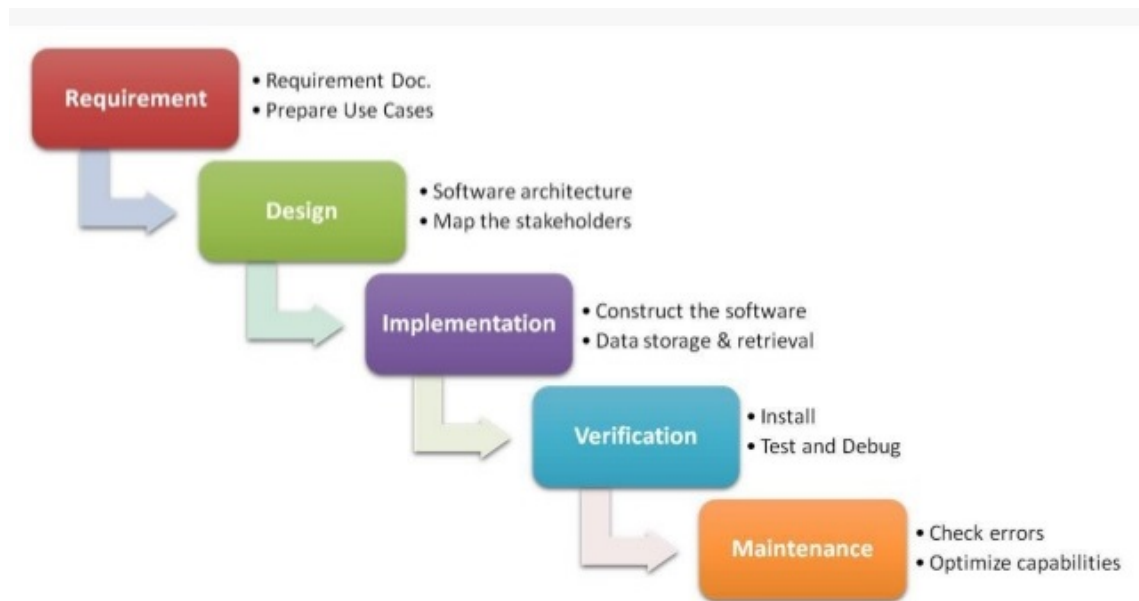


Figure 2.3: Common Life-Cycle in Software Projects.



**The waterfall model:** This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on.

**Incremental development:** This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.

**Reuse-oriented software engineering:** This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

## 3 Software Configuration Management

This chapter will help to define Software Configuration Management in simple terms as the mechanism used to control the evolution of software projects. An understanding of what we mean by Software Configuration Management (SCM) is crucial because if we don't know what we want to do, we have no hope of converging in a good Software-Development environment. So, on this section we will discuss some of the Software Development best practices. Also it will introduce the main concepts of the SCM goals, systems and processes that are used to implement those best practices.

### 3.1 What is Software Configuration Management?

Configuration management is the management of an evolving software system. Because software changes frequently, systems can be thought of a set of versions, each of which has to be maintained and managed. SCM is the task of tracking and controlling changes in the software. SCM practices include revision control and the establishment of baselines. If something goes wrong, SCM can determine what was changed and who changed it, according to [Pea06].

In summary, when multiple people are working on a given project, for each project there are different versions of documents/software; and there are also different releases to customers. Thus, a SCM is needed because it is easy to lose track of what changes and component versions have been incorporated into each system version.

### 3.2 Configuration Management Processes

1. **Change management** This involves keeping track of requests for changes to the software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
2. **Version management** This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.

3. **System building** This is the process of assembling program components, data, and libraries, and then compiling and linking these to create an executable system.
4. **Release management** This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Regarding SCM processes all of them are important. But in this section we will focus mainly on the Version Control (VC) Processes.

## 3.3 SCM Terminology

A brief definition of the most common/used terms in VC. These are: Configuration Item (CI), Version, Variant, and Revision, Code-line, Baseline, Mainline and Release, and finally Workspace.

### Configuration Item

Anything associated with a software project. Candidate CI's are:

- Source Code
- Project Specification
- User Documentation
- Test plans and data
- Supporting software: compilers, editors
- Any artifact that will undergo modification or need to be retrieved at some time after its creation.

#### Version

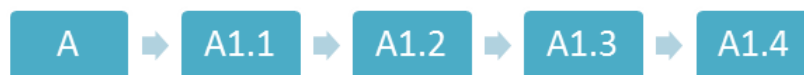
An instance of configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number. New versions of software system are created as they change; Offering different functionality or tailored for particular user requirements.

#### Variant

Functionally equivalent versions, but designed for different settings, e.g. hardware and software, for different machines/OS.

#### Code-line

A code-line is a set of versions of a software component and other configuration items on which that component depends. A code-line is a sequence of versions of source code with later versions in the sequence derived from earlier versions. Code-lines normally apply to components of systems so that there are different versions of each component.



*Figure 3.1: Codeline: sequence of versions.*

#### Baseline

A baseline is a collection of component versions that make up a system. The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc. Baselines are important to recreate a specific version of a component system. For example: You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired. The baselines should be created at the end of each project iteration.

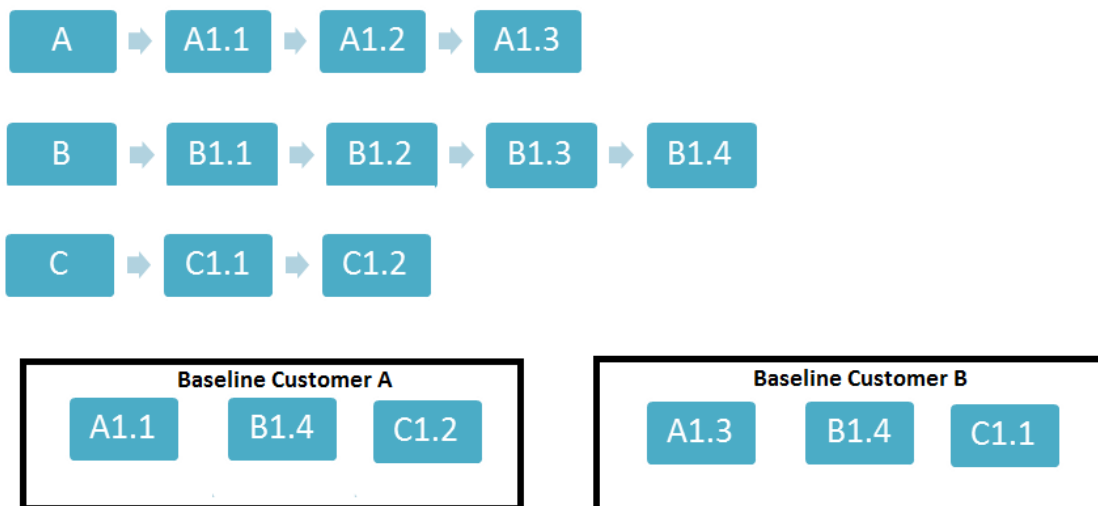


Figure 3.2: Baseline is a collection of component versions that make up a system.

### Mainline

A sequence of baselines representing different versions of a system.

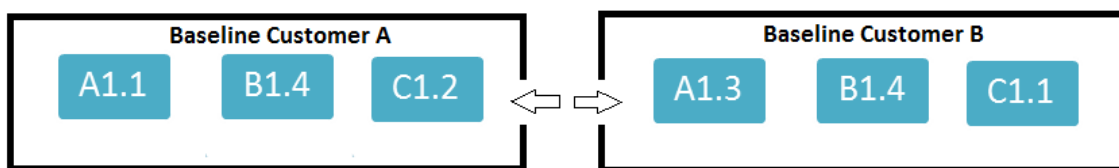


Figure 3.3: Mainline.

### System Release

A System Release is a version of a software system that is distributed to customers.

For mass market software, it is usually possible to identify two types of release:

- **Major Releases** Which delivers significant new functionality.
- **Minor Releases** Which repairs bugs and fixes customer problems that have been reported.

#### Workspace

An isolated environment where a developer can work (edit, change, compile, test) without interfering with other developers. Examples: Local directory under version control, private workspace on the server.

Common Operations:

- Import: put resources into version control in repository.
- Update: get latest version on the default branch.
- Check-out: get a version into workspace.
- Check-in: commit changes to the repository.

## 3.4 Version Management

Version management is the process of keeping track of different version of software components or configuration items and the systems in which these components are used. It also involves ensuring that changes made by different developers to these versions do not interfere with each other. Therefore version management can be thought of as the process of managing code-lines and baselines.

Version management systems normally provide a range of features:

1. **Version and release identification** Managed versions are assigned identifiers when they are submitted to the system.
2. **Storage management** To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities.
3. **Change history recording** All of the changes made to the code of a system or component are recorded and listed.
4. **Independent development** The version management system may support the development of several projects, which share components.
5. **Project support** A version management system may support the develop-

ment of several projects, which share components.

## 3.5 SCM Best Practices

When implementing SCM tools and processes, you must define what practices and policies to employ to avoid common configuration problems and maximize team productivity. Many years of practical experience have shown that the following best practices are essential to successful software development:

- Identify and store artifacts in a secure repository.
- Control and audit changes to artifacts.
- Organize versioned artifacts into versioned components.
- Organize versioned components and subsystems into versioned subsystems.
- Create baselines at project milestones.
- Record and track requests for change.
- Organize and integrate consistent sets of versions using activities.
- Maintain stable and consistent work-spaces.
- Integrate early and often.
- Ensure reproducibility of software builds

## 4 Software Design

This chapter will focus on another one of the main steps of the Software Development life-cycle: the Software Design. It covers architecture design, class design, User interface design, algorithm design and finally protocol design. So in this part the concept of Software Design and its characteristics will be explained and some different concepts to better understand the point of this research will be discussed.

### 4.1 What is Software Design?

Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints. Software design may refer to either "all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems" or "the activity following requirements specification and before programming, as ... [in] a stylized software engineering process." [[Wik13b]].

A "software design" is a representation or model of the software to be built. Its purpose is to enable programmers to implement the requirements by designating the projected parts of the implementation. It is a set of documents containing text and diagrams to serve as the base on which an application can be fully programmed. A complete software design should be so explicit that a programmer could code the application from it without the need for any other documents. They can be understood in two parts: high-level design, often referred to as "software architecture," which is generally indispensable, and all other design, referred to as "detailed design.", according to the book [Som09a, Chap. 15].

### 4.2 Goals/Characteristics of Software Design

Now that the definition and purpose of Software Design are clear, the next step is to define the goals of this science of Software Design. Every software design plan should cover:



- **Correctness:** does what it should, faces the requirements?
- **Understandability:** understood by intended audience.
- **Modularity:** divided into well-defined parts.
- **Cohesion:** like-minded elements are grouped.
- **Coupling:** minimize dependence between elements.
- **Robustness:** can deal with wide variety of inputs.
- **Flexibility:** adaptable to shifting requirements.
- **Re-usability:** use parts in other applications.
- **Information hiding:** module internal hidden from others.
- **Efficiency:** executes within acceptable time/space.
- **Reliability:** executes with acceptance failure rate.

## 4.3 Software Design Terminology

### 4.3.1 Component

Refers to a component any software or hardware that has a clear role. A component can be isolated, allowing the programmer to replace it with a different component that has equivalent functionality.

