BUILDING A FRAMEWORK TO SPECIFY AND AUTOMATICALLY TEST NON-FUNCTIONAL REQUIREMENTS IN BEHAVIOUR-DRIVEN DEVELOPMENT

Master Thesis

Fabio Ricchiuti

31 March 2017

SUPERVISORS

Universitá degli studi dell'Aquila
Prof. Dr. Henry Muccini

Vrije Universiteit Amsterdam
Dr. Natalia Silvis-Cividjian

ING Bank N.V.
MSc. Alessandro Vermeulen
Drs. Egbert-Jan Buiten

Contents

Chapter 1

Introduction

In the last years, information technology obtained a crucial role in business. The engineering of methods and processes related to the lifecycle of the software became essential in order to guarantee products that fulfill the expectations of the customers. The need for higher quality and shorter time-to-market made software engineering evolving.

In particular, the software engineering community witnessed a progressive shift of processes aimed to guide the design of the software. One of the most famous legacy processes is represented by the waterfall model [Roy+70]. As [LN04] states "The waterfall model is a sequential (non-iterative) design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance." Nevertheless, in a world where the time for software production is never enough and requirements can change everyday, dynamic processes with a high degree of flexibility became crucial. Agile[Agi01] turned to be the de facto standard for managing the software life cycle. Agile is defined as "The ability to create and respond to change in order to succeed in an uncertain and turbulent environment."[Agi16].

The philosophy introduced with Agile created the basis to build new methodologies. An ecosystem of methodologies and framework have been created. Behaviour-Driven Development(BDD) is one of them. BDD emerged from the so-called Test-Driven Development(TDD). These methodologies highlight the importance of the testing, giving it a key role in the development, contrary to Waterfall that put testing on the last steps of the whole process. BDD encourages the use of scenarios to describe which features the software should have and how these interact with the user and the other components. More details about TDD and BDD will be provided in Chapter 2.

Specification languages used with BDD, such as Gherkin[1], allow, firstly, to express functional requirements written in a simple English in such a way that they are understandable even for non-IT people or managers and, secondly, to specify tests through the definition of scenarios that guide the developers in delivering exactly what the stakeholders want. Given the simplicity and efficiency of this approach, its adoption is continuously increasing.

When developers have to deal with quality concerns the process turns to be less straightforward. The traditional notion of non-functional requirements (NFRs),

[1] https://github.com/cucumber/cucumber/wiki/Gherkin

or quality requirements, describe them as constraints on a particular property or behavior of the system. Nonetheless, the line between a functional requirement and a NFR is sometimes really subtle or nonexistent. Their nature makes them hard to specify, especially in the early stage of a project, and consequently, they are often overlooked and treated as second-order requirements. There are several reasons:

- NFRs are in general poorly understood[PK04]
- For a customer without IT background, it is easy to describe what the software should do but hard to describe how. For instance, it is sure that the owner of an e-commerce desires his web shop operative 24 hours per day, but it is unlikely that he knows how the software should scale under pressure, for example when an abnormal amount of user are trying contemporaneously to purchase a product. Therefore, the definition of NFRs is left to the assumptions of the developers. This can bring unpleasant consequences. As well as delivering a product that does not fulfill the need of the customers due to wrong assumptions, stakeholders can underestimate the necessary work for managing the quality concerns resulting in creating wrong expectations in term of required effort of the developers.
- The need of certain NFRs may become evident when it is too late. For instance, this is often the case in the development of complex systems, which are the result of the composition of subsystems. A subsystem can have several factors that impact its efficiency. A software engineer can design strategies to tolerate them. The problem arose when unexpected events or unforeseen combinations of expected events occur, i.e. an abnormal amount of users try to purchase a product while an energy blackout is happening in the data center. These events may lead to a degradation or disruption of the service. A well-known example is the one of Ariane 5[Lio+96], a rocket that exploded 40 seconds after the launch for an exception thrown from an erroneous floating point conversion.

A detailed overview of the issues related to the elicitation of NFRs is provided by [UIK11]. [Mar11] showed the impact of NFRs in process success. In addition, he outlined a correlation between the application of NFRs verification techniques earlier in the software lifecycle and the possibility of a successful project. Thus, risks are lowered and a realistic planning can be made.

Management and reduction of risk are important in every enterprise but it is even more crucial in the banking sector. Financial transactions constantly move at a frenetic pace and software plays a central role. Low performances of infrastructures and interruptions of the services can lead to catastrophic consequences. Furthermore, the high competition in the global landscape and the implicit level of criticality of the field does not leave room for mistakes.

ING Bank is one of the leaders of the North European banking sector, strongly interested in embracing the latest technologies to improve their services and the customer experience. They are a global financial institution with a strong European base, offering banking services. The customer base comprehends individuals, families, small businesses, large corporations, institutions and governments. With more than 52,000 employees, ING offers retail and wholesale banking services to customers in over 40 countries. Moreover, ING is currently among leaders in the Dow Jones Sustainability Index 'Banks industry' group.

ING Bank strives for enhancing the processes related to the production of their software. In particular, specification and verification of NFRs. Current processes allow to identify and satisfy their NFRs during the acceptance phase. The lack of a general framework bring every team to create their own strategies and to use ad-hoc manual tests or specialized tools. On one hand, this approach can be considered effective, on the other hand, it can be improved. In order to verify NFRs developers are required to learn and use additional technologies and platforms, which can lead to an excessive overhead of their development process. In addition, for the reasons above mentioned NFRs are tested and verified only in the later stages of the projects.

The objective of the thesis is to improve the process concerning specification and verification of NFRs.

The research questions are the following:

RQ1) How can we improve the process regarding specification of non-functional requirements?

RQ2) How can we improve the automation of the testing of non-functional requirements?

This thesis proposes a tool that can guide the specification of correct and measurable NFRs in English within the context of Behaviour-Driven Development called TomatoFramework. The framework gives the instruments to developers to determine and verify NFRs since the definition of the very first scenario. Moreover, it automates the verification of NFRs providing test suites that can interpret quality requirements and execute the appropriate verification techniques. This master thesis has been carried out within ING Netherlands in collaboration with the API Platform team during an internship of 6 months. Firstly, a research of the state of the art of the approaches of specification and testing of NFRs was carried out. This overview highlighted the open challenges in the literature. In addition, a previous work concerning a framework for BDD and NFRs called ProBDD emerged [BE11]. However, ProBDD lacked a validation in industrial settings. Consequently, several experts withing ING were interviewed. The consequent analysis of the gathered data identified the open challenges from literature and empirical experience concerning NFRs specification and verification. This information leads the design of an architecture for a new framework. Lastly, the framework was evaluated by other experts who did not take part of the previous interviews.

In the next chapter, BDD will be described. In Chapter 3, an overview of the state of the art of specification and verification of NFRs will be shown and a picture of the current methodologies adopted in ING will be presented. Next, design and implementation of TomatoFramework will be introduced, respectively, in Chapters 4 and 5. Finally, an evaluation of the framework, the discussion and the final remarks are provided in Chapter 6

Chapter 2

Behaviour-driven development

One of the practices emerged in the last years to guide software design and development was Test-Driven Development(TDD). The simple idea behind TDD is to define and execute tests before implementing the software itself. As described in [Bec03], TDD is a process of five steps:

- Adding a test. In order to deal with the complexity of the software, a first process of abstraction is carried out dividing the project in features. The development process starts by selecting a feature and writing a test. The test should be written in agreement with the specification of the software.
- Running the tests. All the tests already defined are launched in conjunction with the new test. The new test should fail because the code to be tested does not exist yet.
- Writing the code. In this phase, the bare minimum code is written just to make the new test passing. The code will be enhanced in the next stages.
- Running all the tests again. If the written code is correct and satisfies the specification, the new test should pass. It is crucial that the other tests must keep showing positive results. In fact, negative results would highlight bugs or degradation of the overall performance introduced by the new code in other features.
- Refactoring the code. Developers can now adjust the code rearranging classes and methods. A better structure can be provided moving the code where it logically belongs. In some cases, it is useful running all the test again.

The previous steps should be applied iteratively for each new test. This will ensure a continuous feedback of the status of the development of the software, as well as a greater confidence of testers and developers in their code. This practice turns to be extremely effective in enterprise environments characterized by frequent releases and short sprints.

However, TDD is not exempt from limitations. Defining effective and maintainable tests can be a hard task depending on the experience of the developers and the complexity of the domain of the project. This can lead in testing too much or too little with consequent waste of time and effort, not to mention that it easy to define tests cases that do not reflect the requirement specification. Moreover, during the design of a software, stakeholders without IT background are often involved in the

process. Their opinion may be crucial since they are the ones that have an extended knowledge of the business domain in which the software will operate. They might provide an extensive feedback during the definition of the tests in order to ensure that all the important aspects are covered. Unfortunately, the complexity of the IT concepts can represent a tough obstacle for them. The unfortunate conclusion of such scenario is that misunderstandings are dragged until the last stage of a project, or even in production, with serious consequences.

An alternative approach overcoming the limitations mentioned in the previous paragraph was needed. Under those circumstances, BDD[Nor06] emerged. BDD was invented by Dan North. His revolutionary approach, mainly conceptual, introduced the term "behaviour" over "test". The underlying idea behind BDD is that using behaviours to describe the characteristics of a software is more natural than using tests. Behaviours can be expressed with examples and scenario,"instruments" used commonly by each of us in our everyday communication. Requirements can be gathered with a collaborative approach Specification By Example[Fow04] which helps to pose the right questions in order to narrow down the scope to meaningful scenarios. They are expressed trough the *Given-When-Then* paradigm and composed by steps. A step is a sentence following one of the keywords *Given*, *Then*, *When*, *And*, *But* and they should be expressed as simple as possible without compromising their expressiveness. Hence low-level details should be omitted. Every step is associated with a method called step definition. A step definition is a snippet of code that turns into concrete actions the statements written in the steps. Therefore, every scenario comprehends the execution of steps, and consequently step definitions. The structure of a scenario commonly adheres to the following template:

- Preconditions of the scenario follow the keyword *Given*. Actions like an instantiation of variables and objects or opening of files are usually executed in this phase.

- Triggering events are declared after the keyword *When*. In this case, the step definitions associated will simulate the execution of the event.

- After the *Then* keyword, the operations executed when all the preconditions are valid and the triggering events are completed are described. These operations are the outcome of the execution of the whole scenario.

Among the tools that support BDD, Cucumber[1] is the most popular.

Cucumber guides developers into the application of the BDD methodology. Figure 2.1 shows an overview of the process. Scenarios are defined with a language called Gherkin. Gherkin is "Business Readable, Domain Specific Language" [17] that can be used for documentation and automatic testing. Listing 2.1 displays a simple example of a scenario expressed with Gherkin. It describes the action of purchasing a product. The steps are the following: First, the price of a product and the credit left in the account of the user are defined. Second, the user executes an action that shows his/her intention of buying a certain amount of that product. Finally, an amount equal to the price of the product is charged to the account of the user. It is worth noticing that the level of abstraction is intentionally high. It is not described whether the user pushes a button or performs other actions since the

[1] https://cucumber.io/

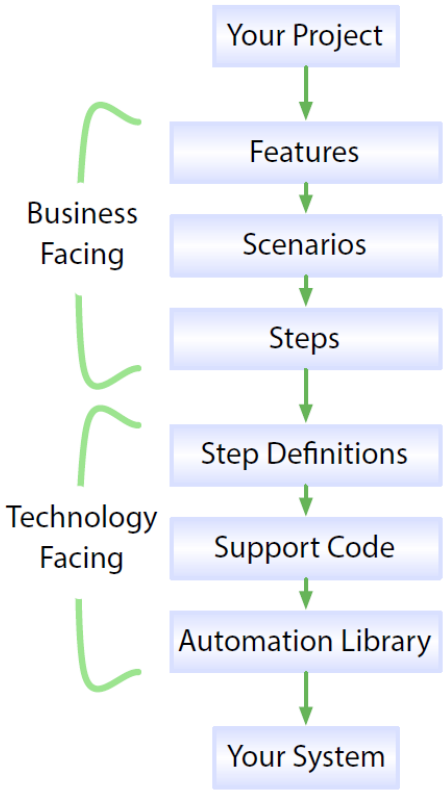

Figure 2.1: Structure of a typical Cucumber test suite. Diagram (from [WH12])

objective of the scenario is describing as simple as possible the performed action. An high level of abstraction keeps the scenario decoupled from its implementation and its interface and it is easy to understand. However, new details can be added in the multiple iterations when new information is available. Every Gherkin file contains a single feature with all the scenarios describing its functionalities.

```
Feature: Purchase
Scenario: Purchasing of product A
Given the price of a "product A" is 40 euro
And my account contains 1 euro
When I order 1 "product A"
Then 40 euro should be deducted from my account
```

Listing 2.1: Scenario of a purchase of a product in an e-commerce

Every step of a scenario is bound to a method called Step Definition. They are linked with the Gherkin files through regular expression. The whole set of files containing the steps defintion is called glue code. Listing 2.2 encloses the step definitions and the respective regular expressions.

```
@Given("^the price of a \"([^\"]*)\" is (\\d+) euro\$")
public void the_price_of_a_is_euro(String product, int price)
    throws Throwable {
     Store.setPrice(product,price);}

@Given("^my account contains (\\d+) euro\$")
public void my_account_contains_euro(int amount){
     Account.initializeTo(amount);}

@When("^I order (\\d+) \"([^\"]*)\"\$")
public void i_order(int quantity, String product) throws
    Throwable {
     Store.order(product,quantity);}

@Then("^(\\d+) euro should be deducted from my account\$")
public void euro_should_be_deducted_from_my_account(int price)
    throws Throwable {
     Account.deductAmount(price);}
```

Listing 2.2: Steps definition in Java of the scenario "Purchasing of product A"

To summarize, the flow of execution starts when Cucumber is executed. Cucumber parses every feature and every scenario associated. Every scenario contains steps which in turn are linked through regular expression to step definitions. In the previous example Cucumber will execute the steps definition *the_price_of_a_is_euro*, *my_account_contains_euro* , *i_order* and *euro_should_be_deducted_from_my_account* in this order. The operator parenthesis inside the regex extrapolates the parameters to pass to methods. If every step definition is executed without exception, the scenario is considered passed.

