Extending BDD
A systematic approach to handling non-functional requirements

Pedro Moreira

Kellogg College
University of Oxford

A dissertation submitted for the
MSc in Software Engineering

Abstract

Software engineering methods have evolved from having a prescribed and sequential nature to using more adaptable and iterative approaches. Such is the case with Behaviour Driven Development (BDD), a recent member of the family of agile methodologies addressing the correct specification of the behaviour characteristics of a system, by focusing on close collaboration and identification of examples.

Whilst BDD is very successful in ensuring developed software meets its functional requirements, it is largely silent regarding the systematic treatment of its non-functional counterparts, descriptions of how the system should behave with respect to some quality attribute such as performance, reusability, etc.

Historically, the systematic treatment of non-functional requirements (NFRs) in software engineering is categorised as being either product-oriented and based on a quantitative approach aimed at evaluating the degree to which a system meets its NFRs or process-oriented, qualitative in nature and where they are used to drive the software design process. Examples of the latter category, are the NFR Framework – a structured approach to represent and reason about non-functional requirements – and the goal-oriented requirements language (GRL) that provides support for evaluation and analysis of the most appropriate trade-offs among (often conflicting) goals of stakeholders.

In this thesis, we investigate the extent to which goal-oriented principles can be integrated in BDD, with the aim of handling non-functional requirements in an explicit and systematic way, whilst respecting the principles and philosophy behind agile development.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr Jeremy Gibbons, for his guidance, support, comments and encouragement.

I would also like to thank my family for their constant support and love, and in particular my wife, Tamara Moreira, for her endless patience whenever I so often disappeared to my office to work on this thesis.

The author confirms that: this dissertation does not contain material previously submitted for another degree or academic award; and the work presented here is the author's own, except where otherwise stated.

Contents

List of Figures

List of Tables

List of Features

1 Introduction

This thesis presents an extension to Behaviour Driven Development (BDD) (North, 2006; Smart, 2014) to support the elicitation, communication, modelling and analysis of non-functional requirements. It includes concepts and techniques from goal-oriented requirements engineering (GORE) (Van Lamsweerde, 2001), and more specifically, allows the definition of goals in BDD and modelling and analysis in Goal Requirements Language (GRL) (Amyot et al., 2010). This is achieved by integrating notions of goals in Gherkin (Wynne and Hellesoy, 2012) – a domain specific language for the representation and specification of requirements. We also present a translator from Gherkin to GRL, allowing Gherkin-defined actors and goals satisfactions levels to be subject to qualitative and quantitative analysis in a GRL tool.

1.1 Motivation

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended (Nuseibeh and Easterbrook, 2000). Shortcomings in the ways that people learn about, document, agree upon and modify such statements of intent are known causes to many of the problems in software development (Wiegers and Beatty, 2013). We informally refer to these statements of intent as Requirements and the engineering process to elicit, document, verify, validate and manage them as Requirements Engineering [1].

The importance of requirements in software engineering cannot be understated. In his essay *No Silver Bullet*, Brooks (1987), referring to the critical role of requirements to a software project, states that

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later. (Brooks, 1987)

More recently, Davis (2013) reveals that errors introduced during requirements activities account for 40 to 50 percent of all defects found in a software product. When arguing for the importance of requirements, Hull et al. (2011) reason that to be well understood by everybody they are generally expressed in natural language and herein lies the challenge: to capture the need or problem completely and unambiguously without resorting to specialist jargon or conventions. The authors follow by positing these needs may not be clearly defined at the start, may conflict or be constrained by factors outside their control or may be influenced by other goals which themselves change in the course of time.

Furthermore, requirements can be classified in multiple and at times conflicting ways. Glinz (2007) points out that in every current classification scheme there is a distinction between re-

[1] These topics will be explored in depth in Chapter 2

quirements concerning the functionality of a system and all other, often referred to as non-functional requirements. In another paper, the same author points out issues with current classification schemes such as sub-classification, terminology and satisfaction level whereby some requirements are considered *'soft'* in the sense that they can be weakly or strongly satisfied (e.g *the system shall have a good performance*; or *the System shall be secure*). Chung and Leite (2009) contribute that, in spite of this separation, most existing requirement models and requirements specification languages lack a proper treatment of non-functional requirements. In addition, this separation of functional and non-functional requirements has lead to the latter being either neglected, addressed later in a project or completely ignored. This problem applies to both traditional and agile software development processes.