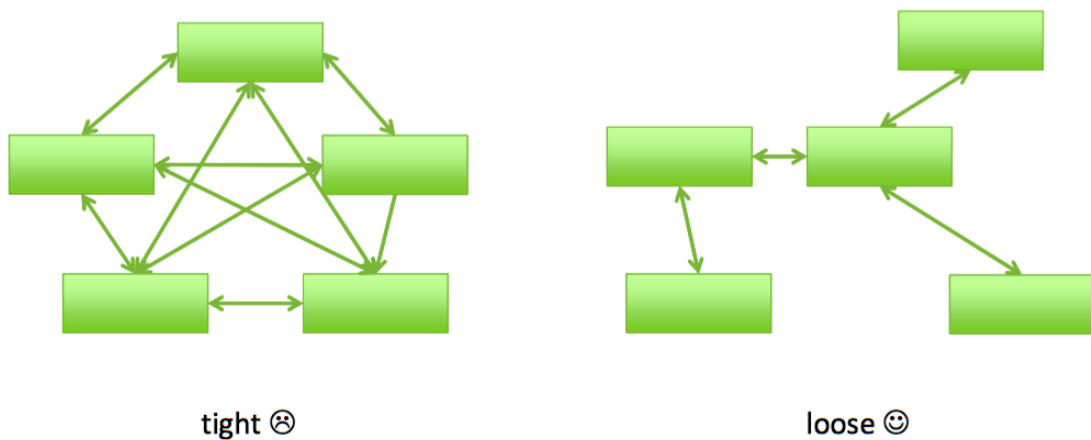
### 4.3.2 Coupling

Coupling is a measure of the interdependences of components and of the extent to which a module (package, class, method) relies on another module. It is an indication of the strength of interconnection between the components in a design.

*Tight coupling (equal bad):* when a group of classes are highly dependent on one another. It makes modifying parts of systems difficult.

*Loose coupling:*

- Minimal dependences of the method on the other parts of the source code
- Minimal dependences on the class members or external classes and their members
- No side effects
- If the coupling is loose, we can easily reuse a method or group of methods in a new project



*Figure 4.1: Coupling. Tight/loose coupling.*

### 4.3.3 Cohesion

In computer programming, cohesion refers to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationship between pieces of functionality within a given module. Cohesion and coupling are closely related. Usually higher Cohesion results in lower Coupling (and vice versa) [[Wik04a]].

There is 7 types of cohesion:

- **Co-incident cohesion (worst form of cohesion)**

Unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization, e.g. Utils- or

Helper-Classes.

- **Logical cohesion**

Logically categorized elements are put together into a module.

- **Temporal cohesion**

Elements of a module are organized such that they are processed at a similar point in time.

- **Procedural cohesion**

Elements are grouped together, which are executed sequentially in order to perform a task.

- **Communicational cohesion**

Elements of a module are grouped together, which are executed sequentially and work on same data (information).

- **Sequential cohesion**

Elements of module are grouped because the output of one element serves as input to another.

- **Functional cohesion (best form)**

It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

# 5 Software Testing

This section is the last on the life-cycle of SW Development before the release. Thus, it is really important that the SW passes this phase. In this chapter, the testing will be explained, as well as the differences between verification and validation. The "defect" will be defined, and multiple testing practises, levels, types, processes and finally automated testing will be summarized and schematized.

## 5.1 IEEE - Definition of Testing

*"The process of exercising or evaluating a system by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results." - IEEE Definition of Testing*

### "Defect"

According to the [Cer13a], a defect is an error or a bug in the application which is created. A programmer, while designing and building the software, can make mistakes and error. These mistakes or errors mean that there are flaws in the software.

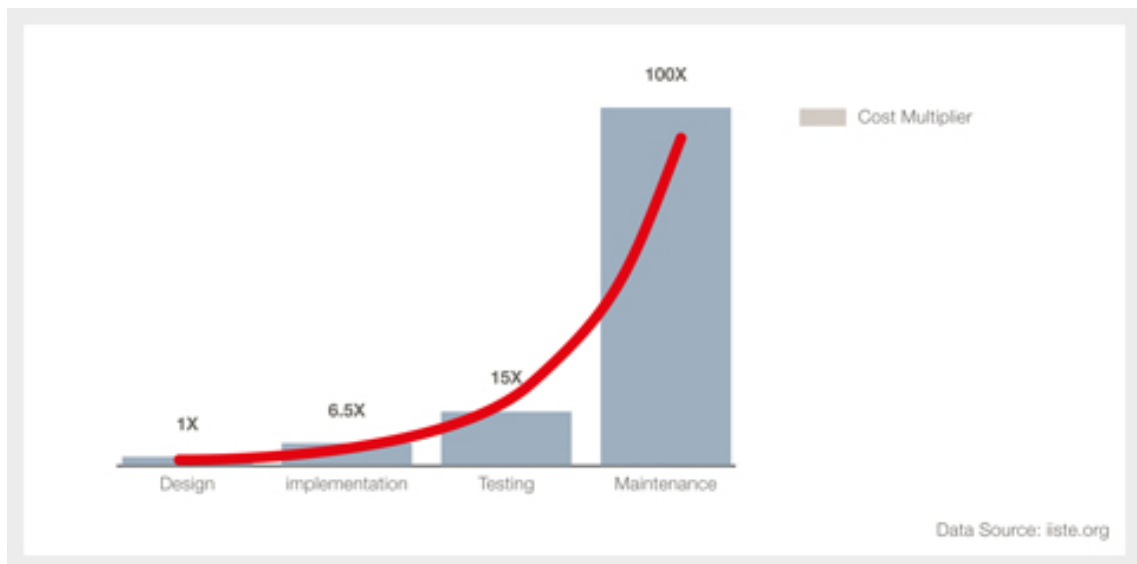


Figure 5.1: The Relative Cost of Fixing Defects.

## 5.2 Testing Goals

Testing is intended to show that a program does what it is intended to do and to discover program defects before the SW is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification.

## 5.3 Verification vs. Validation

So, SW testing is a process of validating and verifying that a software program or application or product meets the business requirements and works as expected. But verification and validation are not the same thing, although they are often confused.

### **Verification**

- Have we build the software right?
- Does it match the specification?

### **Validation**

- Have we build the right software?
- Is this what the customer wants?

## 5.4 Testing methods

There are several approaches of Software Testing: **Static v.s Dynamic testing approach** and **The Box approach**.

### 5.4.1 Static testing

Is performed in a non-runtime environment. Under Static Testing code is not executed. Rather it manually checks the code, requirement documents, and design documents to find errors.

### 5.4.2 Dynamic testing

Under Dynamic testing code is executed. It checks for:

- functional behavior of software system,
- memory/CPU usage and
- overall performance of the system.

It involves working with the software, giving input values and checking if the output is as expected by executing specific test cases which can be done manually or with the use of an automated process.

Static testing	Dynamic testing
Testing done without executing the program	Testing done by executing the program
Static testing is about prevention of defects	Dynamic testing is about finding and fixing the defects.
Static testing involves checklist and process to be followed	Dynamic testing involves test cases for execution.

Table 5.1: Static v.s Dynamic testing approach.

### Black Box Testing

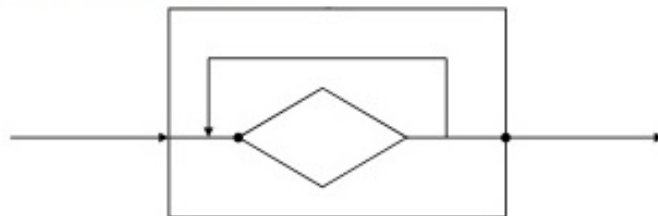
Treats the software as a "black box", examining functionality without any knowledge of internal implementation. It includes: equivalence partitioning, boundary value analysis, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory testing and specification-based testing.

## Black Box Testing v.s White Box Testing

### *Black Box Testing*



### *White Box Testing*



*Figure 5.2: Box Testing Approach.*

The tester is only aware of what the software is supposed to do, not how it does it. It also known as **Specification-based testing technique** or input/output driven testing technique because it views the software as a black-box with inputs and outputs.

### White Box Testing

White-box testing tests internal structures or workings of a program. It includes multiple testing methods: API testing, code coverage, fault injection, mutation testing, static testing. Here the testers require knowledge of how the software is implemented. In white-box testing the tester is concentrating on how the software operates.

## **Visual (GUI) Testing**

Is the process of testing a product's graphical user interface to ensure it meets its written specifications like testing images and buttons alignment on any web page.

## **5.5 Testing levels**

There are generally four recognized levels of tests: unit testing, integration testing, component interface testing, and system testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by the SWEBOK guide are unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model. Other test levels are classified by the testing objective, as stated in [Wik01b].

### **5.5.1 Unit Testing**

Testing parts (unit) of an application in isolation (separately). A unit is the smallest testable part of an application like functions/procedures/methods, classes, interfaces. Unit testing is a method by which individual units of source code together with associated control data are tested to determine if they are fit for use. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended [[Wik01c]].

### **5.5.2 Integration Testing**

Integration testing is the phase in which individual software modules are combined and tested as a group to verify that the integrated system is ready for system testing. It occurs after unit testing [[Wik03]].

### **5.5.3 System Testing**

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner



design of the code or logic. System testing is most often the final test to verify that the system to be delivered meets the specification and its purpose [[Wik04b]].

### 5.5.4 Operational Acceptance Testing

After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing. Acceptance testing is a test conducted to determine if the requirements of a specification or contract are met prior to its delivery. Acceptance testing is basically done by the user or customer although other stakeholders may be involved as well [[Cer13b]].

## 5.6 Testing types

### 5.6.1 Functional testing vs. non-Functional testing

Functional testing is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding input and examining the output, and internal program structure is rarely considered (not like in white-box testing). Functional testing usually describes what the system does [Wik06b].

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or other performance, behavior under certain constraints, or security. Testing will determine the breaking point, the point at which extremes of scalability or performance lead to unstable execution. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users.

### 5.6.2 Functional testing

#### *Smoke Testing*

Smoke Testing is preliminary testing to reveal simple failures severe enough to reject a prospective software release. e.g. a smoke test may ask basic questions like "Does the program run?","Does it open a window?". The purpose is to determine the degree of necessary repair to the application and to evaluate if further testing should

## 5 Software Testing

be done [[Wik13a]].

### *Regression Testing*

Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, in existing areas of a system after changes have been made to them. Common methods of regression testing include rerunning previously completed tests and checking whether program behavior has changed and whether previously fixed faults have reemerged [[Wik02]].

### *Destructive Testing*

Destructive software testing attempts to cause a piece of software to fail in an uncontrolled manner, in order to test its robustness. It verifies that the software functions properly even when it receives invalid or unexpected inputs, thereby establishing the robustness of input validation and error-management routines [[Wik01a]].

### *Recovery Testing*

Recovery testing is the activity of testing how well an application is able to recover from crashes, hardware failures and other similar problems. Example: While an application is receiving data from a network, unplug the connecting cable, wait and plug in again [[Wik06c]].

## 5.6.3 Non-Functional testing

### **Compatibility Testing**

Compatibility testing, part of software non-functional tests, is testing conducted on the application to evaluate the application's compatibility with the computing environment. As is said in [Gur16], a computing environment may contain:

- different OS types (IOS, Android, Linux, Windows, ...)
- different types of browsers (Chrome, Firefox, IE,...)

## Software Performance Testing

Performance testing is generally executed to determine how a system or sub-systems performs in terms of responsiveness and stability under a particular workload. It includes three different testings:

- **Load Testing** is a testing that the system can continue to operate under a specific load, that be large quantities of data or a large number of users.
- **Volume testing** is a way to test software functions even when certain components (for example a file or database) increase radically in size.
- **Stress testing** is a testing beyond normal operational capacity, often to a break point in order to observe the results.

## 5.7 Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system, according to [Gur16].

While automation cannot reproduce everything that a human can (and the ways how humans think), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

### System building

The process of compiling the components or units that make up a system and linking these with other components to create an executable program. System building is normally automated so that recompilation is minimized. This automation may be built into the language processing system (as in Java) or may involve software tools to support system building, as stated by [Som09b].

# 6 Software Metrics

This section covers the difference between measurements and metrics and the purpose of metrics, based on the book [Ken07] wrote by Dave Nicolette.

## 6.1 What is a Metric?

Before defining a metric it's necessary to first distinguish metrics from measurements [Nic12].

A **measurement** is a quantitative observation of one of the following:

- Something relevant to the decisions you have to make.
- Information you have to report regarding the progress of development.
- The effects of process improvements.

### 6.1.1 Metric

A metric is a recurring measurement that has informational, diagnostic, motivational, or predictive power of some kind. It helps understand the risk of missing expected results, or whether changes in processes or practices are resulting in improved performance.

Motivations for using metrics in software engineering

- Inform stakeholders.
- Report measurements so that stakeholders can understand activities and results.
- Promote the value of the organization.
- Determine the best way to communicate the information to the stakeholders.
- Perform better stakeholder analysis to facilitate stakeholders buy-in.