Chapter 3

Specification and testing of non-functional requirements

This chapter focuses on the identification of the potential shortcomings of current NFRs methodologies. Firstly, Section 3.1 identifies open challenges in non-functional specification and testing gathered from the literature. Section 3.2 shows an investigation carried out in an industrial context.

3.1 Literature studies overview

Despite the importance that NFRs play in the production of successful software projects, they are often treated as second-order requirements[Mar11]. Their abstraction and complexity did not permit to reach a shared consensus in the software engineering community on how to handle of NRFs[Gli07]. Several frameworks, tools and processes have been created to drive the treatment of NFRs, but studies appear to be scattered in the literature and heterogeneous. In this context, diverse systematic literature reviews were conducted to outline the state of the art of NFRs specification and testing. [LSZH13] catalogued methodologies and tools according to three dimensions:

- Context: They focus on either early requirements or late requirements.
- Process: They affect one or more of the steps of the development such as elicitation, specification, negotiation and validation.
- Application: Whether they address domain-specific concerns or generic.

Among the outcome of the study two important conclusions were:

- Approaches managing early requirements are significantly outnumbered by approaches for late requirements. This is particularly interesting in light of the work of [Mar11] that showed a positive correlation between successful projects and the application of methodologies focused on early requirements.
- The scarcity of domain-specific approaches.

As an extension of the previous study, [HLN14] carried out a more refined categorization of NFRs methodologies. They identified five steps of requirement analysis such as elicitation, specification, prioritization, modeling and verification & validation. They classified the approaches in three main categories:

Challenges	Extracted from	RQs
C1: The need of approaches managing early requirements. There is a lack of them despite that they appear to be more effective.	[Mar11],[LSZH13]	(1)
C2: Current approaches concerning alignment between specification and testing of requirements do not take enough consideration of NFRs.	[BE11]	(1),(2)
C3: Evaluation and validation of new proposals are not fully performed.	[BE11]	(1)(2)
C4: Evaluation and validation of new proposals are often not carried out in industry settings.	[BE11]	(1)(2)

Table 3.2: Challenges emerged from the literature study overview.

Goal-oriented Goals are conditions and situations that requirements experts want to achieve. These approaches highlight possible conflicts among the goals and help in taking the right design decisions to bring the best trade-off among goals.

Aspect-oriented These approaches aim in simplify NFRs analysis focusing only on one aspect of the system per time. An aspect consists of the implementation of a module that handles a particular property or a quality attribute of the system.

Pattern-based Reuse of existing practices compatible with the system in development is the objective of these approaches. Templates and formal specification patterns languages are an example of a pattern-based approach for the specification step.

Furthermore, [BE11] carried out another systematic review focusing on three steps: specification, testing and the alignment of the two during the development. Among the conclusions reported on the review of specification methodologies, it is mentioned that combination of approaches are more popular because they allow handling the specification from different perspectives at the same time. Moreover, evaluation and validation of the proposals in industry settings are often not carried out. The review of testing approaches depicted a similar picture. Evaluation and validation are often not fully performed as well as "Studies on NFR testing have been primarily conducted in academia. Further effort is therefore required to verify the industrial applicability of the study results." as reported by [BE11]. Concerning the alignment of specification and testing, "a significant gap between these areas" is still present, both, in industry and scientific literature."

3.2 Survey at ING

An enterprise with a strong IT footprint that involves the finances of millions of consumers and hundreds of thousands of businesses like ING requires software development processes that evolve and adapt to the market and the need of the customers. For this reason, ING is continuously researching new ways to improve the

quality of its software platform and the services offered. At the same time, it strives to reduce the time in which new services or updates to existing ones are delivered. Methodologies and technologies used for specification and testing of requirements can impact the whole process. Impacting factors are not only technical but also social and psychological. In order to have a detailed picture of the potential weaknesses of current methodologies and the effectiveness of the communication among the participants of a software project, a survey was carried out. The investigation was made trough a series of semi-structured interviews. The format was chosen due to the qualitative nature of the data researched. Participants were encouraged to talk freely about their work experience gathered over the years. For this reason, although a general structure for the interviews was defined, the participants could add their opinion, lessons learned and anecdotes. The interviews were recorded and examined. Consequently, the main concepts were extracted and transcribed, being careful of the context in which they were expressed. Interviews were conducted following the guidelines provided by [HA05]. In addition, data were reported with a similar methodology used in [MLMPT13]. A set of teams was selected based on what extent their products or services were impacted by NFRs. The criteria were: either the software produced by the team is affected significantly by quality concerns or that their activity is focused on training other teams on quality practices and assuring the compliance to quality standards. For time restrictions, the sample was constrained to the following six teams:

- API Platform Team: This team manages the access to the APIs of the whole software infrastructure. The APIs are called trough REST by the services developed by other teams of ING software ecosystem.
- iDEAL Team: iDeal is an e-commerce payment system widely used in the Netherlands. As stated in [BV17]"iDEAL is not a centralised electronic payment system but a collection of technical agreements between banks and transaction processors". This team manages the portal that leads ING customers to make their payment through the iDEAL system. Their goal is to keep the portal up and running with the highest level possible of availability.
- Mobile API Platform Team: Among ING services, there is the mobile app for personal banking. MAP manages the API called from the mobile app.
- Cassandra Team: A middleware engineering team maintaining and updating Cassandra[1] distributed database. They manage the platform and guarantee the access to the software services that use the database.
- CIO Security NL Team: They define security guidelines and standards. Furthermore, they provide the other teams training and technology to ensure that their software complies with the regulations.
- Quality Engineering Team: They provide assistance to other teams to help them in defining better requirements and execute effective testing.

Q1) Which kind of methodology and technology do you use for requirements specification? (RQ1)

[1]http://cassandra.apache.org/

Four of the interviewees follow the Scrum framework defining requirements with User Stories and Epics. In particular, one of them created a document of 69 pages containing requirements shared with the other teams that make use of Cassandra platform. Another one is moving towards Gherkin scenarios for some projects. The rest adopted and advocate Specification By Example and BDD.

Q2) Can you describe how usually works the process of definition of a new software or feature? (RQ1)

Four respondents declared that ideas can come from every member of the team and stakeholders. Eventually, proposals are stored in product backlogs. One respondent organizes meeting with teams and their stakeholders to guide them in addressing the right questions on the domain in order to define examples of the usage of the product. Furthermore, the last respondent does not define requirements on a project basis but general security guidelines recommendations in the form of Gherkin scenarios.

Q3) Who are the typical stakeholders of your projects? Do they have IT background? (RQ1 - RQ2)

Business expert, managers and other teams are among the stakeholders of all the respondents. In addition, Custom Journey Experts(CJE) are stakeholders of three respondents. Business experts and managers do not always have an IT background, with the only exception of one respondent who stated that CJEs are part of team and have a good grasp of IT concepts. Stakeholders are also other teams of developers.

Q4) Do you think that the communication with them is effective? (RQ1-RQ2)

One respondent declared to be quite satisfied with the communication, while other three believe that improvements can be done. Business stakeholders do not often understand NFRs and to some extent they have to trust developers. In addition, one of them asserted to have witnessed several times communication issues among developers. Two interviewees claimed that it is not rare that a language barrier occurs even among developers with different backgrounds while another one believes that the communication becomes challenging when the amount of the teams working on a single project increases. Nonetheless, two respondents believe that Specification By Example and Gherkin Scenario are really helping a lot.

Q5) How do you define and specify non-functional requirements? (RQ2)

Two interviewees responded that NFRs are shared orally and usually not documented. Another one follows a checklist provided by guidelines and internal validation tools. Two believe that especially in BDD, NFRs are not really different from FRs. They define NFRs as test cases created with domain experts and, consequently, written as scenarios. One respondent uses Gherkin scenarios, Gatling script and Gitlab issues.

Q6) To what extent your stakeholders are involved in the elicitation, specification and testing of NFRs? (RQ1-RQ2)

Two respondents claimed that using BDD, stakeholders are almost completely involved. Another interviewee responded that the teams working with them execute their own tests and report to them what to improve. Instead, as also stated by the remaining respondents business experts and managers are little involved, except for demonstrations.

Q7) How do you verify and validate your non-functional requirements? (RQ2) One interviewee performs the tests suggested by the guidelines. In addition, several members of the teams cross-check the execution, the outcome of the tests and perform new ones if necessary. Two respondents validate requirements executing scenarios with Cucumber, therefore stakeholders can use them to validate the software on their own. Furthermore, one of them executes exploratory testing. The other three respondents develop code for testing and manually execute demonstrations. In addition, one of them runs Gatling scripts for performance requirements.

Q8) Which quality attributes are your team most concerned about? (RQ2)

Overall, respondents reported availability, reliability, security, integrity of data, confidentiality, performance and scalability.

Q9) Did you have issues with testing of NFRs in the past? How did you solve it? (RQ1-RQ2)

- Performance issues spotted too late in the development process. Gatling was adopted to provide a detailed picture of performance concerns.
- Browser incompatibilities and unexpected combination of issues in the same moment. They started assuming that failures may occur unexpectedly. Thus, they are designing more flexible and resilient code.
- The software was behaving differently on test data and production data. After, test data sample was created more accurately.
- Defining tests for security concerns was hard. BDD improved this process.
- Developers did not have clear expectations about NFRs. They perform tests without having defined requirement upfront.

Q11) How would you improve NFRs specification approach? (RQ2)

- A language that would guide the definition of resiliency requirements. They are already using BDD for some project but they find hard to express NFRs without a proper guideline.
- One responded declared to be satisfied even though he is aware that in the case a bigger amount of teams would be involved in the definition of requirements, the whole process will break down. However, they also think that documenting detailed requirements in advance is a waste of time, but it is more efficient developing solutions with vague requirements refining the product in future iterations.
- Their tooling does not allow to define use cases that are descriptive enough.

- There is no a standard way to work with BDD to integrate scenarios for a given domain in a project.
- NFRs should be written more. It is a big issue within the bank that can lead to not test enough.

Q12) How would you improve testing of NFRs? (RQ2)

- NFRs should be more embedded in Continuos Delivery(CD). CD is already widely used to test functional requirements.
- Faster testing. It is hard to recreate the ideal conditions where a test can be executed. For example, the environment has to be always reinitialized.
- If the use cases were more detailed, testing would be easier.
- It is hard to write test cases on a large scale test that can be tested by every engineer in every moment. They also adopted ZAP[2] for automating tests with BDD but they realized to have less control in the what was executed during the tests and the generated information.
- Teams should embrace more BDD.