A software process is generically defined as a set of activities, methods, practices, and transformations that are used to develop and maintain software and its associated products (Cugola and Ghezzi, 1998). Agile software development approaches have become more popular during the last few years. Several methods have been developed with the aim of delivering software faster and to ensure that the software meets customer changing needs. All these approaches share some common principles: Improved customer satisfaction, adopting requirements, frequently to changing delivering working software, and close collaboration of business people and developers (Paetsch et al., 2003).

One of such agile approaches is Behaviour Driven Development (BDD). The understanding of BDD is far from clear and unanimous (Solis and Wang, 2011). Some authors refer to BDD as a development process (Smart, 2014), others state that it is not a fully fledged software development methodology but rather *'supplement other methodologies, provide rigour in specifications and testing, enhance communication between various stakeholders and members of software development teams, reduce unnecessary rework, and facilitate change.' (Adzic, 2011)*

In spite of the above mentioned differences of interpretation, it is unanimously accepted that BDD focus on deriving from business goals, a sufficiently set of software features that contribute to achieve these business goals. This process makes use of Gherkin (Wynne and Hellesoy, 2012) – a domain specific language which promotes the use of a ubiquitous language [2] that business people can understand – to describe and model a system. However the focus has been on functionality and quality characteristics such as performance, security, maintainability are not explicitly addressed. To the best of our knowledge, the single exception to the above, is the work of Barmi and Ebrahimi (2011), but with restricted applicability to probabilistic – those that can be written using probabilistic statements (Grunske, 2008) – non-functional requirements only.

None of these agile practices, treat non-functional requirements in a systematic way, certainly not in a way that allows reasoning about which requirements interdependencies may exist, and the positive or negative influences each may have on each other. Among many proposals, goal-oriented approaches were the first to treat non-functional requirements as first-class citizens. Mylopoulos et al. (1999) observed that goal-oriented requirements engineering is generally complementary to other approaches and, in particular, is well suited to analysing requirements early in the software development cycle, especially with respect to non-functional requirements and the evaluation of alternatives.

It seems only logical and expectable that, improvements to the discovery and communication of requirements, and in particular non-functional requirements, will lead to an increase in success rates of software projects.

[2] Eric Evans first introduced that term in *Domain-driven Design: Tackling Complexity in the Heart of Software* Evans (2004)

1.2 Aim and limitations of study

The context described in the previous section justify research aimed at capturing, documenting and communicating requirements using natural language tools and techniques in a precise, complete and unambiguous way, but also with the flexibility and adaptability to allow requirements to change and evolve through the course of time.

In this thesis, we investigate the extent to which goal-oriented principles can be integrated in BDD, with the aim of handling non-functional requirements in an explicit and systematic way, whilst respecting the principles and philosophy behind agile development. In particular, we consider how BDD can be extended, and also Gherkin modified, to incorporate actor and goal concepts as defined and treated in GRL.

We do not however investigate the integration of GRL with use case maps (UCM), as part of the User Requirements Notation (Liu and Yu, 2004). UCM targets modelling scenarios of functional or operational requirements and performance and architectural reasoning. This is left as an area for further research.

We also do not aim at providing another classification scheme and address the, sometimes artificial, separation of functional and non-functional requirements. Instead, we adopt the notion of goals as an objective the system under consideration should achieve and goals formulations as properties to be ensured. We share the view that goals cover different types of concerns: functional which are associated with the services to be provided, and non-functional concerns associated with quality of service such as safety, security, accuracy, performance, and so forth (Van Lamsweerde, 2001).