- Improve performance - people do what is measured.

## 6.2 Purpose of the Metrics

"When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginnings of knowledge but you have scarcely in your thoughts advanced to the stage of Science."

–Lord Kelvin (Physicist)

"You cannot control what you cannot measure."

– Tom DeMarco (Software Engineer)

At Reval, metrics are used for two purposes: to help control the direction of work in progress and to help monitor the effectiveness of process-improvement efforts.

Metrics have three functions or effects: informational, diagnostic, and motivational. Any metric can perform more than one of these functions simultaneously, and oftentimes the motivational effect is inherently covered without overt intention.

## 6.3 Types of Metrics & Common Software Measurements

Measurement is done using metrics. Three parameters are measured: process measurement through process metrics, product measurement through product metrics, and project measurement through project metrics.

Process metrics assess the effectiveness and quality of software processes, determine the maturity of the process, effort required in the process, effectiveness of defect removal during development, and so on. Product metrics is the measurement of work product produced during different phases of software development. Project metrics illustrate the project characteristics and their execution.

In software engineering, there are three kinds of entities and attributes to measure

1. **Processes** are a collection of software-related activities. A process is usually associated with some timescale. The timing can be explicit, as when an activity must be completed by a specific date, or implicit, as when one activity must be completed before another can begin.
2. **Products** are any artifacts or documents that result from a process activity. Products are not restricted to the items that the management is committed to deliver to the customer. Any artifact or document produced during the software life cycle can be measured.
3. **Resources** are entities required by a process activity. The resources that we want to measure include any input for software production. Thus, personnel (individuals or teams), materials (including office supplies), tools (both software and hardware), and methods are candidates for measurement.

## 6.4 Goal-Question-Metric (GQM) Paradigm

The GQM paradigm is based on the theory that all measurement should be goal-oriented. Each measurement collected is stated in terms of the major goals. Questions are then derived from the goals and help to refine, articulate, and determine if the goals can be achieved. The metrics that are collected are then used to answer the questions in a quantifiable manner.

The article [Agi13] defines GQM as a measurement model on three levels:

**Conceptual level (Goal)** A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view and relative to a particular environment.

**Operational level (Question)** A set of questions is used to define models of the object of study and then focuses on that object to characterize the assessment or achievement of a specific goal.

**Quantitative level (Metric)** A set of metrics, based on the models, is associated with every question in order to answer it in a measurable way.

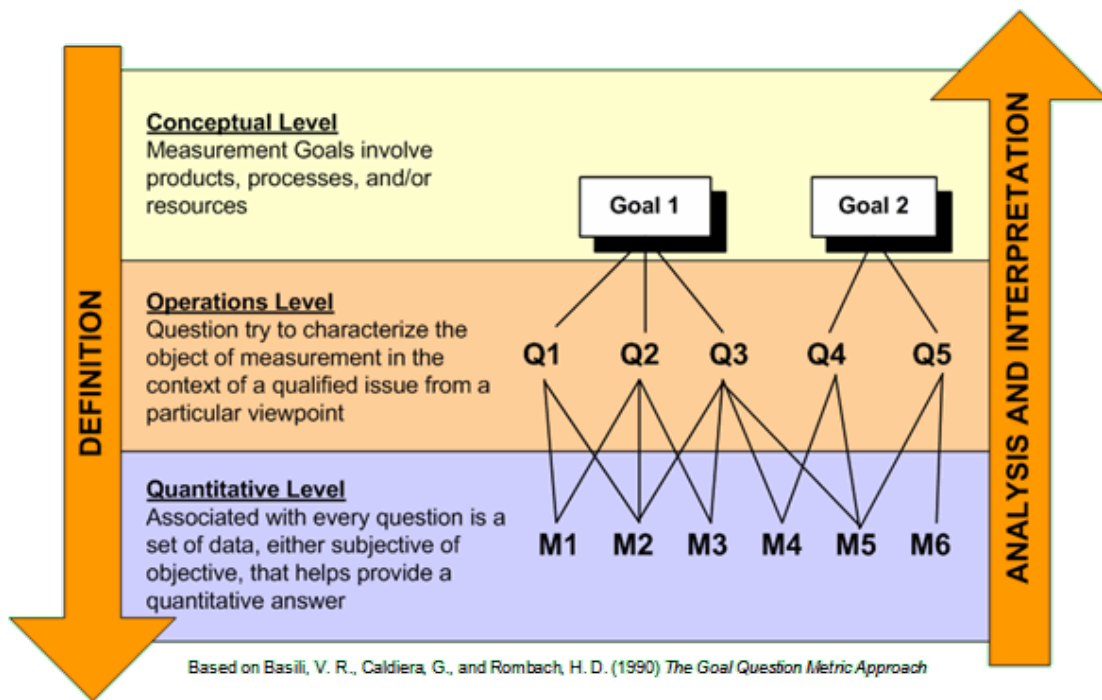


Figure 6.1: THE GOAL QUESTION METRIC APPROACH - Victor R. Basili.





# 7 Dashboarding

This chapter will focus on the creation of a Dashboard for the visualization of the metrics and key performance indicators (KPIs) that were discussed previously. So it will cover the definition and purpose of Dashboard and types of data representation. Also, the frame-tool that is used for this project will be discussed.

## 7.1 Benefits of using a dashboard

As is said in [Wik06a], in the management information systems field, a dashboard is "an easy to read, often single page, real-time user interface, showing a graphical presentation of the current status (snapshot) and historical trends of an organization's or computer appliances key performance indicators to enable instantaneous and informed decisions to be made at a glance." It is composed by one or more widgets; where widgets are small applications with limited functionality, which in this case show different kinds of data (graphs, images, numbers etc.).

Benefits of using digital dashboards include:

- Visual presentation of performance measures
- Ability to identify and correct negative trends
- Measurement of efficiencies/inefficiencies
- Ability to generate detailed reports showing new trends
- Ability to make more informed decisions based on collected business intelligence
- Alignment of strategies and organizational goals
- Time savings compared to running multiple reports

## 7 Dashboarding

- Gain of total visibility of all systems instantly
- Quick identification of data outliers and correlations

### 7.2 Dashing

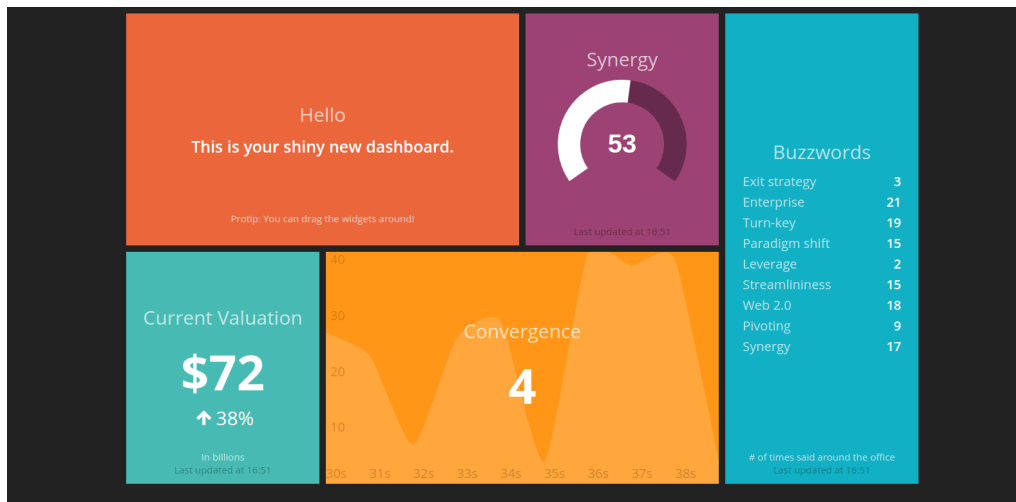


Figure 7.1: Dashing Dashboard Sample.

Dashing is a Sinatra based framework that allows anyone to build personalized dashboards. It was created by the developers of an retail on-line company (Shopify) to keep themselves up-to-date on the performance of their business. It displays the metrics for different teams like Marketing, Business Development and Client Services [[Cai14]]. It is a very easy to use framework ; the internal architecture is divided into Dashboard (html files), Widgets (coffescript, html and scss files) and finally jobs (Ruby files). The main key features are:

- Use pre-made widgets, or custom made widgets with scss, html, and coffeescript.
- Widgets harness the power of data bindings to keep things easy and simple.
- Use the API to push data to the dashboards, or make use of a simple Ruby DSL (Domain-Specific Language) for fetching data.
- Drag & Drop interface for re-arranging widgets

- Dashing is completely customizable. Moreover, since it is an open source project you can find a lot of documentation on the topic.

As explained before, every Dashing project comes with sample widgets and sample dashboards to explore [Wik12]. The directory is setup as follows:

- **Assets:** All images, fonts, and js/coffeescript libraries. Uses Sprockets
- **Dashboards:** One .erb file for each dashboard that contains the layout for the widgets.
- **Jobs:** Ruby jobs for fetching data (e.g. for calling third party APIs like Twitter).
- **Lib:** Optional ruby files to help out the jobs.
- **Public:** Static files that should be served. A good place for a *favicon* or a custom 404 page.
- **Widgets:** All the html/css/coffee for individual widgets.

It comes with multiple predefined and sample widgets, like: Alert, Clock, Graph, Comments, iFrame, Image, List, Meter, Number and Text (the Figure 7.1 displays some of them). Since all the widgets are fully customizable this framework is used to show the metrics.



## 8 Project Overview

The second part of this dissertation focuses on the project I have been developing for four months at Reval Holdings Inc.. It starts with a small introduction which gives a global examination of the companies workload. Then it will cover the project architecture, making small comments on the different parts of it. At the same time the data layer and the user interface will be analyzed. Finally, each part of the dashboard will be presented followed by the conclusion.

Reval is a leading, global Software-as-a-Service provider of comprehensive and integrated Treasury and Risk Management solutions. Reval's cloud-based software provides clients with tools to manage cash and financial risks. The company has different products depending on the client necessities. The software production process at Reval is an slightly changed Scrum model. Here is why: There is a product which has two major releases per year, it means that the life-cycle of each major release is 6 months. During those six months, the company has 4 minor or patch releases. So, 1 major release equals 4 minor releases.

The major release usually indicates very significant changes in the program - for example, if the program has been completely rewritten or new important functions have been added.

The minor releases (also know as patches), usually don't add new features or content, patches are merely revision or bug fix releases.

Each major release (6 months) is divided in six Iterations. The first four are only development. At the end of the 4th Iteration the Feature is completed, that means that the software has all the functionality intended for the final version but is required some improvements and fixes before release. So, at the 5th Iteration the Code is Freeze (is usually at the beginning of the system test phase) and the last Iteration (6th) is the user-beta-testing before the release. At the end of the 6th Iteration a new major starts, it will take again 6 months to complete, an so on.

The same happens in the patch release cycle. A difference of the major release,

## 8 Project Overview

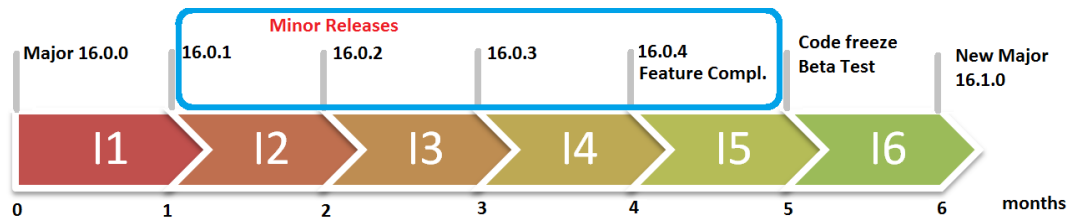


Figure 8.1: Software production work-flow diagram.

the patch takes 1 month each to be completed, and has four iterations: two for development, one featured completed and code freeze too.

So, during those six months the Reval Product Development department is split into two parts. On one side we have Product Engineering which is working constantly on the major version (current 16.1.0) on the other side there is the SWAT team which is in charge of the patch versions (current 16.0.2).