The survey depicted that the six teams have a different relationship with NFRs. Quality constraints were a concern for every team, even tough with different degrees. There is a team with few new functionalities and well-known guidelines for the requirements focused towards providing stability and availability of the service. Therefore, stakeholders such CJEs are part of the team itself and the communication turns to be effective. Differently, another team reported problems of communications. In addition, other three teams highlighted the presence of misunderstandings among developers. The misunderstanding could increase proportionally with the number of teams involved, as pointed out by one respondent. One of the teams is dealing with this issue creating a shared documentation among the teams involved. Communication and collaboration improved significantly according to the teams in which NFRs specification and testing are main activities. However, other improvements can be done like the definition of a formal language and the creation of models for common scenarios.

[2]https://www.continuumsecurity.net/bdd-security/

Challenges	Extracted from	Rationale
C5: A formal language that helps the definition of NFRs in BDD should be defined.	(Q11),(Q12)	"Formal" in this context is intended as a language which guides the definition of quantifiable and measurable requirements with descriptive scenarios
C6: Alignment of requirement specification and testing with the production	(Q7),(Q12)	Specification and testing of NFRs are considered an overhead for the release of a software when the time is limited. A solution is needed to make these activities more aligned with software production.
C7: More automation in testing of NFRs	(Q5),(Q12)	Adoption of CD and CI allow higher software quality and shorter time to market. More test automation allows embedding the verification of NFRs with CD and CI
C8: More control over test automation of NFRs	(Q12)	the solution adopted in the past for automation of testing in NFRs generated a lot of data and executed code that was hard to debug and control. The process turned to be inefficient. New solutions to increase customization of automatic testing is needed
C9: Language barrier and background diversity	(Q4),(Q6)	Communication can be ineffective among developers with different backgrounds. Specification and testing of NFRs require deep knowledge of the domain. Scenarios can be used to improve awareness and avoid misunderstandings.
C10: Stakeholders involvement	(Q4)	The complexity of NFRs leads to the consequence that sounding proof of verification and validation of NFRs are not always provided to the stakeholders, requiring them to trust the developers. Moreover, stakeholders could not be aware of which quality requirements their products need.
C11: Collaboration	(Q1),(Q11)	Specifying requirements is unavoidable when several teams are involved in the process. An approach adopted is the creation of shared documentation among the teams. However, documentation can be hard to maintain and require an overhead of time.

Table 3.4: Challenges emerged from the survey.

Chapter 4

Designing the framework

4.1 Purpose of the Framework

The previous chapter explored the state-of-the-art of quality requirements specification methodologies and described the findings obtained in semi-informal interviews of ING experts. The knowledge acquired in the previous steps allowed to lay the foundations for the architecture of the framework. Some of the challenges emerged from Table 3.4 and Table 3.2 are:

- Complexity: Quality constraints tend to be hard to grasp and to explain and, therefore, often ignored. Furthermore, it is possible that the need of a NFR become clear only in the late stages of a project;
- Poor specification: Documentation and knowledge about NFRs are sometimes only shared by oral communication, reports or stored separately from functional specification. This can impact the traceability of requirements.
- Automation: While Cucumber helps in driving and automating the testing process for FRs, this is often not the case of NFRs
- Lack of non-functional specification: Tendency in testing quality concerns without a previous definition in the form of NFRs, with the risk that serious potential issues could be uncovered.
- Communication: Interaction with stakeholders are not always effective if their background is not IT-related. In addition, communication may be an issue even among developers.
- Validation & Verification: Sounding proof of verification and validation of NFRs are not always provided to the stakeholders, requiring them to trust the developers. On the other hand, stakeholders could not be aware enough whose quality requirements their products need.
- Formal language: NFRs are often vague and ambiguous. In this context "formal language" refers to a language that describes correct and measurable requirements. A formal language can reduce misunderstandings.

The framework will be designed to handle these and other challenges. A detailed explanation of how the framework addresses them is shown in Section 6.2 Along the

same lines of Cucumber, the framework will be named TomatoFramework. In the following sections, the design decisions will be described.

4.2 Architecture

The first step of software creation consists of the communication of the requirements to the team members to create a shared understanding. BDD can guide the definition of features. However, not having an agreement in the terminology and the language concerning NFRs can undermine the consistency of the shared awareness among members. A solution is the use of a formal language.

In languages, symbols are combined according to a set of rules called syntax. Every legal combination of symbols assumes a meaning and the set of meanings is called semantics. The combination of syntax and semantics represents the language. A grammar defines boundaries in the space of what can be expressed. In TomatoFramework this is, both, a limit and an advantage. The limit lies in the implicit abstractness of quality constraints, e.g. "what makes a system secure?", which makes hard defining a unique non-ambiguous grammar that fits every domain. However, this limit can be partially bypassed providing the system with the capability to be customized and, therefore, to adapt to the users and the domain. The advantage, instead, consists of the fact that sentences are finite and therefore it is easier to provide instructions to the system on how to interpret and elaborate them. This constitutes a fundamental step to ensure automation.

Cucumber encourages the use of given-when-then scenarios to write requirements. TomatoFramework embraces this approach from a different perspective. It guides the definition of correct steps for NFRs, automating the generation of the relative step definitions. The process is divided into two parts that can be executed in different moments. Firstly, it explores and elaborates a given grammar to generate glue code. Secondly, it parses the specification of requirements in the form of scenarios. Trough the glue code, TomatoFramework interprets the NFRs and performs the actions that bring to a deterministic measure of the fulfilment of the requirement.

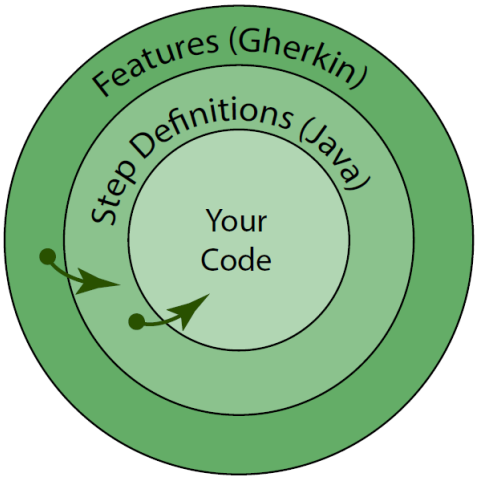

Figure 4.1: Layers wrapping a typical software designed and tested with Cucumber [WH12]).

Figure 4.1 shows the structure that developers need to follow in the design of their software using Cucumber. Scenarios are described inside Gherkin files that are, subsequently, linked with the steps definition and executed by Cucumber. Any possible implementation must coexist with this scheme. So, quality requirements should be expressed following the Given-When-Then paradigm.

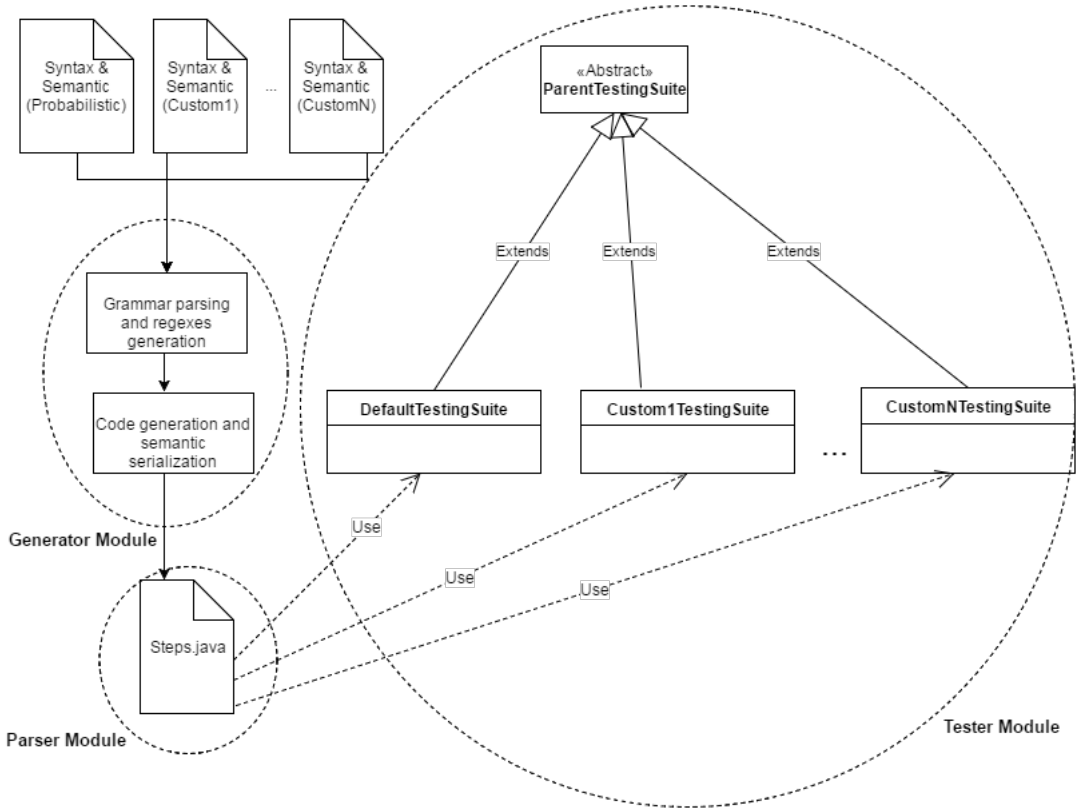

Figure 4.2: Abstract architecture of TomatoFramework

As shown in figure 4.2, the framework is composed of the following modules:

- Generator Module: This module takes as input a grammar defined in sets of production rules and their semantics. The main task of the Generator Module is executing controls on the correctness of the language and, consequently, generating the Parser Module.

- Parser Module: It consists of a file containing the step definitions called by Cucumber trough regular expressions. The regexes match all the legal sentences of the formal language. The Parser Module must be placed in the glue code of the projects that use TomatoFramework.

- Tester Module: This module executes testing, generates reports or glue-ware. It supports the definition of rules on the parameters and the customization of the information provided in the reports. Therefore, the main tasks of the Tester Module are the communication with the Parser Module and the interpretation and execution of the commands received. In addition, the Tester Module "observes" the execution of Cucumber to store the methods necessary for the testing. The Tester Module can be extended with custom test suites to address new domain and grammars. However, a default implementation is provided. The Default Testing Suite will be described in section 4.4.

Before going any further, let us focus on the structure of the grammar that a custom formal language provided as input to the Generator Module should follow.

A grammar can be formed by a syntax and a semantic. The syntax regulates the correct usage of the symbols that are part of the language. In TomatoFramewok, the correctness of a language is governed by a set of production rules. Moreover, a semantic must be provided to instruct TomatoFramework on which operations to perform. Therefore, if the properties of quality attribute domain can be expressed through basic relations of equality and inequality represented by the symbols $=, <, >, \leq$ *and* $\geq$ then a syntax can be built and an automation based on the semantic can be developed.

The power of Behaviour-driven development is the possibility of expressing scenarios in English. Therefore, a grammar defined for a quality attribute should be made of expressions used in natural language. However, a spoken language differs from a programming language for several reasons. One reason is that in a spoken language the meaning of a sentence can be strictly related to the context in which that sentence is expressed. Nevertheless, there is evidence that English is not a context-free language, as stated in [Hig84]. Another reason consists of the expressiveness of the natural language that allows defining the same concept with many alternatives. Concerning the possibility of expressing the same concept with different forms, TomatoFramework permits grammars divided into two sets of production rules. The first set, called low-level set, contains the lexical representations of the operators that add a semantic, for example, the sentences "is equal to" and "is exactly" correspond to the operator "=". The second set, called requirements model set, contains rules and additional words to make the sentences more readable. The basic structure of the grammar is defined with the low-level set and should be modified only when new rules are introduced, while new alternative expressions in the requirement models set must be added easily by the users. In Figure 4.5, the first two rows from the top represent the requirements model set and the remaining rows are part of the low-level set.

After that requirements model set and low-level set are created and validated by the framework, the regular expressions that match the sentences defined by the grammar are generated, concurrently to their step definitions. The semantics previously declared works as a protocol of communication between the test suite and the Gherkin file. Indeed, the test suite receives the constraints and translates them into commands.

TomatoFramework should be flexible enough to include the definition and the verification of NFRs coming from different domains. For this reason, a grammar and a Tester Module consist of, respectively, a specification language of a given domain and the operations triggered by expressions of that language. In Section 4.3 a general execution of TomatoFramework it is described without referring to any specific domain of NFRs. Instead, in Section 4.4 it is shown the execution with a real implementation of a Tester Module called Default Test Suite and a grammar called Probabilistic.