Besides SWAT team, there are others teams inside the company. Those are dedicated to distinct aspects depending on the software modules that they develop. They are TS Business, TS Technology, TS UX, Corp UI, Corp Treasury, Corp Payments, Corp Cash, Platform, Documentation, Build Automation and SWAT G.

It is essential to follow all these processes, iterations, teams/squats and observe how they perform.

# 9 Architecture

The architecture of this project is divided into five parts: Data Sources, Data Layer, Data Base, Business Logic and User Interface. On this project a CentOS machine has been used to host the Data Layer, Data Base and the Dashing (BL & UI) framework. The following diagram shows the global architecture used in this project:

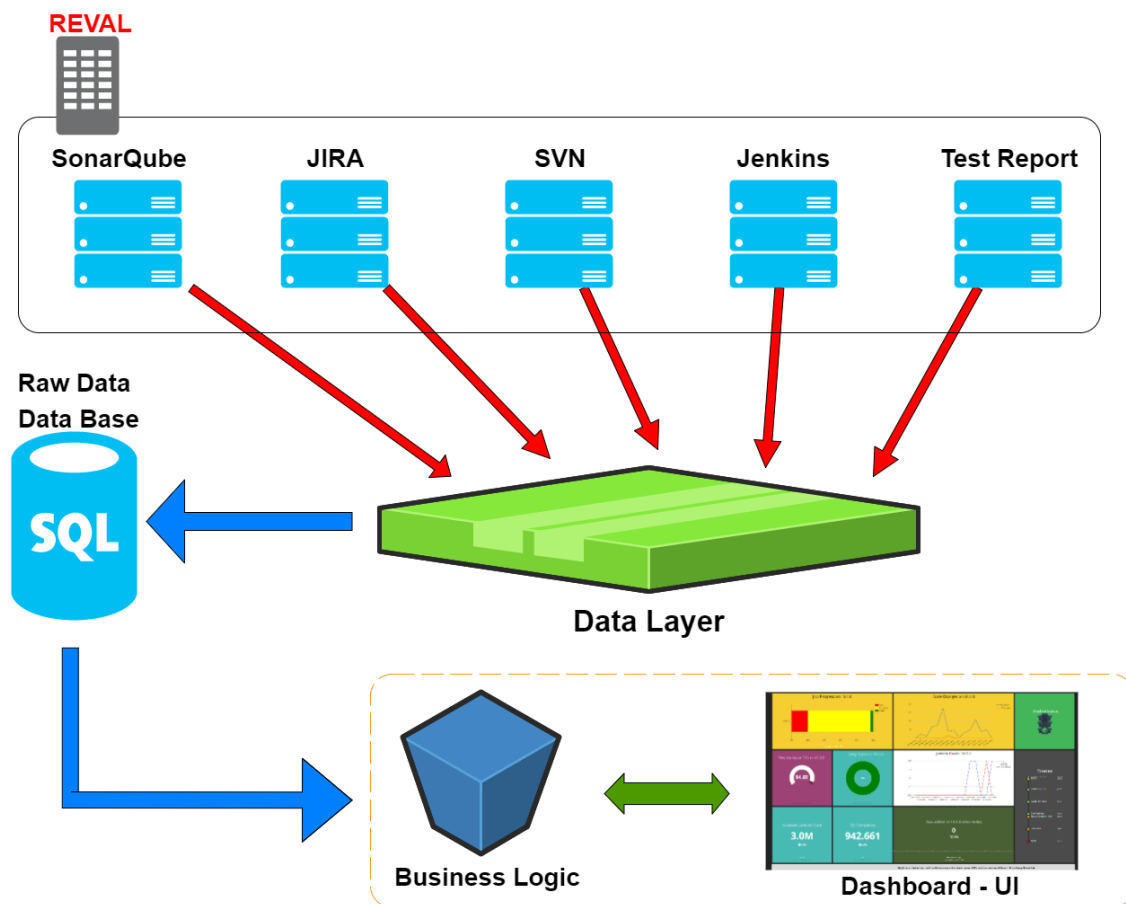


Figure 9.1: Application architecture diagram.

## 9.1 Data Sources

At the top of the diagram the multiple systems that Reval Inc. is hosting are displayed. The company stores the source code there, test reports, tasks management, automation processes, etc. This is the data source of the project. It includes the next systems: SonarQube, JIRA, SVN, Jenkins and Test Report. Each one of them gives useful information about the software production and life-cycle phases.

## 9.2 Data Layer

The Data Layer is of the most important pieces of the architecture. It is Java program build as seven modules and fourteen libraries. Every module is independent (except DB connection and JSON parser) those are DBConnection, SVNSystem, JiraSystem, SQSystem, JenkinsSystem and TestSystem. The purpose of this module is to fetch the data from the company systems and store the "raw data" in the database for further calculation at the Business Logic. Finally, it is important that each module is executed by scheduler (crontab) with different time rates. Most of the data comes with a XML or JSON format, so a decoder has to be implemented for this section.

## 9.3 Data Base

For this project, a simple SQLite database is used. For each source of data a table is created, and it has to have the same structure: ID, NAME, VALUE and TIME. The ID is incremental in all the tables. The name is different depending on the source, the information that contains and its type. The VALUE will always be an integer that represent a measurement or a metric. Finally, the TIME is a time-stamp of when this VALUE was collected from the system.

The following figure shows a series of columns from the JIRA table. As it is explained before, JIRA is used for keep track of Issues or Tasks. The first element that it shows is the NAME, which is composed by the information about the VALUE. For instance, in this case this value is about the version "16.0.2" of the software, the squad/team is "SWAT G" and the type of Issue is "New". Following is the VALUE which in this case is 0. And finally the time-stamp; this was stored in the database at "3:01:05 PM" the "30/05/2016". Thanks to these values historical data for the widgets can be obtained.



```
|16.0.2_SWAT_G_New|0|30/05/2016 3:01:05 PM  
|16.0.2_SWAT_G_Open|1|30/05/2016 3:01:05 PM  
|16.0.2_SWAT_G_In_Analysis|2|30/05/2016 3:01:05 PM  
|16.0.2_SWAT_G_Waiting_On_Response|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_Accepted|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_In_Estimation|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_Pending_Estimate_Approval|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_In_Proposal|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_Pending_Proposal_Approval|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_Request_Definition|0|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_In_Definition|1|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_Request_Development|4|30/05/2016 3:01:06 PM  
|16.0.2_SWAT_G_In_Development|11|30/05/2016 3:01:07 PM
```

*Figure 9.2: Data Base Example.*

## 9.4 Business Logic

This part of the architecture is managed by the Dashing frame tool. As was explained before Dashing has three important elements: Dashboard, Widgets and Jobs. In this case, the jobs are the business logic of this project. What they do is query the necessary data from the DB, process it returning one metric or more metrics and send them to the Dashboard file.

## 9.5 Dashboard - User Interface

At this point the Dashbord is a simple html file. This links the Business Logic with the User Interface. Thus, it is a grid full of widgets that show the metrics calculated at the BL.

# 10 Systems Overview

This last selection gives an overview of the the multiple Data Sources. All the environments are hosted in the internal private network in Reval Headquarters in Graz (Austria). Those are: Sonar Qube, JIRA, SVN, Jenkins and Test Reports. The data from these systems will be fetched and stored in the database by the Data Layer.

Each of these systems controls different aspects of the software production. On one side, there are some that control or analyze the software production (SQ and SVN). At the same time, there are others that focus on the software testing (Jenkins and Test Report) and finally there is JIRA which is used for the whole product life-cycle.

## 10.1 Sonar Qube

SonarQube (formerly known as Sonar) is an open source tool-suite to measure and analyse the quality of source code [Gei14] It is written in Java but is able to analyse code in different programming languages, as: Java (including Android), C/C++, Objective-C, C#, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL, Swift, etc. It offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, potential bugs, comments and design and architecture. Records metrics history and provides evolution graphs ("time machine") and differential views.

It provides fully automated analysis: integrates with Maven, Ant, Gradle and continuous integration tools (Atlassian Bamboo, Jenkins, Hudson, etc.). And integrates with Eclipse, Visual Studio and IntelliJ IDEA development environments through the SonarLint plugins. It also integrates with external tools like: JIRA, Mantis, LDAP, Fortify, etc. At the same time, it can be expanded with the use of plugins. And finally, a great characteristic is that it implements the SQALE methodology to evaluate technical debt [[Wik10]] [Sonar [Doc16]].

Inside the company, Sonar is the latest of the five systems added. The Quality Assurance department is in charge of it, and its updated once a day (in the morning). It could be updated more often, but it would not make sense since the source code does not change that often. This is a very simple frame-tool which doesn't calculate any metrics . It measures them and displays the software development metric. The most common metrics categories that we can find in Sonar Qube are the following:

- Complexity: Based on the number of paths through the code.
- Documentation: Number of lines containing either comment or commented-out code.
- Duplications: Number of duplicated blocks of lines, duplicated files, lines and duplicated lines in %.
- Issues: Number of new issues, count of issues with severity (blocker, critical, major, minor or info), Open issues, etc.
- Quality Gates.
- Reliability: Bugs, Reliability Rating and Effort
- Security: Vulnerabilities, Security Rating, Security remediation efforts, etc.
- Basic Metrics: Lines of Code, Number of classes, directories, files, methods etc.
- Tests: Units tests, Integration tests and Coverage tests.

To obtain the data from Sonar Qube the Rest API provided at the Data Layer will be used. Easy access to the metrics provided by SQ through the rest API which will be used to fetch the data from the DataLayer.

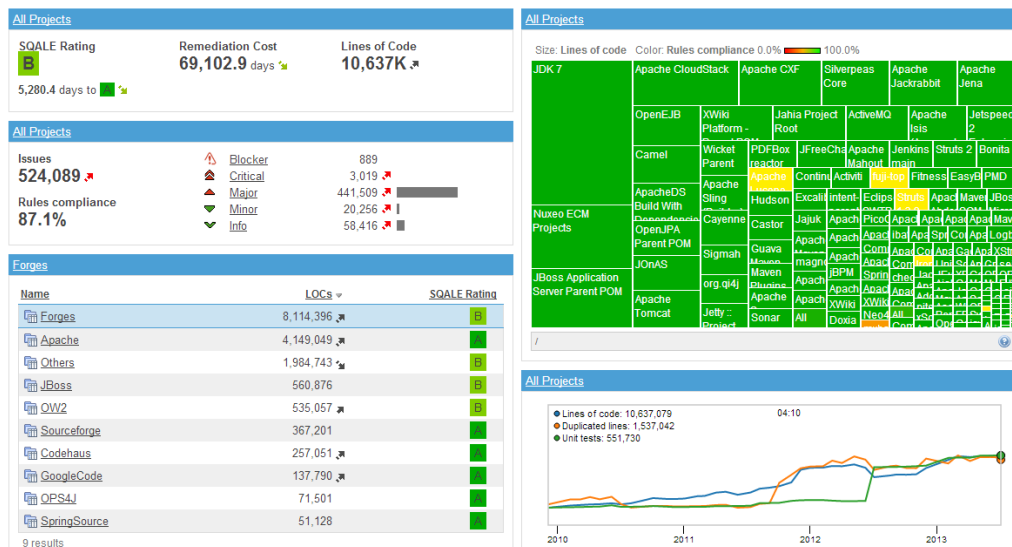


Figure 10.1: Sonar Qube Dashboard Sample.

## 10.2 JIRA

JIRA is an application that can be used to track all issues of a project. JIRA makes the life cycle of issues transparent, and allows for a lot of collaboration. In JIRA, issues can be organized, work assigned, and team activity can be followed through a workflow. One of the benefits of JIRA is that it can be customized to reflect the project elements, the type of issues, and the fields and screens available in each workflow.

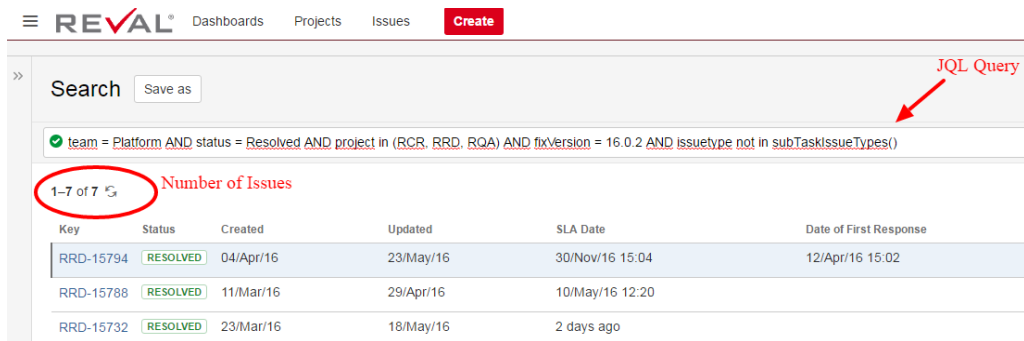


Figure 10.2: Jira sample query.

A JIRA workflow is a set of statuses and transitions that an issue moves through during its life-cycle and typically represents processes within your organization.

Inside the company we have multiple teams, like: TS Business, TS Technology, Platform, SWAT Team, Corp. Cash, etc. and all of the use this tool for issue tracking and ticketing. So it is very useful to obtain information about the performance of each team. To obtain the data from the JIRA system we use the REST API of the framework, as is explained at the [Doc14] in the official JIRA site.

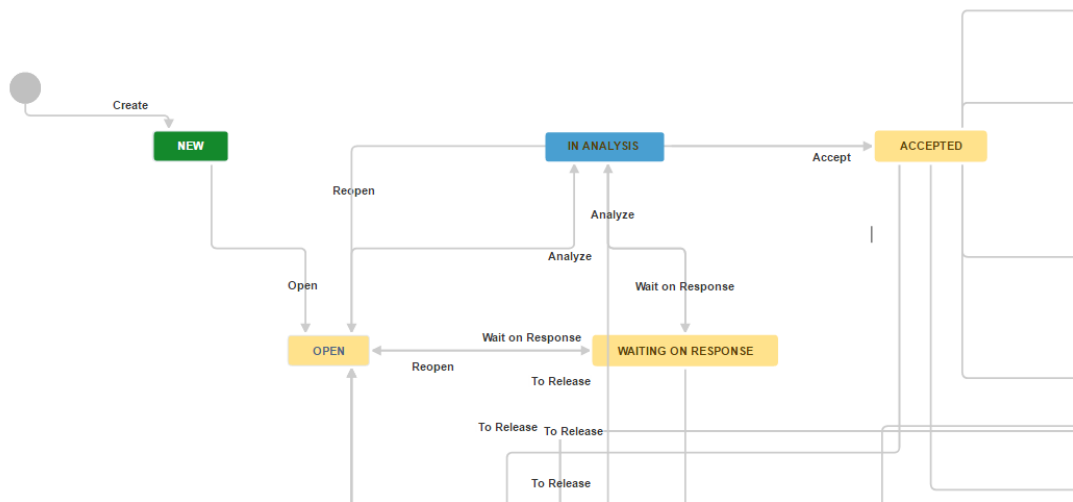


Figure 10.3: Jira Open Issue workflow diagram.

## 10.3 SVN

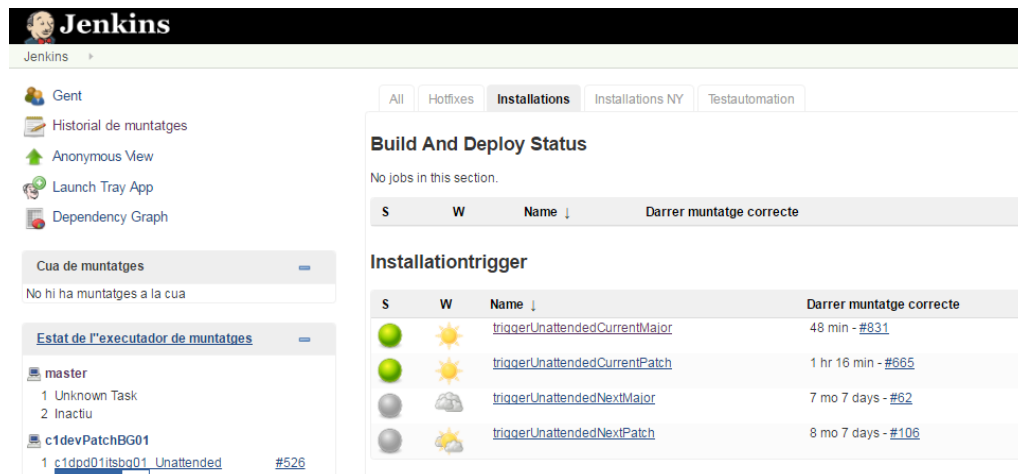
SVN is a shorthand abbreviation of the name “Subversion”. Subversion is a powerful open-source version control system that is typically used to manage the collections of files that make up software projects. However, a SVN repository may actually be used for managing any collection of files that are changed or modified over time.

According to [Hut13], a SVN repository (or Subversion repository) is a collection of files and directories, bundled together in a special database that also record a complete history of all the changes that have ever been made to these files.

/The Development teams is constantly working on this platform. There are all the software versions of the software. Thus, together with the JIRA and Sonar Qube we can get heavy information to process afterwards.

## 10.4 Jenkins

Jenkins is a powerful application that allows continuous integration and continuous delivery of projects, regardless of the platform that is being worked on. It is a free source that can handle any kind of built or continuous integration. You can integrate Jenkins with a number of automated testing and deployment technologies.

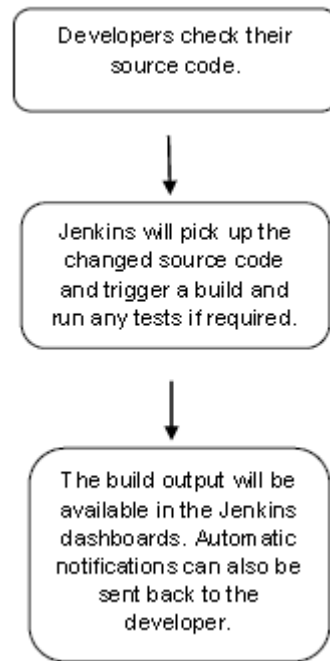


The screenshot shows the Jenkins main page interface. On the left, there is a sidebar with navigation options: Gent, Historial de muntatges, Anonymous View, Launch Tray App, and Dependency Graph. Below this, there are sections for 'Cua de muntatges' (No hi ha muntatges a la cua) and 'Estat de l'executador de muntatges' (master, c1devPatchBG01). The main content area is titled 'Build And Deploy Status' and 'Installationtrigger'. The 'Build And Deploy Status' section shows 'No jobs in this section.' The 'Installationtrigger' section displays a table of jobs with columns for status (S), warning (W), name, and last build status.

S	W	Name ↓	Darrer muntatge correcte
●	☀	<a href="#">triggerUnattendedCurrentMajor</a>	48 min - #831
●	☀	<a href="#">triggerUnattendedCurrentPatch</a>	1 hr 16 min - #665
●	☁	<a href="#">triggerUnattendedNextMajor</a>	7 mo 7 days - #62
●	☀	<a href="#">triggerUnattendedNextPatch</a>	8 mo 7 days - #106

Figure 10.4: Jenkins main page.

It is used by the software testers to build and test the project/product continuously in order to help the developers to integrate the changes to the project as quickly as possible and obtain fresh builds. Jenkins is installed on a server where the central build takes place. The following flowchart shows the basic work-flow of how Jenkins works. For this system the information fetched that is interesting from the Rest API will be provided in a JSON format. Afterwards the data is parsed to the main database, as is show in [tut13].



*Figure 10.5: Jenkins simple work-flow.*

## 10.5 Test Report

This is the internal testing interface developed by the Quality Assurance department. It displays all the results of the automation tests, all of which are stored in a data base, thus the purpose is only for research. The tests are for the current major version (current 16.1.0) and the current patch (16.0.2).

Every week around 40.000 different tests are run with a lot of combinations and environments. The variables of these environments are different OS (Windows, Linux & MAC OS), multiple browsers (Chrome, Firefox or IE), divergent databases (MYSQL or Oracle). The combination of all these environments are run for more than 4.000 test-cases.

As was explained in the theory (Software Testing chapter), here is where the multiple test types (suites) are applied. One of them is a basic suite for testing new, edit or deleted code from the development department. It is also used as the functional suite (black-box testing that bases its test cases on the specifications of the software component under test), so it checks the functionality of the application. Most of the test cases -> click logs in the test runs are made by a robot which follows the test cases created in advance by the automation team.

## 10 Systems Overview

Execution Round	Testrun	Starttime	Result %	Current Test/Status	Total	Tests Classic	Web	Initial F/E #	Jiras #	Rerun #	Duration (min:s)	Version	ExtJS Version	Config / Run Name	Server / Host	Client / SiteID	Display / Host	Browser	Display SID	RunID	SVN-Revision	
Round 0	21458	2016-05-27 10:52:07	64.56 %	FINISHED	886	18	868	384	27	9	19:00:10	16.1.0.194	ExtJS6	fs.us.ora12.ch	seu23	T01	sent06	OC 50.0.2661.102	3	runb	2749 (trunk)	
Round 0	21457	2016-05-27 10:51:18	83.30 %	FS_IRD_HR_FairValueHedge_Regres...	1551	1551	0	326	25	0	93:12:14	16.1.0.191	ExtJS3	fs.win.mssql14.classic	sent22	T01	sent22		3	runb	2749 (trunk)	
Round 0	21455	2016-05-27 10:50:00	95.73 %	FINISHED	1968	1968	0	115	23	1	22:58:50	16.1.0.194	ExtJS3	bs.us.ora12.classic	seu23	B01	sent22		2	runb	2749 (trunk)	
Round 0	21453	2016-05-27 10:48:26	63.74 %	Could not start browser	1652	276	1376	641	28	3	18:27:15	16.1.0.191	ExtJS6	bs.win.mssql14.ie	sent22	B01	sent27	IE 11	3	runb	2749 (trunk)	
Round 0	21452	2016-05-27 10:48:18	62.50 %	FINISHED	8	8	0	3	1	0	0:00:02	hs16.1.0.uc	ExtJS3	fs.versionnit.grawqa01	grawqa01	T01	grawqa01		0	runx	2749 (trunk)	
Summary Lines:			73.97 %		6065	3821	2244	1469	0	13	153:38:32											

Figure 10.6: Reval Test Report Interface.

Visible in the userface is:

1. The Execution Round: for each iteration there are four Rounds: Round 0 (1st week), 1 (2nd week) & 2 (3rd week) uses current builds, then the Round 3 (4th week) is the final round where code is freeze.
2. Test Run: is the test ID, which represents the combination of all attributes. Every testrun is different to the others because it covers a specific version, in different scenarios and host servers for an unequal build.
3. Starttime: when the testrun got going.
4. Result %: progress percentage of the testrun.
5. Current Status:
6. Tests: number of tests, there we find Total, Classic and Web.
7. Initial F/E, JIRAs, Rerun: This is the amount of failed test. Usually at the beginning of the testing some test fail. In case that a test fails it is tested again (rerun), and if it still failing a JIRA task is created for revision (so someone will check it manually) in case an application fails the JIRA task will be send to RQA (to define) if in case the robot fails then a JIRA is created for manual revision.
8. Duration: how long has been this testrun under testing.
9. Version: major or patch plus the actual build.
10. Config / Run Name: environment where the testrun is being tested.
11. Server Host: which server holds the testing.