4.3 Description of a general execution

The Generator Module analyses the grammar and verifies its correctness. Consequently, the Generator Module digs in each expression defined in the requirement

models set using the production rules of the low-level set, creating a tree containing all the possible sentences generated from a requirement model. Then, it reduces the whole tree of possible derivations of a requirement model to a regular expression. A regular expression is created for each requirement model. In addition, the Generator Module stores the semantics of the non-terminal symbols that encounters. Examples of semantics are the symbols "=,<,>" or types String and Integer. At this point, the Generator Module proceeds in producing the code of the Parser Module. For each regular expression, a function is created. Within these functions, the Generator Module inserts the instructions to identify the semantics of the symbols handled by the regular expressions. It is worth noticing that the generation of the Parser Module is needed only when the grammar is created or updated.

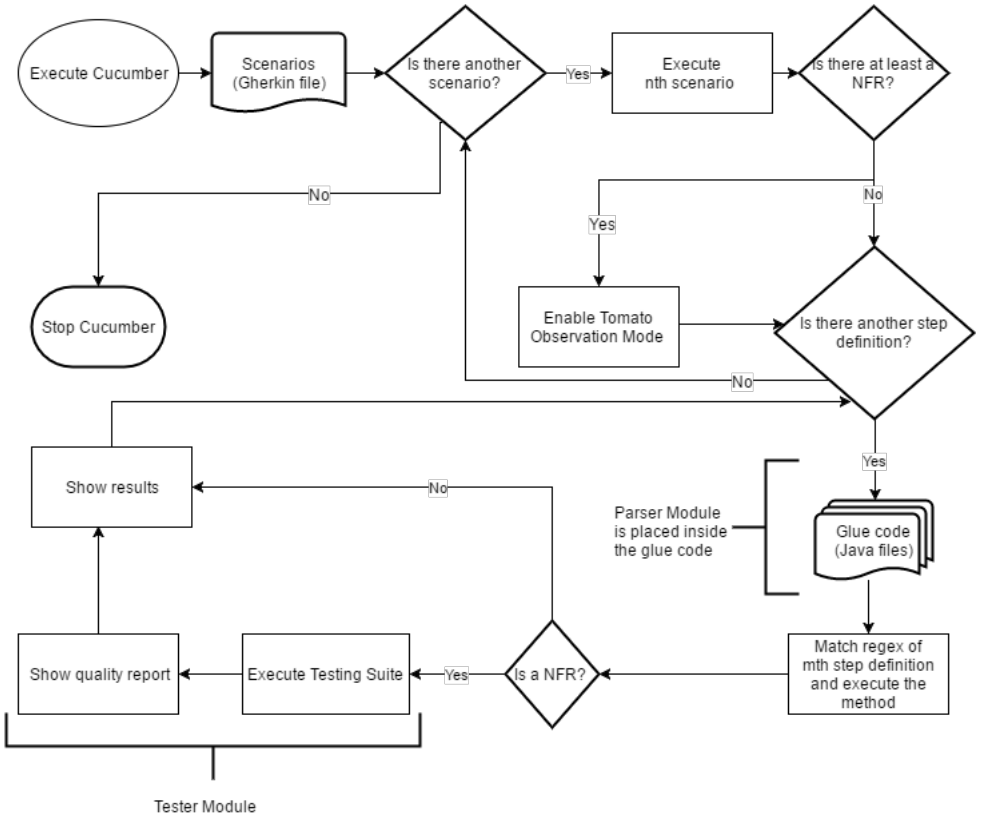

Figure 4.3: Flow chart of execution of Tomato Framework during the parsing phase and testing phase

In figure 4.3 is described the whole phase of parsing of a Gherkin file. For the sake of simplicity, in the figure is shown a typical execution of the framework without taking into account possible exceptions. The technical details are omitted and more information is provided in the next chapter. When launched, Cucumber parses all the scenario within the Gherkin files. The methods placed in the glue code of the software under test are called when a step matches one of their regular expression. Therefore, before that Cucumber executes a method inside the Parser Module, TomatoFramework transparently observes and stores all the previously executed methods which are part of the scenario and all the variables that may influence the execution of Scenario. This is what in the figure is defined as "Observation Mode". After that a quality constraint in the Gherkin file is matched, the framework enters in the "Testing Phase". After displaying the result of the test,

Tomato gives back the control to Cucumber returning the result of the function linked with the current step definition. Cucumber iterates the process until every step of all the scenarios is parsed. In the Testing Phase, TomatoFramework activates the Test Suite linked with the grammar of the current step. The methods and the variables stored during the executions of the previous steps are retrieved. At this point, various options are available. For example, it is possible to re-execute sequentially all the steps modifying the parameters with random values or custom values. Custom operations can be performed at this stage.

4.4 Default Test Suite

As a starting point, common quality attributes as reliability, availability, performance, safety and security were taken into account. It was necessary to understand how the requirements related to this set of quality attributes can be expressed. This analysis was carried out by [Gru08], who executed a comprehensive literature review about verification techniques of quality attributes with an intrinsic probabilistic nature. Indeed, many quality properties are expressed in terms of the probability that a given state of the system will hold within given time constraints. Grunske developed a set of specification patterns called ProProST [Gru08] that contains formal probabilistic logic the mathematical representation of the patterns. A structured grammar in English was implemented from ProProST to provide to practitioners an instrument that would have helped them in employing formal verification techniques.

Start	(1) probabilisticProperty	:== "The system shall have a behavior where" *patternStateFormula*
ProProST Pattern	(2) patternStateFormula	:== "with a probability" *probabilityBound* "it is the case that" *occurenceStateFormula*| *orderStateFormula*
	(3) occurenceStateFormula	:== *transientStateProbability* | *steadyStateProbability* | *probabilisticInvariance* | *probabilisticExistence*
	(4) orderStateFormula	:== *probabilisticUntil* | *probabilisticPrecedence* | *probabilisticResponse* | *probabilisticConstResponse*
	(5) transientStateProbability	:== *stateFormula* "holds after exactly" $t \in \mathbb{R}^{\geq 0}$ "time units."
	(6) steadyStateProbability	:== *stateFormula* "holds in the long run."
	(7) probabilisticInvariance	:== *stateFormula* "holds continuously" *timeBound* "."
	(8) probabilisticExistence	:== *stateFormula* "will eventually hold" *timeBound* "."
	(9) probabilisticUntil	:== $stateFormula_2$ "holds" *timeBound* "after" $stateFormula_1$ "has held continuously before".
	(10) probabilisticPrecedence	:== $stateFormula_2$ "holds before" $stateFormula_1$ "can hold" *timeBound* "."
	(11) probabilisticResponse	:== "if" $stateFormula_1$ "holds, then as a response" $stateFormula_2$ "becomes true" *timeBound* "."
	(12) probabilisticConstResponse	:== "if" $stateFormula_1$ "holds, then as a response" $stateFormula_3$ "becomes true" *timeBound* "without" $stateFormula_2$"holding in between."
Time	(13) timeBound	:== *noTimeBound* | *upperTimeBound* | *lowerTimeBound* | *timeInterval*
	(14) noTimeBound	:== ""
	(15) upperTimeBound	:== "within" $t \in \mathbb{R}^{\geq 0}$ "time units"
	(16) lowerTimeBound	:== "after" $t \in \mathbb{R}^{\geq 0}$ "time units"
	(17) timeInterval	:== "between" $t_1 \in \mathbb{R}^{\geq 0}$ "and" $t_2 \in \mathbb{R}^{\geq 0}$ "time units"
Probability	(18) probabilityBound	:== *lowerProbBound* | *lowerOrEqualProbBound* | :== *higherProbBound*| *higherOrEqualProbBound*
	(19) lowerProbBound	:== "lower than" $p \in \mathbb{R}^{[0.1]}$
	(20) lowerOrEqualProbBound	:== "lower or equal than" $p \in \mathbb{R}^{[0.1]}$
	(21) higherProbBound	:== "greater than" $p \in \mathbb{R}^{[0.1]}$
	(22) higherOrEqualProbBound	:== "greater or equal than" $p \in \mathbb{R}^{[0.1]}$
Final	(23) stateFormula	:== SIMPLESTATEPROPERTY
	(24) $stateFormula_1$	:== SIMPLESTATEPROPERTY
	(25) $stateFormula_2$	:== SIMPLESTATEPROPERTY
	(26) $stateFormula_3$	:== SIMPLESTATEPROPERTY

Figure 4.4: Structured English grammar in natural language to express probabilistic patterns provided by [Gru08]).

Nevertheless, the grammar proposed can be hard to understand and to use by

somebody who does not have a mathematical or theoretical computer science background. A further step forward accomplished in this direction can be found in [BE11], new rules are built on top of the one introduced by Grunske.

probabilisticInvariance	:==	'the system shall have a behavior where with a probability' probabilityBound 'it is the case that' stateFormula 'holds continuously' timeBound
alternativeOne	:==	'the' probability of stateFormula timeBound 'is' probabilityBound
alternativeTwo	:==	'the' probability 'is' probabilityBound 'that' stateFormula timeBound
alternativeThree	:==	'with' ['a'] probability ['of'] probabilityBound stateFormula timeBound
alternativeFour	:==	'there is' ['a'] probability ['of'] probabilityBound of stateFormula timeBound
alternativeFive	:==	'with' aProbabilityBound probability stateFormula timeBound
alternativeSix	:==	'there is' aProbabilityBound probability of stateFormula timeBound
alternativeSeven	:==	stateFormula timeBound ['in'] probabilityBound ofTheTime
alternativeEight	:==	'in' probabilityBound ofTheTime stateFormula timeBound
alternativeNine	:==	stateFormula timeBound 'with' ['a'] probability ['of'] probabilityBound
Probability	:==	'probability' | 'chance'
probabilityBound	:==	(atBound | thanBound) p ∈ $R^{[0,1]}$
aProbabilityBound	:==	(atBound ['a'] | ['a'] thanBound) p ∈ $R^{[0,1]}$
atBound	:==	'at most' | 'at least'
thanBound	:==	lowerOrEqualThan | higherOrEqualThan | greaterThan | lowerThan | equalTo
lowerOrEqualThan	:==	('lower' | 'less') ('or equal than' | 'than or equal to')
higherOrEqualThan	:==	('greater' | 'higher') ('or equal than' | 'than or equal to')
greaterThan	:==	'greater than' | 'higher than'
lowerThan	:==	'lower than' | 'less than'
equalTo	:==	['equal to']
of	:==	'of' | 'to' | 'that' | 'in which'
timeBound	:==	upperTimeBound | lowerTimeBound | timeInterval | noTimeBound
upperTimeBound	:==	('within the next' | 'in less than' | 'in at most' | 'within' | 'in' | 'before') t ∈ $R^{>0}$ timeUnits
lowerTimeBound	:==	('after' | 'in more than' | 'in at least') t ∈ $R^{>0}$ timeUnits
timeInterval	:==	'between' t ∈ $R^{>0}$ 'and' t ∈ $R^{>0}$ timeUnits
noTimeBound	:==	''
timeUnits	:==	'time units' | 'time steps'
ofTheTime	:==	'of the' ('cases' | 'time')
stateFormula	:==	SimpleStateProperty

Figure 4.5: Structured English grammar in natural language to express probabilistic patterns provided by [BE11]).

The grammar of [BE11] introduces nine alternatives that can be reduced to recurring sentences in the English language used in the definition of quality requirements. Their rules are shown in Figure 4.5

For example, a performance constraint can be in the form of "The probability of a correct response of the system between 3 time units and 6 time units(milliseconds) is at least 80%". The sentence is extended from alternativeOne in form of "the {probability} {of} {stateFormula} {timeBound} is {probabilityBound}. Furthermore, an example of availability concern is the amount of time that a system should be up and running. Therefore, the NFR can be "there is at least a probability of 99% that the system is operative without errors after 12 time units(months)" that is extended from alternativeSix in the form of "there is {aProbabilityBound} {probability} {of} {stateFormula} {timeBound}"

However, the two structured grammar allow to express only a limited set of expressions. In addition, there are terms used only in certain domains. Thus, a grammar too generic can lead to misunderstandings. For this reason, TomatoFramework should be versatile enough to permit to be extended and configured by the user.

4.4.1 Extending the Default Test Suite

As pointed above, the Default Test Suite does not contain testing techniques focused on specific quality attributes. Therefore, it was expanded with specific solutions for some of the quality attributes mentioned by the interviewed in Section 3.2 namely

performance, reliability, availability and scalability. The metrics and techniques that are going to be introduced in this section are not intended to constitute an exhaustive overview of all possible solutions in measuring the quality properties of a software. Rather, the purpose is providing insights of the potential of the framework. Metrics and techniques can be used specifying them in requirements following the templates provided by their grammars. The implementation will be shown in chapter 5.

Reliability

As stated in [KB14], the reliability of a software consists of "the probability of the failure-free software operation" in a given range of time in a defined environment. [KB14] provided the following metrics:

- Mean time to failure (MTTF): MTTF is defined as the time interval between the successive failures.
- Mean time to repair (MTTR): MTTR measures the average time it takes to track the errors causing the failure & to fix them.
- Mean time between failures(MTBF): MTBF = MTTF + MTTR
- Rate of occurrence of failures (ROCOF): It is the number of failures occurring in a unit time interval.
- Probability of failure on demand (POFOD): POFOD is defined as the probability that the system will fail when a service is requested. It is the number of system failures given a number of systems inputs.

When a scenario involving a reliability constraint is executed, the execution time of the steps interested in the assessment is measured and the error exceptions are tracked. Then, the value of MTTF, ROCOF and POFOD can be calculated. MTTR and MTBF can be elaborated from a log tracking exceptions over time. A grammar will be provided only for MTTF, ROCOF and POFOD given that there is no need to express MTTR and MTBF in a specification file. However, the visualisation of the values can be enabled. Therefore, TomatoFramework will allow to show reliability reports without the need of specifying constraints.

Availability

In [KB14], availability is defined as "the probability that the system is available for use at a given time. It takes into account the repair time and the restart time for the system". Many metrics are used when measuring availability. Nevertheless, in this context, only Operational Availability will be taken into account.

- Operational Availability (AVAIL) = MTBF / (MTBF+MTTR)

Therefore, repair time assumes a main role in the measuring of AVAIL. Repair time is calculated from when a failure happens until the moment that it has been fixed. For this reason, a log that keeps track of the evolution of the failures is needed. Availability and reliability aspects often refer to scenarios over a long time, where the system can be interested in several modifications. For example, an integration of a new component can lead to unexpected behaviour to legacy components even if a

direct relationship is not evident. Automating the storage of information regarding failures permits the generation of snapshots describing the evolution of the scenarios in production.

Performance

Performance testing consists of a "Black Box" testing, which measures the system behaviour from outside, without inspecting the code within the system. The main objectives of the performance testing are:

- Maximum number of concurrent users that can be supported while offering "acceptable performance."
- Maximum number of concurrent users that can be supported prior to causing a system failure.
- Location of bottlenecks within the application architecture.
- Impact of a software or hardware change on the overall performance of the application.
- Scalability issues

Several kinds of metrics allow to measure the aspects listed above:

- System metrics: Set of parameters measuring the performance of internal components of the system, i.e. maximum connections to a database available at the same time or maximum number of processes admitted in a software system.
- Resource metrics: Every system needs to use and consume a set of resources in order to produce output. The most trivial example is the one of a normal personal computer. In order to make an elaboration, the computer needs a power supply, CPU, RAM, hard disk and so on. Resource metrics express the usage of hardware and software assets.
- Workload metrics: These parameters are directly related to the interaction that the system has with the users or external systems. They are used to provide a picture of how the system reacts to a massive flow of inputs over a long period time. Examples are calls to an API per day and authentication requests to a server per day.

From now on only workload metrics will be taken into account. System metrics and Resource metrics are tracked on a deeper level during development and their values are an indirect consequence of high level domain-dependant requirements. In addition, these metrics need a deeper understanding of the subject and the impact that they have on the system falls outside of what a typical non-IT stakeholder can formulate. In contrast, workload metrics depend on the business logic to the system. For instance, their thresholds can vary a lot if the system to develop is either a web application for buying products or a website hosting an encyclopedia. The challenge of the requirement engineer is understanding and formulating which values these metrics should have.

In order to calculate workload metrics, typical techniques are:

- Load test: The aim of the load test is recreating the workload that the system under test will experience during its regular life cycle. Metrics are used to measure the reaction of the system when certain conditions happen in a small time window.

- Stress test: Stressing a system means bringing that system to the worst possible scenario. In such a way, the weakest ring of the chain is identified and the conditions that make it break are highlighted.

- Resistance or Endurance test: It is not rare the case where a certain scenario causes a bug appearing only after an unpredictable period of time even if, apparently, that scenario has been executed tonnes of times without issues. The nature of these bugs makes them hard to spot with classic strategies. Resistance (or endurance) tests are used to apply load tests and stress test over an extended time span.

TomatoFramework will support the execution of the above-mentioned techniques. They will be executed by expressively defining constraints on the feature file, or by enabling them with method calls. In addition, the framework will execute the scenarios parallel to simulate more realistic situations.

Scalability

The definition of Scalability varies dependently to which context refers. Indeed, Scalability can be measured in various dimensions, such as size, geographical and administrative. In this context, scalability refers to size scalability. As said in [Van04] "A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance." In a more general way, scalability measures also how the system reacts to a growth or reduction in a number of inputs that users or other system generate.

Scalability was mentioned above among the objectives of performance testing. The reason is that if a system has a low degree of scalability, the effects are primarily shown in the performance of the system. Therefore, metrics and techniques used for performance testing can be reused to create an overview of the scalability of the system.

Chapter 5

Implementation

In this chapter, the components of TomatoFramework are described in more detail and their implementation is shown.

5.1 Generator

The Generator Module is composed of two classes, namely the *Grammar* class and the *Generator class*.

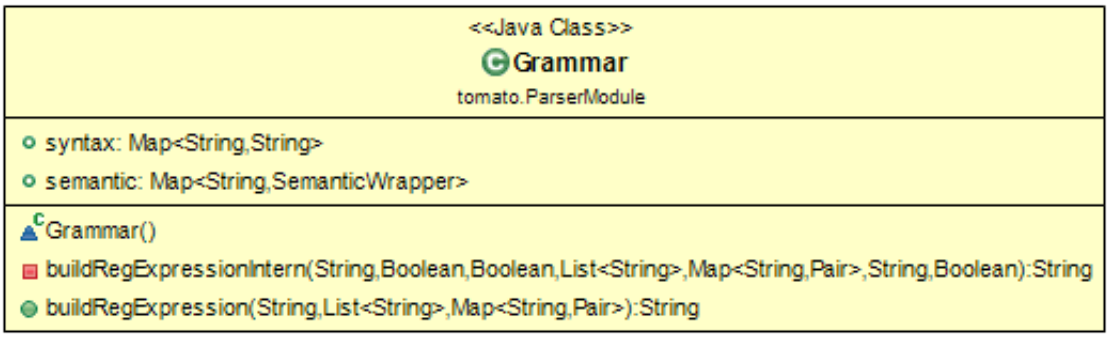

Figure 5.1: The Grammar Class

The Grammar class contains two hash maps. The first map called *syntax* consists of pairs of expressions, representing the production rules of the grammars. The second map is called *semantics* and contains the mapping between the expressions and their meaning. An expression can be represented by a string or a class. The class *SemanticWrapper* is used to hide the two types in order to implement a map with a different kind of value. In addition to *syntax*, there is another set of productions rules named *requirement models*. A requirement model is the starting point of the generation of regular expressions. The task of the function *buildRegExpression* is iterating on every requirement model, applying recursively all possible production rules on each part of the sentence. From the resulting tree, the equivalent regular expression is returned. Every regular expression will be associated with a step definition method. Both, step definition and regular expression will be written in a file generated with Freemarker[1]. This file is the Parser Module.

[1] http://freemarker.org/

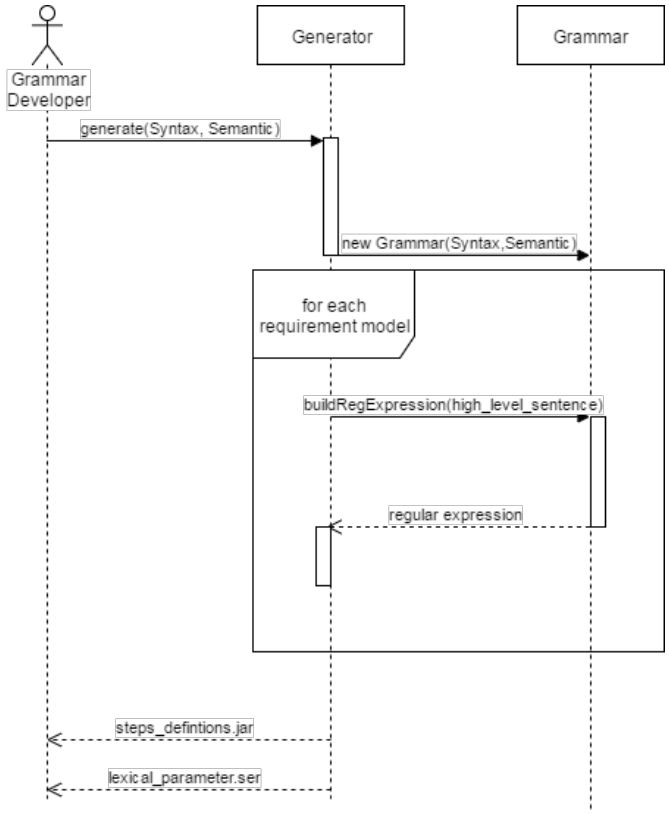

Figure 5.2: Sequence diagram of the generation phase

5.2 Parser

The Parser Module must be placed within the glue code of a project. Therefore, it must follow the same structure of any step definition file, that is a java class with several methods and no constructors. Every method has a regular expression annotated with *@given*, *@when* or *@then*. Step definitions are launched when Cucumber parses a step that matches with the associated regular expression. Therefore, a method of the parser module is executed when a NFR is matched.

5.2.1 Specification of Non-Functional Requirements

In 5.1, an example of scenario is displayed. The scenario describes two clients trying to send a message to a server inside an intranet. The last step, recognisable by the square brackets, is a NFR. Square brackets are necessary to avoid ambiguities. In fact, when a step matches multiple regular expression Cucumber raises an exception. Given that TomatoFramework generates the regular expressions automatically, testers could have less control in the definition of their regexes. Indeed, regular expressions of the parser module reduce the set of available alternatives. The NFR in the example originates from the probabilistic grammar, introduced in the section **??**. In the example the parameters passed to the Parser Module are the stateFormula "the server receives all the messages", the desired probability of success and execution time. A stateFormula identifies the condition that a quality constraint must fulfil and it can be mapped with a method defined ad-hoc by the developer or with one or more steps definition.

```
Feature: Messages exchange
    @NFR
    Scenario: Server messages reception
    Given 2 clients accessed the intranet
    When every client sends a message to the server
    Then the messages are delivered
    And [the probability that "the server receives all the
       messages" in less than 4.0 seconds is at least 90%]
```

Listing 5.1: A scenario showing several clients trying to contact the server at the same time

The methods *initializationPhase* and *finalizationPhase* of the Parser module are executed before and after every scenario containing quality constraints.

```
@Before("@NFR")
public void initializationPhase(Scenario scenario){
    this.scenario=scenario;
    this.ptt = new DefaultTestingSuite();

    ptt.assignStateFormula("the server receives all the
       messages", "clients_accessed_the_intranet");
    ptt.assignStateFormula("the server receives all the
       messages", "every_client_sends_a_message_to_the_server");
    ptt.assignStateFormula("the server receives all the
       messages", "the_messages_are_delivered");
    ptt.assignRuleRandom("the server receives all the messages",
       "clients_accessed_the_intranet", "numberOfClients", 2,
       50);

    this.ptt = ReliabilityNature.safelyDecore(ptt);

    ((ReliabilityNature) ptt).enableReliabilityReport();

    tmm = ptt.getMessenger();
   }
```

Listing 5.2: Initialization code

In Listing 5.2, the code of initialization is displayed. It shows the instantiation of the test suite and the mapping of the stateFormula. The test suite is chosen from the Generator Module depending on the grammar of the NFR. The stateFormula corresponds to the whole scenario. Thus, the stateFormula is considered true if every step definition is launched without exceptions and the overall execution time is within the range specified in the NFR. The stateFormula can be mapped also with a method external the scenario. In that case, it is required that the return type of the method is boolean instead of void.