# 11 Data Layer

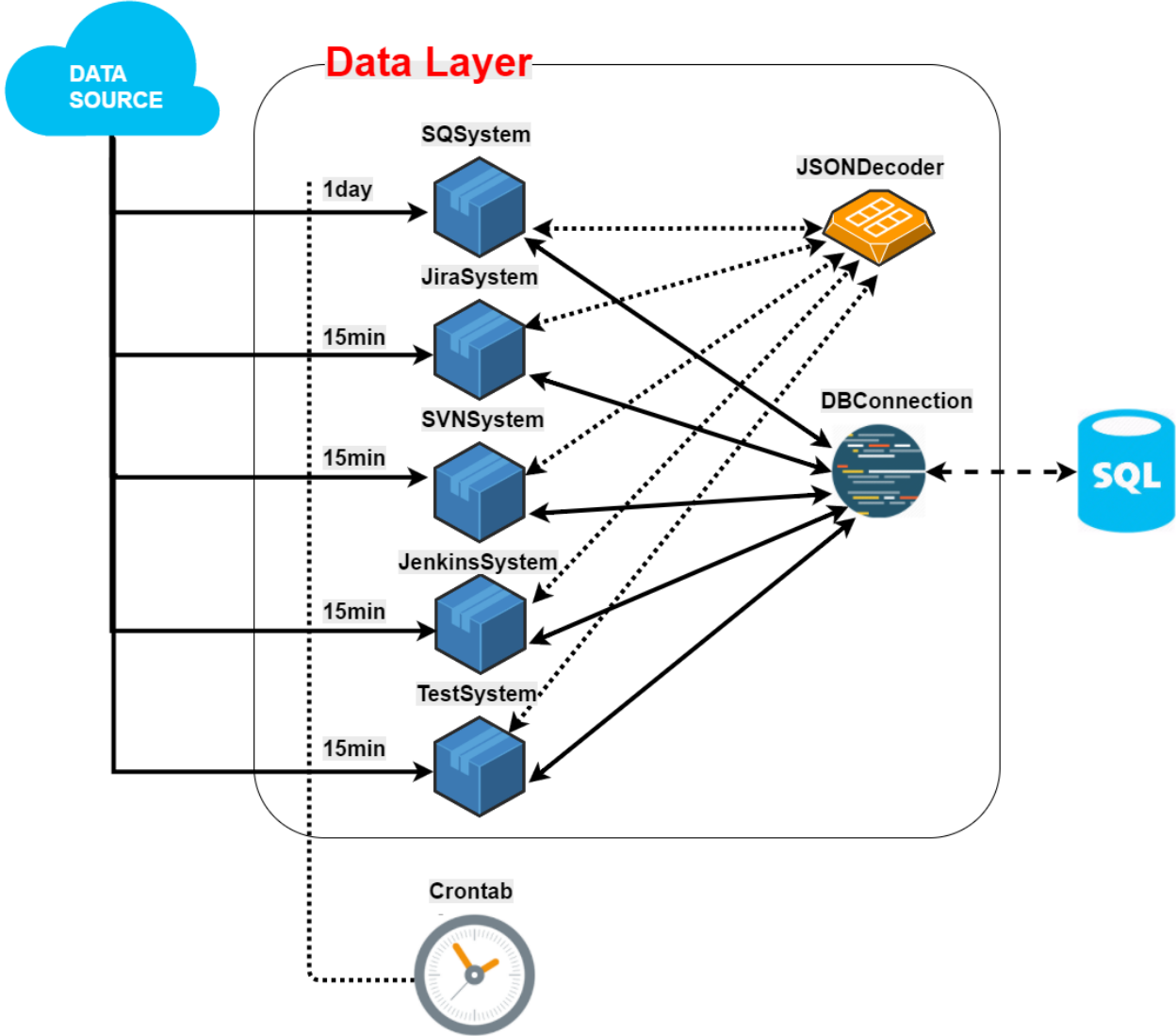


Figure 11.1: Data Layer Architecture Diagram.

## 11 Data Layer

The term Data Layer is a data structure which ideally fetches all data from a data source that should be processed and passed from your website (or other digital context) to other applications that you have linked to or store it in a data base. It is a term used by Google Tag Manager in a variety of contexts and has been adapted in this project architecture [Aha14].

The Data Layer is the core of this application. It is in charge of collecting a large amount the data from the data sources, polish the data and finally gather it in a data base.

The present Data Layer is divided in five individual main modules (one-to-one system) which have an scheduler (Crontab) that executes them and two other shared modules (DB communication and JSON & XML parser).

Each main module collects the data for a different system, as the following table shows:

Main Module	Data Source System
SQSystem	Sonar Qube
JiraSystem	JIRA
SVNSystem	SVN repository
JenkinsSystem	Jenkins
TestSystem	Test Report

*Table 11.1: Main module - Data Source System relation.*

At is said in the Data Sources explanation (see Architecture), most of the Data Sources have available access to the data by a REST API. It is very useful since REST uses HTTP to create, read, update and delete, making it more easy to access the information. With REST it's possible to create URL queries, this URL is sent to the server using a simpler GET request. Then, the HTTP reply is a raw result data probably in an XML or JSON file. For the last step, the JSONParser module is in charge to break it down, as is reported by [Eas10].

At the same time, each module follow a common pattern:

1. Connection to the Data Base.
2. HTTP Proxy Authentication Certificate. This ensures that all data passed remains private and integral.
3. Authentication to the Data Source System.

4. REST API URL generation.
5. Execution of the queries.
6. Return of the XML or JSON file.
7. Convert the XML or JSON file and assign values to variables.
8. Process the data, if necessary.
9. Store data in the Data Base.

## Crontab

A crontab is a simple text file with a list of commands meant to be run at specified times. It is edited with a command-line utility. These commands (and their run times) are then controlled by the cron daemon, which executes them in the system background. According to the Ubuntu Documentation [Doc15].

In our case, this is the crontab file, which calls the scripts that execute each module of the Data Layer:

```
8 * * * /home/hector/develop/SysConnection/src/SQSystem.sh
*/15 * * * * /home/hector/develop/SysConnection/src/TestSystem.sh
*/15 * * * * /home/hector/develop/SysConnection/src/JenkinsSystem.sh
*/15 * * * * /home/hector/develop/SysConnection/src/SVNSystem.sh
*/30 * * * * /home/hector/develop/SysConnection/src/JiraSystem.sh
```

*Figure 11.2: Crontab application file.*

The execution period of every file depends on the update rate of the Data Source System. For instance, Test Report, Jenkins, SVN, and JIRA have more live and fresh data, which changes often. In this case the data fetching is every 15 or 30 minutes. However, the Sonar Qube system is triggered once a day to be update, thus crontab executes the fetching once a day, which represents the maximum efficiency.

## **XML file**

Finally, one important characteristic to remark of the Data Layer is that almost every static variable is set in a XML file in case it is necessary make any changes it is more maintainable, and more successful at accomplishing one of the company's requirements. For instance, it is very useful when a Major or a Patch version changes to another one.

# 12 Dashboard

This chapter will be limited on to the explanation of the framework for the metric representation. In this section, the architecture that the Dashing uses will be displayed and then the configuration of a single widget will be show.

## 12.1 Architecture

It is called Dashing, and this is the architecture that follows:

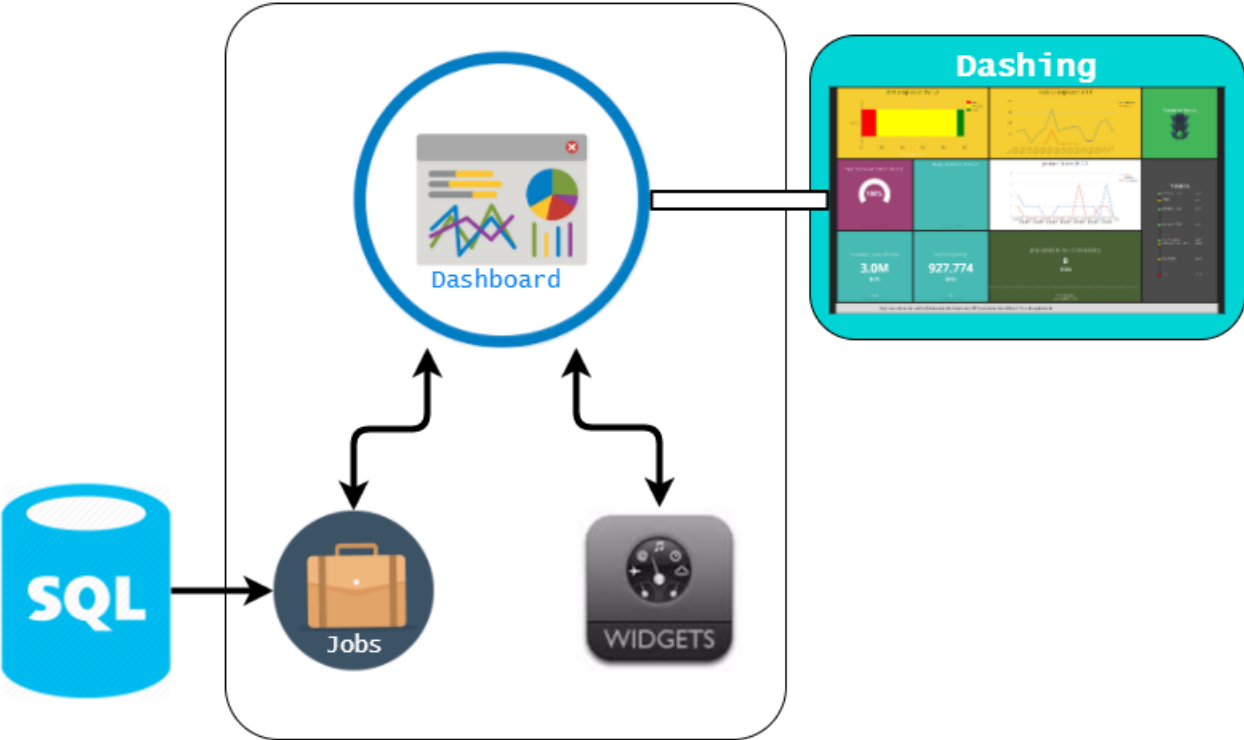


Figure 12.1: Business Logic and UI Architecture diagram.

## 12.2 Dashboard

Each widget is represented by a *div* element needing *data-id* and *data-view* attributes. The wrapping *<li>* tags are used for layout.

*Listing 12.1: One .erb file for each dashboard that contains the layout for the widgets.*

```
<% content_for(:title) { "My super sweet dashboard" } %>
<div class="gridster">
  <ul>
    <li data-row="1" data-col="1" data-size-x="1" data-
      size-y="1">
      <div data-id="valuation" data-view="Number" data-
        title="Current Valuation" data-prefix="\$"></div
      >
    </li>
  </ul>
</div>
```

*data-id*: Sets the widget ID which will be used when pushing data to the widget. Two widgets can have the same widget id, allowing to have the same widget in multiple dashboards. When data is pushed to that id, each instance would be updated.

*data-view*: Specifies the type of widget that will be used.

However, using different *data-* attributes allows customize them. Can be used any arbitrary attribute — each one will be available within the widget logic.

## 12.3 Anatomy of a widget

1. An HTML file used for layout and bindings.
2. A SCSS file for styles.
3. A coffeescript file which allows you to handle incoming data & functionality.

Listing 12.2: One Widget Dashboard example

```

<h1 class="title" data-bind="title"></h1>

<h2 class="value" data-bind="current_|_shortenedNumber_|_
  prepend_|_prefix"></h2>

<p class="change-rate">
  <i data-bind-class="arrow"></i><span data-bind="
    difference"></span>
</p>

<p class="more-info" data-bind="moreinfo_|_raw"></p>

<p class="updated-at" data-bind="updatedAtMessage"></p>

```

Widgets use batman bindings in order to update their contents. Whenever the data changes, the DOM will automatically reflect the changes.

As can be seen the piping '|' characters in some of the *data-bind*'s above. These are Batman Filters, and permits easily format the representation of data.

Listing 12.3: Widget's Coffeescript

```

class Dashing.Number extends Dashing.Widget
  ready: ->
    # This is fired when the widget is done being
    rendered

  onData: (data) ->
    # Fired when you receive data
    # You could do something like have the widget flash
    each time data comes in by doing:
    # $(@node).fadeOut().fadeIn()

    # Any attribute that has the 'Dashing.AnimatedValue'
    will cause the number to animate when it changes.
    @accessor 'current', Dashing.AnimatedValue

    # Calculates the % difference between current & last
    values.
    @accessor 'difference', ->
      if @get('last')
        last = parseInt(@get('last'))

```

```

current = parseInt(@get('current'))
if last != 0
  diff = Math.abs(Math.round((current - last) /
    last * 100))
  "#{diff}%"
else
  ""
# Picks the direction of the arrow based on whether the
  current value is higher or lower than the last
@accessor 'arrow', ->
  if @get('last')
    if parseInt(@get('current')) > parseInt(@get('last',
      )) then 'icon-arrow-up' else 'icon-arrow-down'

```

## 12.4 Job

The Jobs provide the data to the widgets. To specify which widget has to be used, is necessary assign the widget id. In this case, *"Karma"*.