Testing functional requirements means executing the operations providing that functionality, checking whether they are correctly performed and that the outcome matches the expectations. Another approach is needed for non-functional testing. The time scope of a quality constraint may enclose several statuses of the system under test or even its entire lifetime. Often, a constraint should remain valid despite

variation of contexts. An example can be the response time of a cluster of servers that must be available more than a certain time threshold even during abnormal request peak or when one of the servers shuts down unexpectedly. Therefore, the task of a non-functional tester is to identify the contexts that lead to unwanted behaviours. Unfortunately, tonnes of different factors such as inputs, resources, time, may impact the execution of a system. Furthermore, combinations of factors may lead to additional unpredicted scenarios. In order to understand how a system reacts when the factors are too many or too complex, it is needed a solution that makes possible to define the boundaries of the possibilities letting a test software simulate the consequences.

In TomatoFramework, a step in this direction is made with *Rules*. Rules are applied on the variable elements of a scenario. TomatoFramework will execute the same scenario over and over again modifying the parameters according to the rules. Trough the rules the framework can be instructed to change variables randomly, acquire them from external sources, compute them with lambda functions or generate the with algorithms. A similar mechanism is used in property-based testing[VA16] with tools like QuickCheck[2]. In the above snippet of code, a random value is assigned between 2 and 50 the argument numberOfClients of *void clients_ accessed_ the_ intranet(int numberOfClients)* at each iteration.

The method of Listing 5.2 is called when Cucumber parses a scenario with a tag @NFR. Tags can be used to link one or more scenario in the feature file to methods of the glue code. Another usage is the definition of filters, labels and hooks. More information on the usage of tags can be found in [WH12]. It is worth noticing that assigning all stateFormulas and rules on the initialization phase can slightly decrease the overall performance of the execution of Cucumber. In fact, stateFormulas and rules are assigned every time a scenario with the tag @NFR is parsed. Dependency injection can help overcoming the issue. To provide a better modularization and organisation of the code, TomatoFramework, like Cucumber, supports the use of dependency injectors[Pra09]. PicoContainer[3] is supported by default.

5.2.2 The Messenger and the Observer Module

The last operation in Listing 5.2 consists of the instantiation of the Messenger. The Messenger conveys the data needed to perform the tests. In other words, it enables the communication between the different modules. In order to keep a loose coupling of the elements and, thus, to make them easier to maintain, during the execution every module can send its data only to the Messenger.

The flow of the data passing trough the messenger is shown in Figure 5.3. The elements involved are the Parser Module, the Tester Module, which will be described in detail in the next section, and the Observer Module. The Observer Module runs in the background during the parsing phase. It is built with AspectJ[4], that provides to the Observer module the capability of handling the execution of Java methods and their parameters at run-time. The module is triggered by the execution of the method *initializationPhase*. After, Cucumber will proceed its normal execution call-

[2]https://hackage.haskell.org/package/QuickCheck
[3]http://picocontainer.com/
[4]https://eclipse.org/aspectj/

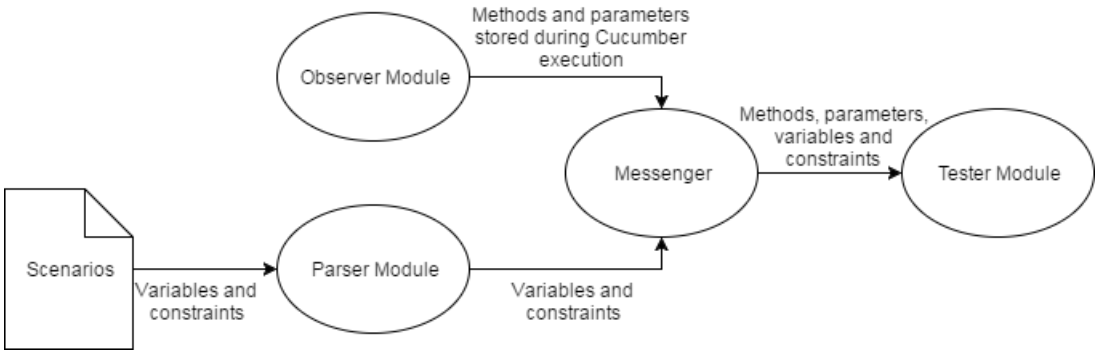

Figure 5.3: Architecture of the data flow from and to the Messenger

ing step definitions for each step of the scenario. In the background, the Observer module will store every step definition and relative parameters launched with Cucumber trough reflection[5]. Consequently, prior to executing the actions to assess the NFR, the Observer Module will insert the methods and parameters gathered in the messenger. Then, the messenger is passed back to the Tester Module. Lastly, the Tester Module owns all the necessary data to proceed with non-functional testing.

```
@Given("^\\[the (?:probability|chance) (?:of|to|that|in which)
    \"([^\"]*)\" ((?:within the next|in less than)|(?:after|in
    more than)) (\\d+\\.\\d+) s is ((?:at most|(?:lower than|less
    than))|(?:at least|(?:greater than|higher than)))
    (\\d+\\.\\d+)\\]\$")
public void alternativeOne(String stateFormula,String
    timeBound,Double t,String probabilityBound,Double p) throws
    Throwable {
     tmm.insertParameter("stateFormula", stateFormula);
     tmm.insertParameter("timeBound", timeBound);
     tmm.insertParameter("t", t);
     tmm.insertParameter("probabilityBound", probabilityBound);
     tmm.insertParameter("p", p);

     if(this.ptt.invokeTestSuite(tmm))
        System.out.println("Constraint fulfilled");
     else {
         Assert.fail("Constraint not fulfilled");
         }

     tmm.cleanParameters();
}
```

Listing 5.3: A step definition in the parser module. The regular expression was derived from alternativeOne in Figure 4.5

[5]https://docs.oracle.com/javase/tutorial/reflect/

5.2.3 Non-functional step definitions

A step definition of the Parser Module connects a set of alternatives forms derived from a single requirement model to the Tester Module that will interpret them and make the necessary operations to assess the fulfillment of the condition stated in the NFR. Therefore, we can refer to the step definitions of the Parser Module as "non-functional step definitions".

Listing 5.3 shows a non-functional step definition and the annotated regular expression. Variables and constraints are extracted by the operator "capturing group" of the regex engine. Consequently, they are inserted in the messenger. Next, the method *invokeTestSuite* activates the execution of the test suite related to that requirement model. Furthermore, methods and parameters of the previous steps stored by the Observer module are inserted in the messenger at run-time before the invocation is completed.

5.3 Tester

The Tester Module selects and executes the set of tests necessary to assess the constraint. The outcome of the execution of the tests can be metrics, logs or reports. Moreover, the Tester module checks the fulfilment of the NFR raising an *assert* exception if it is not respected.

The Tester Module is composed of several classes called Test Suites. A Test Suite can be coupled with a set of expressions or none of them. Therefore, test suites can be invoked by steps definition or directly in the code. TomateFramework offers a default test suite and permits to define custom ones.

The Default Test Suite enables the recognition of NFRs derived from the default structured grammar. It provides the methods needed to apply, and consequently, to verify the logical formulations extracted from the specification patterns of Grunske. The Observer Module mentioned in the previous section makes this possible, providing to the Default Test Suite the data and the methods necessary to recreate the execution of the scenarios. Hence, a total control on the execution flow of every scenario is given to Default Test Suite. A list of snapshots containing a representation of each iteration of a scenario is kept in memory and updated when the Default Test Suite launches the successive step definitions. Eventually, checks on constraints are carried out within the snapshots.

As pointed out in the previous chapter, customization and adaptability of the framework are a primary concern. It must be possible to introduce easily new grammars and expressions. In the same way, the operations related to their verification should be added efficiently. For this reason, the methods that constitute the infrastructure of the Tester Module were gathered in an abstract class called *ParentTestSuite*. Every test suite, included the default test suite, must inherit from it. Figure 5.4 shows the class diagram of *ParentTestSuite*. It is demanded to this class the instantiation of the messenger, the assignment of stateFormulas and rules, and the application of the rules on the variables. In addition, it contains the abstract method *invokeTestSuite*. This method represents the only admitted way to invoke the test suites. Therefore, every operation that implements the execution of the test suite must be defined extending *invokeTestSuite*.