Dashing uses *rufus-scheduler* to schedule jobs. This job will run every minute, and will send a random number to ALL widgets that have *data-id* set to *"Karma"*.

Jobs are where to put stuff such as fetching metrics from a database, or calling a third party API. Since the data fetch is happening in only one place, it means that all instances of widgets are in sync. Server Sent Events are used in order to stream data to the dashboards.

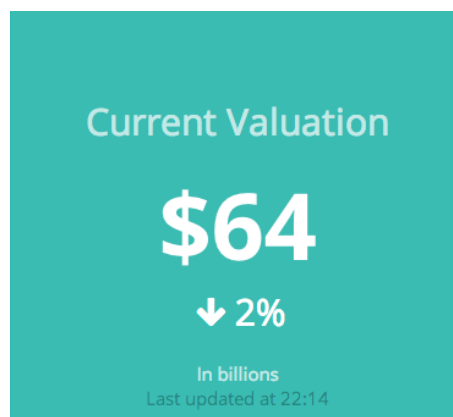


Figure 12.2: Result of the sample widget explained above.



*Listing 12.4: Job example*

```
# :first_in sets how long it takes before the job is
  first run. In this case, it is run immediately
SCHEDULER.every '1m', :first_in => 0 do |job|
  send_event('karma', { current: rand(1000) })
end
```

# 13 Final Implementation: Metrics - Widgets

## 13.1 JIRA Progress

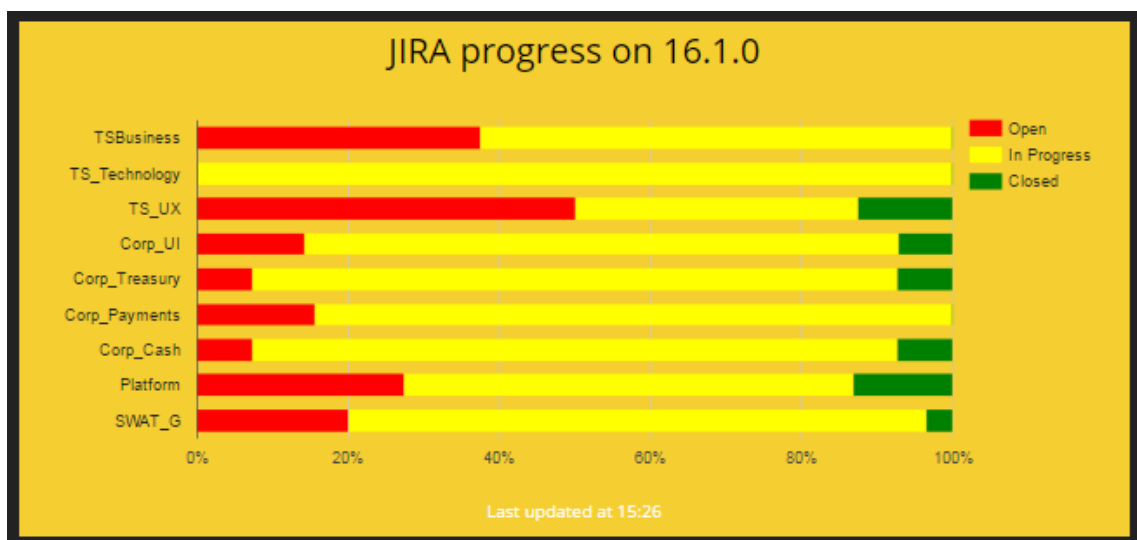


Figure 13.1: JIRA Progress Widget.

- **Name:** JIRA Progress.
- **Widget Type:** Live bar-char.
- **Data Source System:** JIRA
- **Metric:** Teams task tracking.
- **Description:** (process metric) This widgets gets the New, In Progress and Closed task from JIRA and displays them first in a global overview for teams and then every 15 seconds each team. This widget pretend to show at glance the amount of tasks and its status. So managers, team leaders and team members can follow the current situation of its work. The amount of task that

have they done, what are they doing and what they will have to do. This widget is only focus on the current Major version.

## 13.2 Code Changes

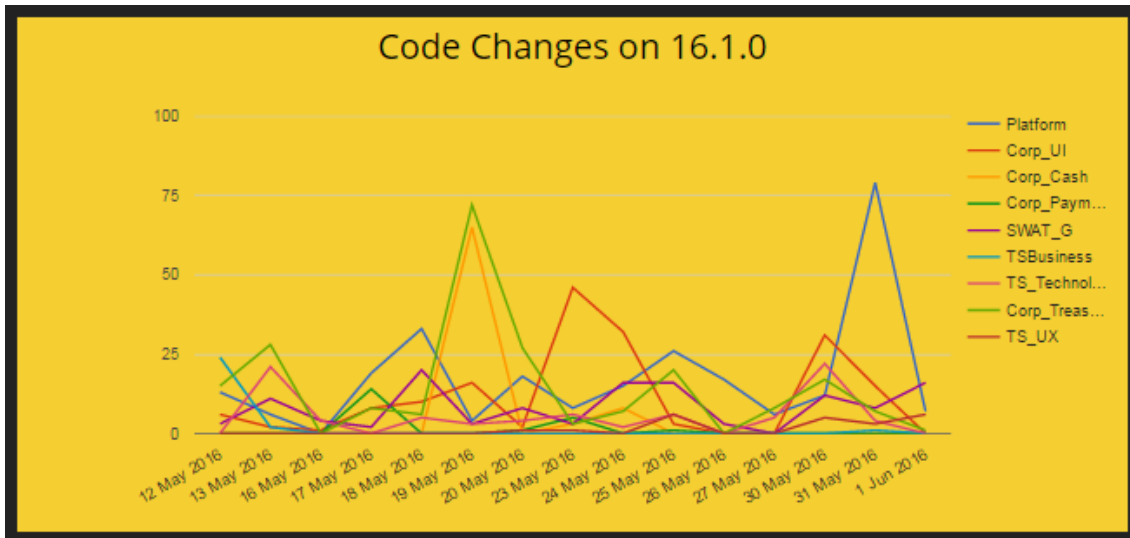
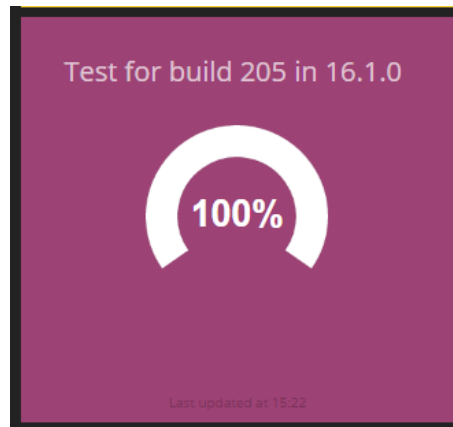


Figure 13.2: Code Changes Widget.

- **Name:** Code Changes.
- **Widget Type:** Historical line-char.
- **Data Source System:** SVN.
- **Metric:** (Process metric) Activity vs. results.
- **Description:** In this case, this is a historical line-char that shows the code changes per team in the last 15 days. Here the internal stakeholders can analyse which teams are working more on the source code and which others are focusing in other tasks. At the same time, it can be used this information with the JUnit widget to analyse if the amount of changes influence to a successful build testing. Sometimes it happens that many changes solve a problem. So on this widget it can be tracked. As the JIRA Progress widget, this one show first the global overview to compare with other teams and then every 15 second is amount of changes per team plus the sum of the total changes per team. Also, this widget only covers the current Major version of the software.

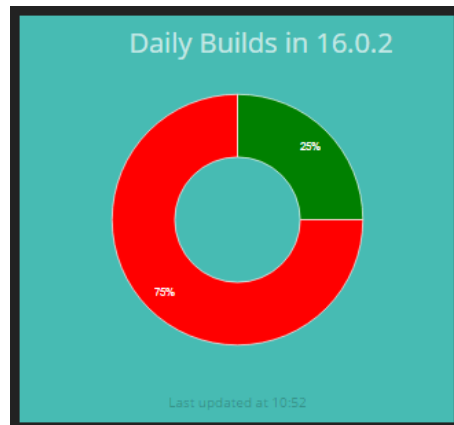
## 13.3 Test for Builds rate



*Figure 13.3: Test for Builds rate Widget.*

- **Name:** Test for Builds rate.
- **Widget Type:** Meter graph.
- **Data Source System:** Testing Report
- **Metric:** (Product metric) Test coverage success rate.
- **Description:** This widget shows the average of the sum of all testings in one build. First, it gets the amount of tests executed for one testrun. Then, get the amount of all tests that were failing for one run given by testrunid (Initial F/E # in overview page). Finally, get all the outstanding errors (not reviewed yet, not rerunned yet, not jira linked yet) and calculate by the rule of three. Giving as a result the success testing rate of the actual build. In this case both Major and Patch current builds are displayed, every 15 seconds as well.

## 13.4 Daily Builds



*Figure 13.4: Daily Builds Widget.*

- **Name:** Daily Builds.
- **Widget Type:** Donut Pie graph.
- **Data Source System:** Jenkins
- **Metric:** (Product metric) Daily builds success percent.
- **Description:** Here is displayed with a Donut-pie Chart and with green (success) and red (fail) the success percent. For instance, if a day there are 4 build which 3 are fail and one is successful the widget will be 75% in red color and the rest 25% in green color. This widget help also to keep track of the actual situation of the builds, so developers and the rest of stakeholders can keep an eye on it.

## 13.5 Jenkins Builds History

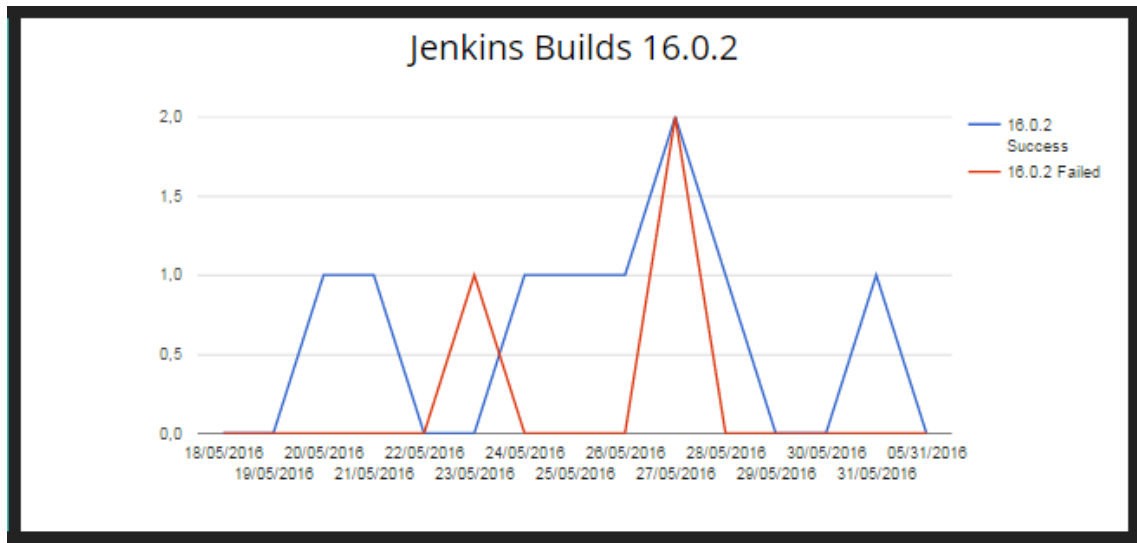


Figure 13.5: Jenkins Builds History Widget.

- **Name:** Jenkins Builds History.
- **Widget Type:** Historical line-char.
- **Data Source System:** Jenkins.
- **Metric:** (Product metric) Historical build success.
- **Description:** This widget is similar to the previous one, since it shows the successful and fail builds for each version. But in this case it gives a historical perspective of the build of the last 15 days. It allow the managers to remember how performed the builds in the last days.

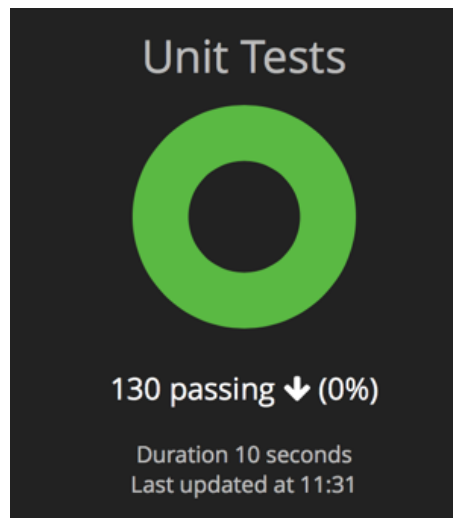
## 13.6 NLOC



*Figure 13.6: NLOC Widget.*

- **Name:** NLOC - Number Line Of Code.
- **Widget Type:** Number with increase or decrease percent with arrow.
- **Data Source System:** Sonar Qube.
- **Metric:** (Product metric) Source code growing rate
- **Description:** LOC is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. It is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced. This widget shows the number of line of code, in this case 3 Million. But, furthermore, it shows the increasing or decreasing percent from the day before. This widget is only updated once a day.

## 13.7 JUnit Tests Rate

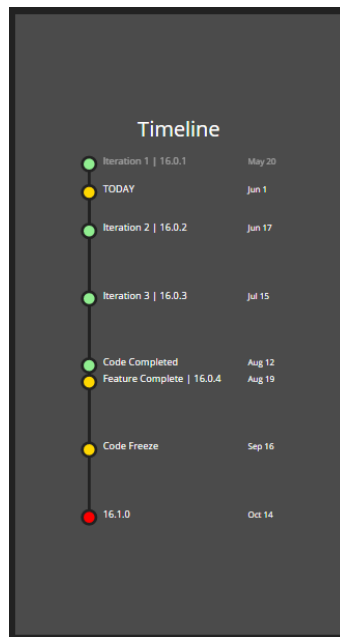


*Figure 13.7: JUnit Test Rate Widget.*

- **Name:** JUnit Tests Rate.
- **Widget Type:** Donut Pie graph.
- **Data Source System:** Jenkins.
- **Metric:** (Product metric) JUnit test success.
- **Description:** This widget display the number of passing, ignored and failing test from a test result file in the JUnit XML format. It also inform the duration of the testing and the last update. With this widges the developers can keep track on the testing of its software.



## 13.8 Timeline



*Figure 13.8: Timeline Widget.*

- **Name:** Iterations, Release and events timeline.
- **Widget Type:** Time line graph.
- **Data Source System:** Outlook Calendar
- **Metric:** (Project metric) Product life-cycle.
- **Description:** This widget does not represent any metric, it just displays the actual situation along the software life. So the purpose of this widget is to allow the people involve in the project to keep track on the future events and in the current phase status.

## 13.9 Ticker Bar



*Figure 13.9: Tickbar news Widget.*

- **Name:** Tickbar news.
- **Widget Type:** Tickbar.
- **Data Source System:** Jira, SVN, Jenkins and others.
- **Metric:** (Project metric) News.
- **Description:** The tickbar is a list of the most recent development or testing news. For instance, the last three issues closed in JIRA, or who was the team that made more commits last week. At the same time it can give information about the build status or the availability of the environments for testing. All in all, it will display the latest news of the project performances.

# 14 Conclusions

Visualization of software metrics in a dashboard is a subject that matters to those companies that develop software projects. It gives an overview of performance data to project managers and other internal stakeholders which have an overall responsibility for developing software that meets the requirements and is delivered on time and within budget.

To better understand the thesis it is necessary to comprehend the basic theoretical aspects of Software Engineering. The key to developing a proper software is to start with clearly defined requirements, come up with a software design that perfectly fulfills the requirements and then have a clean implementation/development. Afterwards, such a software has to be tested for error management. During this process, various measurements can be done that indicate how the company performs.

The main focus during the development process of the application, were the measurements that could be used to improve the processes. All features were streamlined for maximum usability. This process also taught me the impacts of how company KPIs can shape the decision making of the stakeholders. The perfect solution for a company is not always the most powerful tool with the highest number of features. The best solution for a software has to fulfill the specifications as precisely as possible.

While the development of this project, I grew my analytical skills and got a more proactive and dynamic attitude. At the same time, I have been working with new languages and frameworks. And I have acquired a better understanding of the software production life-cycle, the software companies' architecture and workflow.

## 15 Future prospect

The project is not finish yet, soon there will be new implementations. New metrics will be used and more widgets will be created. An historical graph of open/closed task will be featured, also other two that show the meantime to fix a failure and other that display the effort of completing a task. At the same time, the company would like to customize the dashboard for teams. So each team can access to its own info-board from its own computer. Another new feature will be the use of a client/server SQL database engine. In this case, will be an Oracle database. One of the reasons is that Oracle DB been one the leader in relational database is able to handle a lot of data in a well organized way. Moreover, it delivers a very stable system for running databases with fail-over solutions, backups, quick 'data resets' if needed.

# Bibliography

- [Agi13] Leading Agile. *How Do You Know Your Metrics Are Any Good*. 2013. URL: <http://www.leadingagile.com/2013/07/how-do-you-know-your-metrics-are-any-good> (visited on 06/05/2016).
- [Aha14] Simo Ahavas. *Google Tag Manager's Data Model*. [Online; accessed 05-June-2016]. 2014. URL: <http://www.simoahava.com/analytics/google-tag-manager-data-model/#gref> (visited on 06/05/2016).
- [Cai14] Larry Cai. *Learn Dashing Widget in 90 minutes*. 2014. URL: <http://www.slideshare.net/larrycai/learn-dashing-widget-in-90-minutes> (visited on 06/05/2016).
- [Cer13a] ISTQB Exam Certification. *ISTQB Association*. 2013. URL: <http://istqbexamcertification.com/what-is-defect-or-bugs-or-faults-in-software-testing> (visited on 06/05/2016).
- [Cer13b] ISTQB Exam Certification. *ISTQB Exam Certification*. 2013. URL: <http://istqbexamcertification.com/what-is-acceptance-testing/> (visited on 06/05/2016).
- [Doc14] Atlassian Documentation. *Working with workflows*. 2014. URL: <https://confluence.atlassian.com/adminjiracloud/working-with-workflows-776636540.html> (visited on 06/05/2016).
- [Doc15] Ubuntu Documentation. *Cron How to*. 2015. URL: <https://help.ubuntu.com/community/CronHowto> (visited on 06/05/2016).
- [Doc16] Sonar Qube Documentation. *SONAR QUBE*. 2016. URL: <http://www.sonarqube.org/> (visited on 06/05/2016).
- [Eas10] Balamurugan Easwaran. *Implementation advantages of rest*. 2010. URL: <http://www.slideshare.net/BalamuruganEaswaran/implementation-advantages-of-rest> (visited on 06/05/2016).
- [Gei14] Matthias Geigers. *SonarQube – What is it? How to get started? Why do I use it?* 2014. URL: <https://matthiasgeiger.wordpress.com/2014/02/19/sonarqube-what-is-it-how-to-get-started-why-do-i-use-it/> (visited on 06/05/2016).
- [Gur16] Guru99. *All About Compatibility Testing*. 2016. URL: <http://www.guru99.com/compatibility-testing.html> (visited on 06/05/2016).

## Bibliography

- [Hut13] Project Hut. *What is a SVN repository?* 2013. URL: <https://www.projecthut.com/what-is-svn-repository/> (visited on 06/05/2016).
- [Ken07] C.S Kent. *Software Metrics*. 2007. URL: <http://www.cs.kent.edu/~jmaletic/cs63901/lectures/SoftwareMetrics.pdf> (visited on 06/05/2016).
- [Nic12] Dave Nicolette. *Software Development Metrics*. Pearson, 2012. ISBN: 9780198520115.
- [Pea06] Pearson. *What Is Software Configuration Management?* 2006. (Visited on 06/05/2016).
- [Som09a] Ian Sommerville. *SOFTWARE ENGINEERING*. Pearson, 209. ISBN: 978-0-13-703515-1.
- [Som09b] Ian Sommerville. *SOFTWARE ENGINEERING*. Pearson, 209. ISBN: 978-0-13-703515-1.
- [tut13] tutorialspoint. *Jenkins Tutorial*. 2013. URL: <http://www.tutorialspoint.com/jenkins/> (visited on 06/05/2016).
- [Wik01a] Wikipedia. *oftware Testing*. 2001. URL: [https://en.wikipedia.org/wiki/Software\\_testing](https://en.wikipedia.org/wiki/Software_testing) (visited on 06/05/2016).
- [Wik01b] Wikipedia. *Software testing*. 2001. URL: [https://en.wikipedia.org/wiki/Software\\_testing#Functional\\_vs\\_non-functional\\_testing](https://en.wikipedia.org/wiki/Software_testing#Functional_vs_non-functional_testing) (visited on 06/05/2016).
- [Wik01c] Wikipedia. *Unit testing*. 2001. URL: [https://en.wikipedia.org/wiki/Unit\\_testing](https://en.wikipedia.org/wiki/Unit_testing) (visited on 06/05/2016).
- [Wik02] Wikipedia. *Regression Testing*. 2002. URL: [https://en.wikipedia.org/wiki/Regression\\_testing](https://en.wikipedia.org/wiki/Regression_testing) (visited on 06/05/2016).
- [Wik03] Wikipedia. *Integration testing*. 2003. URL: [https://en.wikipedia.org/wiki/Integration\\_testing](https://en.wikipedia.org/wiki/Integration_testing) (visited on 06/05/2016).
- [Wik04a] Wikipedia. *Cohesion (computer science)*. 2004. URL: [https://en.wikipedia.org/wiki/Cohesion\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Cohesion_(computer_science)) (visited on 06/05/2016).
- [Wik04b] Wikipedia. *System Testing*. 2004. URL: [https://en.wikipedia.org/wiki/System\\_testing](https://en.wikipedia.org/wiki/System_testing) (visited on 06/05/2016).
- [Wik06a] Wikipedia. *Dashboard*. 2006. URL: [https://en.wikipedia.org/wiki/Dashboard\\_\(management\\_information\\_systems\)](https://en.wikipedia.org/wiki/Dashboard_(management_information_systems)) (visited on 06/05/2016).
- [Wik06b] Wikipedia. *Functional Testing*. 2006. URL: [https://en.wikipedia.org/wiki/Functional\\_testing](https://en.wikipedia.org/wiki/Functional_testing) (visited on 06/05/2016).
- [Wik06c] Wikipedia. *Recovery Testing*. 2006. URL: [https://en.wikipedia.org/wiki/Recovery\\_testing](https://en.wikipedia.org/wiki/Recovery_testing) (visited on 06/05/2016).
- [Wik08] Wikipedia. *Project management triangle*. 2008. URL: [https://en.wikipedia.org/wiki/Project\\_management\\_triangle](https://en.wikipedia.org/wiki/Project_management_triangle) (visited on 06/05/2016).

- [Wik10] Wikipedia. *Sonar Qube*. 2010. URL: <https://en.wikipedia.org/wiki/SonarQube> (visited on 06/05/2016).
- [Wik12] Wikipedia. *Shopify/dashing*. 2012. URL: <https://github.com/Shopify/dashing/wiki> (visited on 06/05/2016).
- [Wik13a] Wikipedia. *Smoke Testing*. 2013. URL: [https://en.wikipedia.org/wiki/Smoke\\_testing\\_\(software\)](https://en.wikipedia.org/wiki/Smoke_testing_(software)) (visited on 06/05/2016).
- [Wik13b] Wikipedia. *Software design*. 2013. URL: [https://en.wikipedia.org/wiki/Software\\_design](https://en.wikipedia.org/wiki/Software_design) (visited on 06/05/2016).