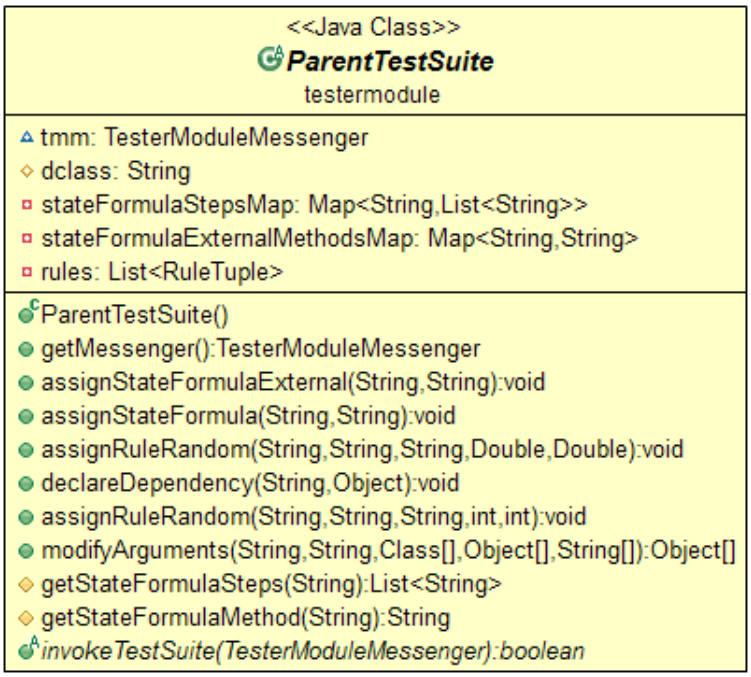

Figure 5.4: Class Diagram of ParentTestSuite

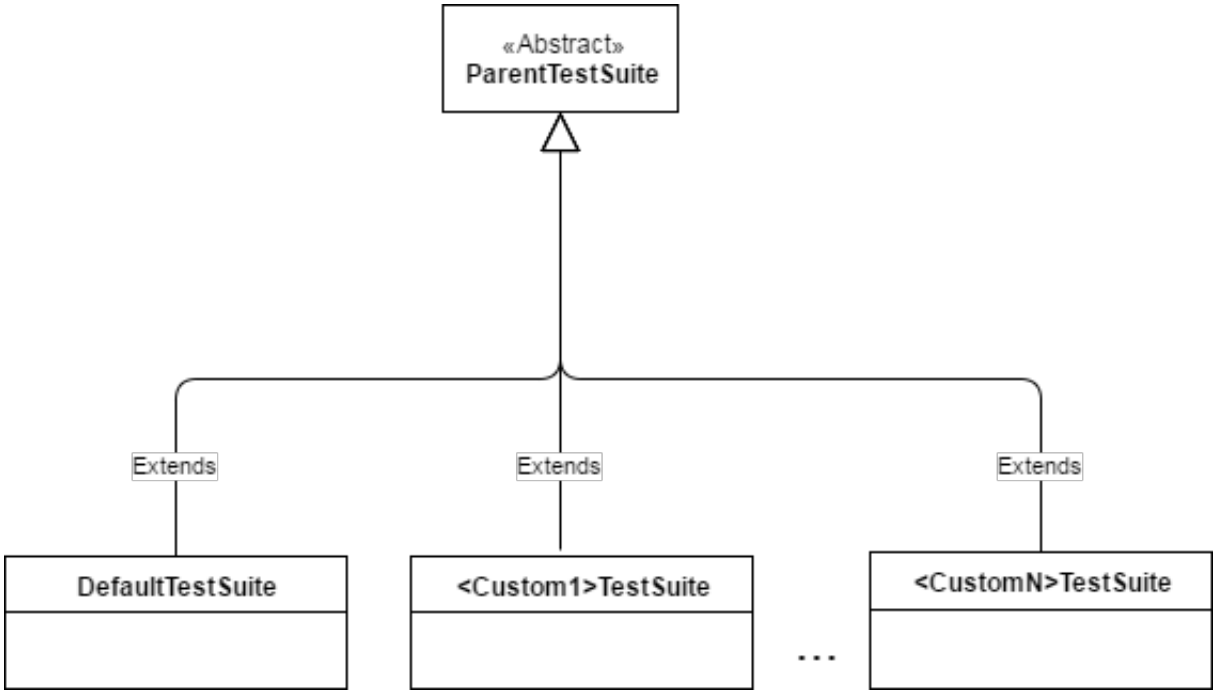

Figure 5.5: Class Diagram of ParentTestSuite

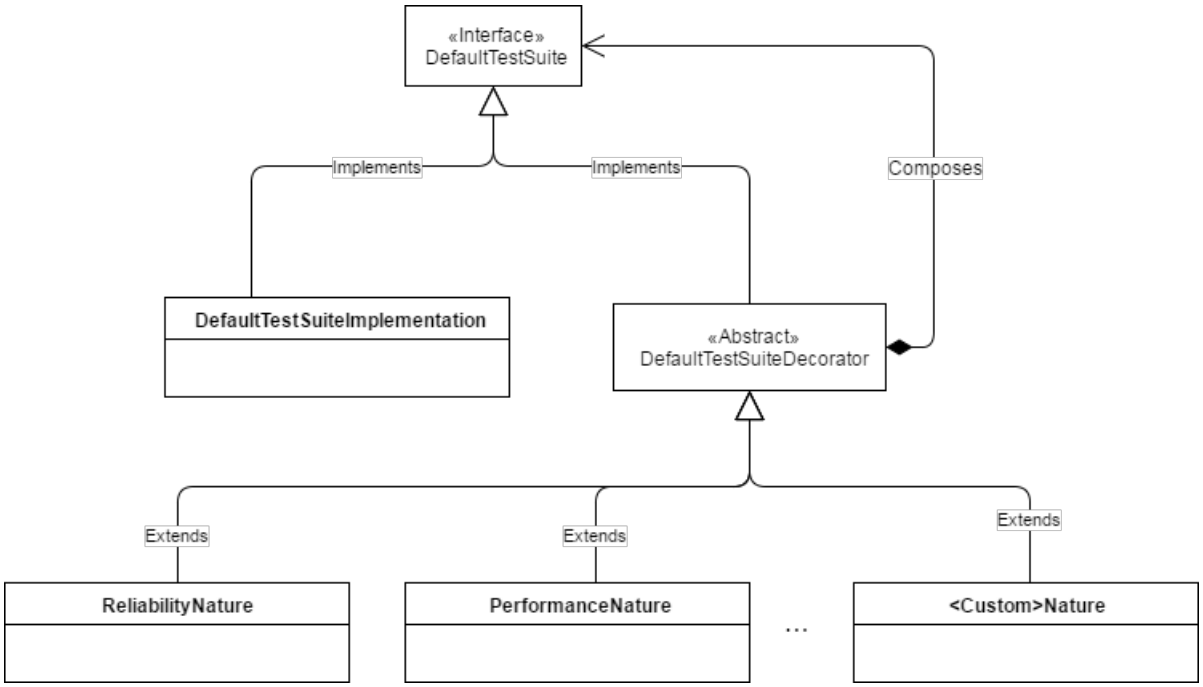

Figure 5.6: Application of the Decorator Pattern on the DefaultTestSuite

However, extending the Parent Test Suite is not always the best solution to add new tester sub-modules. It is likely that new test suites implement additional functionalities and, at the same time, require methods and attributes already implemented and used by other working test suites. Furthermore, among the desired behaviours of test suites that are going to be added, there might be the need of combined operations. For instance, testers may be interested in measuring the Mean Time To Failure of a snippet of code during a load test. The approach followed is the application of the *Decorator Pattern*. The *Decorator Pattern*[GHJV95] permits to add at run-time new features to a test suite instance, wrapping the base class with an abstract Decorator class. New decorations can be added extending the Decorator class. In Figure 5.6, the architecture of the pattern applied to the Default Test Suite is depicted. Casting the instance *DefaultTestSuite* to *ReliabilityNature* the operations concerning the metrics introduced in the Section 4.4.1 are enabled. In addition, new simple expressions are supported. It is now possible to insert NFRs as *"the <metric> of this scenario is <bound> <parameter>"* where <metrics> is one of the reliability metrics, <bound> defines the range in which the value inserted in <parameter> can be considered correct or not. *PerformanceNature* introduces the capability of executing the scenarios in parallel. In the example of Listing 5.2 is shown the operation of adding of a new nature with the method *ReliabilityNature.safelyDecore(ptt)* and consequent activation of the visualisation of the reliability report with the method *enableReliabilityReport()*.

5.4 Result of the execution

Listing 5.4 displays the report generated by TomatoFramework after its execution. The report refers to the code shown in Listings 5.1, 5.2 and 5.3. Additional code like the definition of the first three steps in Listing 5.1 was omitted because not

relevant. In lines 1 - 4 is shown the execution of Cucumber. In the meanwhile, TomatoFramewok stores the step definitons called by Cucumber and displays their name. Line 5 notifies the take-over of TomatoFramework. After, the scenario is executed a custom number of times. In this example the number of iteration was set at 4. From line 6 to line 22 every iteration re-execute the whole scenario applying the *random* rule to the number of clients accessing the intranet Lines 23 - 34 show a report of the execution of every iteration. In this example the scope of the stateFormula is the whole scenario, therefore the execution time of the steps in the scope of the stateFormula is equal to the total execution time. Line 24 shows the value of the time constraint. Only one iteration out of four satisfied the constraint, thus the requirement is not satisfied because it was requested an execution time less than 4 seconds at least 90% of the iterations. From 35 to 39 the reliability report is shown. The values of POFOD, ROCOF and MTTF of the whole scenario are displayed

```
1  TomatoFramework: Observing steps of Scenario Server messages reception
2  2 clients_accessed_the_intranet
3  every_client_sends_a_message_to_the_server
4  the_messages_are_delivered
5  TomatoFramework: Testing Phase =================== Starting execution
6  Iteration nr.1
7  24 clients_accessed_the_intranet
8  every_client_sends_a_message_to_the_server
9  the_messages_are_delivered
10 Iteration nr.2
11 12 clients_accessed_the_intranet
12 every_client_sends_a_message_to_the_server
13 the_messages_are_delivered
14 Iteration nr.3
15 2 clients_accessed_the_intranet
16 every_client_sends_a_message_to_the_server
17 the_messages_are_delivered
18 Iteration nr.4
19 34 clients_accessed_the_intranet
20 every_client_sends_a_message_to_the_server
21 the_messages_are_delivered
22 >><<>><<>><<>><<>><<>><<>><<>><<>><<>><<>><<>>
23 Report ProbabilisticTestingSuite Execution
24 StateFormula verified for the following values where time < of 4.0
25 Iteration nr.0 stateFormulaScope Time: 9.6 seconds, Total Time:9.6 seconds
26 Not Passed
27 Iteration nr.1 stateFormulaScope Time: 4.8 seconds, Total Time:4.8 seconds
28 Not Passed
29 Iteration nr.2 stateFormulaScope Time: 0.799 seconds, Total Time:0.799 seconds
30 Passed
31 Iteration nr.3 stateFormulaScope Time: 13.6 seconds, Total Time:13.6 seconds
32 Not Passed
33 Constraint verified if the previous assertion is true for > 0.9 of iterations
34 Probaility measured:0.25
35 <<Opening Reliability Overview>>
36 Probability Of Failure On Demand(POFOD): 0.75 out of 1
37 Rate Of Occurrence Of Failure(ROCOF): 0.10417028369040592 failures per second
38 Mean Time To Failure(MTTF): 9.599666666666666 seconds
39 <<Closing Reliability Overview>>
40 Quality constraint not satisfied
41 Tomato Framework =================== Ending execution
```

Listing 5.4: Report generated from the execution of the scenario in Listing 5.1

Chapter 6

Results and conclusion

6.1 Evaluation of the tool

After the completion of the proof of concept of TomatoFramework, a meeting was settled with the members of the chapter group "DEV API Back-End". A presentation and a demonstration were carried out. During the meeting, participants could ask questions and give feedback. In addition, an evaluation survey was sent to the participants after the seminar. Seven experts among the participants filled the survey, which is reported in Table 6.1. The survey is divided into two sections. The questions from (e.1) to (e.9) aimed in delineating the background of the respondents while the questions from (e.10) to (e.20) gathered the opinions about TomatoFramework.

The pool of respondents had different amount of years of experience within the company, from less than one year to more than ten years. Questions were asked to identify currently adopted approaches concerning test automation and specification. The degree of automation in testing appears to be low with only 2 respondents relying on it. Developers prefer manual scripts and ad-hoc testing. Successive questions focused on tools used to measure Performance, Availability, Reliability and Security. Performance is mostly measured with Gatling, a tool that allows writing scripts containing scenarios in Scala. Security, Availability and Reliability are measured in large part by ad-hoc manual tests. Moreover, all the respondents had experience in the past with BDD.

Questions (e.10),(e.11) and (e.12) focused in assessing how clear were the objectives and the operating principle of TomatoFramework. Respondents find the objectives pretty clear while only four respondents find that using and extending the framework is straightforward. (e.13), (e.14) and (e.15) requested feedback about the requirement specification phase with TomatoFramework. Only three respondents think that writing requirements based on a predefined template will save time, however four respondents think it is a task that can be done easily. Lastly, five respondent believe that the framework brings value to the phase of specification of requirements.

Concerning test automation, six respondents perceive that the framework brings an added value and five respondents would like to use it. In addition, five of them think that overall the framework permits to save time and resources. However, when it was asked to the participant whether their team would adopt TomatoFramework, six responded "no" or "maybe". An additional question was provided to who re-

sponded negatively, asking what are the reasons that make the framework not suitable for their team. Feedback was gathered at this stage with the purpose of defining the direction that the framework should take to adapt to the business workflow. The feedback and its interpretation in terms of future improvements are listed as follows:

- Not everything is Java Virtual Machine: The framework can be only used in software projects based on Java. In addition, an AspectJ engine-based also on Java is needed to perform property-based testing of the scenarios. Nevertheless, Java is only one of the language used for the production of code. The framework should be expanded to new languages and other research has to be made to enable the potential aspect-oriented capabilities of the most commong languages used in software production.

- The framework appears hard to learn: Several respondents found difficult and time-consuming understanding how to fully manage the framework. Further work has to be done to create tutorial and documentation. Moreover, during the presentation, some developer found the default grammar too formal and cumbersome. New research should be done to identify expressions to be added to the grammar which is part of the everyday communication concerning NFRs.

- A developer stated that for proper resilience test, TomatoFramework is not powerful enough. New metrics and expressions should be added in the grammar to guarantee to provide a complete coverage of methods and techniques used in a specific domain. For example, resilience concerns are often measured using mathematical constructs like percentile, median or standard deviation.

Overall, developers found the framework effective. In all the questions focusing on how effective the framework is perceived, e.g. (e.13), (e.14), (e.15), (e.16), (e.17) and (e.19), positive answers outnumbered negative ones. It is worth noticing that the current version of the framework consists only of a proof of concept and therefore still not suited to be completely adopted.

e.1)	**Question**: What are your roles within ING?
	Answers: All the respondents are developers. In addition, two of them are respectively chapter lead and product owner.
e.2)	**Question**: For how long have you been working in ING?
	Answers: Experience of the respondents was really diversified. One respondent replied "less than one year". Overall, the other participants responded in equal measure "between one and three years","between three and ten years" and "more than ten years".
e.3)	**Question**: How automated is non-functional testing generation/execution in your team?
	Answers: Overall, test automation is little executed. Only 2 respondents declared to rely on automatic scripts running prior to every release. In addition, one of them uses ad-hoc manual tests and unit tests designed for performance testing. The remaining 5 respondents execute manually scripts

e.4)	**Question**: What technologies are used in your team to measure the performance of the software?
	Answers: Six respondents use Gatling and three of them ad-hoc manual testing. One respondent uses Jmeter and ad-hoc manual testing
e.5)	**Question**: What technologies are used in your team to measure the reliability of the software?
	Answers: All the respondents use ad-hoc manual tests. Three of them use other tools like Gatling and Docker and Scalatest
e.6)	**Question**: What technologies do you use to measure the availability of the software?
	Answers: Three respondent do not need to measure availability. The remaining respondents use ad-hoc manual testing and one of them uses also Grafana and Graphite
e.7)	**Question**: Which technologies do you use to measure the security of the software?
	Answers: Six respondents declared to perform ad-hoc manual testing. Two respondents executes statistical code analysis with other tools.
e.8)	**Question**: Which methods or technologies are used to specify non-functional requirements?
	Answers: Six respondents use documentation written in natural language. Two of them share requirements also orally and one of them use Specification By Example. The other respondent does not specify NFRs
e.9)	**Question**: Do you have previous experience with BDD?
	Answers: All the respondents used BDD.
e.10)	**Question**: Did you understand the objectives of TomatoFramework?
	Answers: Overall, the objectvies of TomatoFramework were clear to all the respondents.
e.11)	**Question**: Did you understand how to use and extend TomatoFramework?
	Answers: Three respondents declared to be unsure and four respondents declared it was clear.
e.12)	**Question**: Did you need more time than usual to understand the operating principle of TomatoFramework?
	Answers: One third of the respondents did not need more time then usual while another third did. The remaining one third was unsure about the answer.
e.13)	**Question**: Do you think that you would take too much time to write non-functional requirements based on a predefined grammar?
	Answers: Three respondents believe that writing NFRs with a predefined grammar would not take too much time. Two respondents were unsure and other two think that would take too much time.
e.14)	**Question**: Do you think that it would be hard to write non-functional requirements based on a predefined grammar?
	Answers: Two respondents declared it would be hard and one is unsure. Four respondents think would be easy.

e.15)	**Question**: Do you think that the TomatoFramework provides more value to the phase of specification of non-functional requirements?
	Answers: Two respondents think that TomatoFramework do not bring value to the specification step. Instead, five respondents believe that the framework would bring value.
e.16)	**Question**: Do you think that TomatoFramework provides more value to testing automation of non-functional requirements?
	Answers: Six respondents think that TomatoFramework can bring value to test automation. One respondent provided a negative answer
e.17)	**Question**: Would you like to use TomatoFramework in your work?
	Answers: Five respondents would like to use the framework, while two respondents would not.
e.18)	**Question**: Do you think that your team would adopt TomatoFramework?
	Answers: Three respondents did not provide an answer. Two respondents answered "maybe". One respondent declared "yes" and the other one "no".
e.19)	**Question**: Do you think that TomatoFramework can save time and resources?
	Answers: Five respondents believe that TomatoFramework can save time and reasources. One respondent is unsure and the other one think that TomatoFramework does not help in saving time and resources
e.20)	**Question**: What are the reasons that make TomatoFramework not suitable for your team? (This question was provided only to whom answered "maybe","no" or "I don't know" to the question (e.18))
	Answers: The reasons mentioned by the respondents are: • The framework should work outside JVM. • The effort required to learn the tool seems greater than the value that would provide. • It is not possible to execute proper resilience tests. • Time available to adopt. • TomatoFramework seems to hard to learn and too much maintenance is needed.

Table 6.1: Summary of the answers of the survey

6.2 Discussion

TomatoFramework represents a step forward in the management of NFRs in the production of software. It allows to deal with the complexity of quality requirements enabling the definition of grammars describing the domain of quality concerns. Templates of NFRs are provided to drive developers and stakeholders to write correct and measurable requirements. A template can be associated with automatic operations which can trigger the calculation of metrics, the execution of tests or the generation

of reports. Therefore, TomatoFramework operates in two directions. It provides an expandable knowledge base in the form of grammars created from recurring patterns in non-functional specification and, at the same time, it executes automatic operations to verify the fulfilment of the requirements expressed. Furthermore, functional testing can be modified at run-time with custom ranges of parameters to enable extensive and large scale testing.

The research questions "RQ1: How can we improve the process regarding specification of non-functional requirements?" and "R2: How can we improve the automation of the testing of non-functional requirements?" drove this research. In order to identify open challenges in the state of the art of current processes, two directions were followed: a research on the scientific literature and a series of interviews with experts that cope with NFRs in their daily work. In the specific, challenges C1, C2, C4, C5, C6, C9, C10 and C11 addressed RQ1 while C2, C3, C4, C5, C7 addressed RQ2. The broad scope of the two research questions does not allow to provide a single answer to them. In fact, several factors impact the efficiency of the process of specification and testing of NFRs. The functionalities provided by TomatoFramework represent an attempt in handling these factors. In the following, it is showed how the framework deals with every identified challenge.

- (C1) Need of approaches managing early requirements: One of the primary objectives of the framework consists of enabling specification and testing of NFRs since the early stage of the projects. NFRs can be already specified when the first scenarios are defined. They are expressed according to a grammar and are immediately validated with little need of additional code.

- (C2 and C6) Alignment of specification and testing of NFRs. BDD itself is a revolutionary approach highly focused in a continuous iteration between specification and testing. TomatoFramework provides railroads in the form of templates and grammars to guide specification of quality requirements and their automatic verification.

- (C3) Evaluation and validation of proposals are not fully performed: Evaluation was carried out in the form of a presentation and a survey

- (C4) Evaluation and validation of proposals are often not carried out in industrial setting. The framework was built based on the feedback received on the interview of experts in the IT banking sector. In addition, it was presented to another set of experts and a survey to gather their opinion was provided to them.

- (C5) Need of a formal language. In the framework is defined a default configuration which allows to specify common quality requirement in a correct and measurable way. Furthermore, this language was extended with expressions enabling the use of metrics and techniques to measure performance, availability, reliability, performance and scalability. It is possible to add new expressions and metrics.

- (C7) More automation in the testing of NFRs. Requirements written following the templates provided by the grammars can be interpreted and tested. The file containing the specification is parsed by Cucumber and TomatoFramework and can be inserted in CI and CD processes.

- (C8) More control over test automation of NFRs. TomatoFramework allows a fine-grained control on how the parameters of a scenario are evaluated and which metrics or methods have to be used. In addition, the tester modules can be extended to adapt to the domain of the software.

- (C9) Language barrier and background diversity. As well supporting default constructs, TomatoFramework enables the creation of new grammars. This allows to the domain-expert to define how the language should be used. Other developers can execute black-box testing without the need to be aware of which tests are executed simply writing the requirements in the specification file. For instance, a front-end developer would not need a performance engineering background to realize that the code added is making the whole system performing under the specification.

- (C10) Stakeholders involvement. Despite it is not possible to eliminate completely the complexity of NFRs without impacting their expressiveness, NFRs specified with TomatoFramework can be a step forward in this direction. Requirements are specified in a well-known format that can be supported by wiki and documentation. A stakeholder could execute testing of FRs and NFRs without the need of a deep technical knowledge.

- (C11) Collaboration. TomatoFramework encourages the definition of NFRs in Gherkin feature files. In this way, quality concerns are expressed in an easy but complete language. Developers and stakeholders are more aware of the specification of the software and, at the same time, have immediate knowledge of which NFRs are already fulfilled and which not.

In conclusion, the research questions were answered with the implementation of TomatoFramework.

6.3 Future work

This thesis represents a first step to an evolved management process for specification and testing of NFRs in BDD. Several improvements in TomatoFramework and further research are still needed in order to make the framework ready for the business context. Some of the next improvements to make on the framework are depicted in Section 6.1. In addition, given the limited amount of time, some of the solutions aiming to extend the default grammar discussed in Section 4.4.1 were not implemented. In particular, the grammar related to availability metrics and some of the performance metrics. In the next future, the related grammars and test suites will be provided. Further research is also needed to develop strategies to measure and test security requirements.

It is also suggested to make validate and evaluate the framework by other teams of other domains in order to gather feedback useful to ensure a full interoperability. New methods, metrics and grammars could be added for every domain, while other domains could be extended.

Lastly, to make an overview of the challenges concerning NFRs existing literature reviews were taken into account. However, the last published was in 2014. Furthermore, a new extensive literature study is needed to include most recent findings.

Bibliography

[17] Gherkin. 2017. URL: https://github.com/cucumber/cucumber/wiki/Gherkin (visited on 03/31/2017).

[Agi01] Agile Alliance. Agile manifesto. 2001. URL: http://agilemanifesto.org/ (visited on 03/30/2017).

[Agi16] Agile 101. What is agile? 2016. URL: https://www.agilealliance.org/agile101/ (visited on 03/31/2017).

[BE11] Zeinab Alizadeh Barmi and Amir Hossein Ebrahimi. Automated testing of non-functional requirements based on behavioural scripts. *Chalmers university of technology, msc thesis, goteborg*, 2011. URL: http://publications.lib.chalmers.se/records/fulltext/155661.pdf (visited on 10/10/2016).

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BV17] iDEAL BV. Welcome to ideal. 2017. URL: https://martinfowler.com/bliki/SpecificationByExample.html (visited on 03/31/2017).

[Fow04] Martin Fowler. Specificationbyexample. 2004. URL: https://martinfowler.com/bliki/SpecificationByExample.html (visited on 03/31/2017).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN: 0-201-63361-2.

[Gli07] M. Glinz. On non-functional requirements. In *15th IEEE international requirements engineering conference (RE 2007)*. 15th IEEE International Requirements Engineering Conference (RE 2007), October 2007, pages 21–26. DOI: 10.1109/RE.2007.45.

[Gru08] Lars Grunske. Specification Patterns for Probabilistic Quality Properties. In *Proceedings of the 30th International Conference on Software Engineering*. In ICSE '08. ACM, New York, NY, USA, 2008, pages 31–40. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368094. URL: http://doi.acm.org/10.1145/1368088.1368094 (visited on 03/12/2017).

[HA05] S. E. Hove and B. Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE international software metrics symposium (METRICS'05)*. 11th IEEE International Software Metrics Symposium (METRICS'05), September 2005, 10 pp.–23. DOI: 10.1109/METRICS.2005.24.

[Hig84] James Higginbotham. English Is Not a Context-Free Language. *Linguistic inquiry*, 15(2):225–234, 1984. ISSN: 0024-3892. URL: http://www.jstor.org/stable/4178381 (visited on 12/22/2016).

[HLN14] M. Mahmudul Hasan, Pericles Loucopoulos, and Mara Nikolaidou. Classification and qualitative analysis of non-functional requirements approaches. In Ilia Bider, Khaled Gaaloul, John Krogstie, Selmin Nurcan, Henderik A. Proper, Rainer Schmidt, and Pnina Soffer, editors, *Enterprise, business-process and information systems modeling*, number 175 in Lecture Notes in Business Information Processing, pages 348–362. Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-43744-5 978-3-662-43745-2. URL: http://link.springer.com/chapter/10.1007/978-3-662-43745-2_24 (visited on 10/06/2016). DOI: 10.1007/978-3-662-43745-2_24.

[KB14] Gurpreet Kaur and Kailash Bahl. Software reliability, metrics, reliability improvement using agile process. *International journal of innovative science, engineering and technology*, 1(3):143–7, 2014. URL: http://ijiset.com/v1s3/IJISET_V1_I3_24.pdf (visited on 12/13/2016).

[Lio+96] JL Lions et al. Ariane 5 failure-full report. *Paris: esa*, 1996.

[LN04] Phillip A Laplante and Colin J Neill. The demise of the waterfall model is imminent. *Queue*, 1(10):10, 2004.

[LSZH13] Pericles Loucopoulos, Jie Sun, Liping Zhao, and Farideh Heidari. A systematic classification and analysis of NFRs. *AMCIS 2013 proceedings*, May 30, 2013. URL: http://aisel.aisnet.org/amcis2013/SystemsAnalysis/RoundTablePresentations/1.

[Mar11] Nick Martens. The impact of non-functional requirements on project success. *Utrecht university, msc thesis, utrecht*, 2011. URL: https://pdfs.semanticscholar.org/3137/edff8cca93165c5e760183d4c91283ab9205.pdf (visited on 02/23/2017).

[MLMPT13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: a survey. *Ieee transactions on software engineering*, 39(6):869–891, 2013.

[Nor06] Dan North. Introducing bdd. 2006. URL: https://dannorth.net/introducing-bdd// (visited on 03/31/2017).

[PK04] Barbara Paech and Daniel Kerkow. Non-functional requirements engineering-quality is essential. In *10th international workshop on requirments engineering foundation for software quality*, 2004.

[Pra09] Dhanji R Prasanna. *Dependency injection*. Manning Publications Co., 2009.

[Roy+70] Winston W Royce et al. Managing the development of large software systems. In *Proceedings of ieee wescon*. Volume 26. (8). Los Angeles, 1970, pages 1–9.

[UIK11] S. Ullah, M. Iqbal, and A. M. Khan. A survey on issues in non-functional requirements elicitation. In *International Conference on Computer Networks and Information Technology*, July 2011, pages 333–340. DOI: 10.1109/ICCNIT.2011.6020890.

[VA16] Bill Venners and Artima. Property-based testing. 2016. URL: http://www.scalatest.org/user_guide/property_based_testing (visited on 03/31/2017).

[Van04] Maarten Van Steen. Distributed systems principles and paradigms. *Network*, 2:28, 2004.

[WH12] Matt Wynne and Aslak Hellesoy. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.