Finally, we do not apply this technique to a specific non-functional requirement or use any particular taxonomy as our approach is independent of the NFR being addressed or taxonomy chosen.

1.3 Significance of the study

By reinterpreting behaviours in BDD as not just specifications of functionality of a system but as statements of goals, this thesis brings the following contributions to BDD:

- Allows non-functional requirements to be specified in natural language form in Gherkin
- Allows Gherkin specifications to be converted into goal models and imported and used in jUCMNav
- Allows BDD to consider all non-functional requirements, not just those that are technical, but still relevant for a successful product delivery
- Brings to BDD the capability to assess qualitative and quantitative satisfaction levels of actors and goals

By allowing goals to be elicited and specified in Gherkin, this thesis brings the following contributions to goal-oriented requirements engineering:

- Allows goals elicitation to occur in Gherkin using natural language and therefore more suitable for discussion and fostering communication
- Brings the benefits of executable specifications in BDD to goal formulations

1.4 Overview of contents

We have now reviewed the motivation for this study, stated the aim of the research and identified the contributions our work brings to BDD and goal-oriented requirements engineering and the research community in general. The rest of the thesis is organised as follows:

Chapter 2 contains all the necessary background material related to requirements engineering, the approaches taken by agile processes and, in particular section 2.3.3 on behaviour-driven development, describing the principles and practices of this popular agile process. Chapter 3 presents an overview of the research concerning ways of handling non-functional requirements in software engineering and section 3.3.1 on goal-oriented requirements engineering with a focus on GRL and with a description of jUCMNav (Amyot et al., 2010), an editor for GRL models.

Chapter 4 is the core of the thesis and contains details of extensions to Gherkin; mapping of Gherkin elements to GRL, such as actors and intentional elements and a description of a translator from Gherkin to an XML-based interchange format to be used in jUCMNav.

Chapter 5 contains implications of findings, concluding thoughts, identifies limitations of study and suggests topics for future research.

2 From traditional to agile requirements engineering

In Chapter 1 we have outlined and situated our study around insufficiencies in current approaches to handling non-functional requirements in agile development methods, and behaviour-driven development in particular.

In this chapter, we reflect on how fast-changing technology and increased competition are placing an ever increasing pressure on the development process. We first review the notions of requirements and requirements engineering, highlighting the most used processes and activities, regardless of the software development method in use. We follow with a description of requirements engineering practices in agile methods and finish with a presentation of key concepts of behaviour-driven development (BDD), contextualising BDD as an instance of *Specification by Example* (Adzic, 2011).

2.1 Requirements

Despite decades of industry experience, many software organizations struggle to understand, document, and manage their product requirements. Inadequate user input, incomplete requirements, changing requirements, and misunderstood business objectives are major reasons why so many information technology projects are less than fully successful. Some software teams lack the proficiency of eliciting requirements from customers and other sources. Customers often don't have the time or patience to participate in requirements activities (Wiegers and Beatty, 2013).

Effective requirements engineering is crucial to delivering products and services aligned to the goals and objectives for which they were initially conceived. Hull et al. (2011) state that software is the most powerful force behind changes of new products and is mostly driven by three factors: *arbitrary complexity*, due to most products having software at its core and being often complex; *instant distribution* – new products or changes to existing products can be distributed to its clients in a matter of seconds or minutes, usually the time it takes to download, install and configure a new software version – and *off-the-shelf components*, as most systems can now be built from ready-made components, greatly reducing the product development cycle.

2.1.1 Definition

Many problems in the software world arise from shortcomings in the ways that people learn about, document, agree upon and modify product's requirements. Common problem areas are informal information gathering, implied functionality, miscommunicated assumptions, poorly specified requirements, and a casual change process (Wiegers and Beatty, 2013). Various studies suggest that errors introduced during requirements activities account for 40 to 50 percent of all defects found in a software product (Davis, 2013). Inadequate user input and shortcomings in specifying and managing customer requirements are major contributors to unsuccessful

projects. Despite this evidence, many organizations still practice ineffective requirements methods. There is no definitive definition of requirements that satisfies all purposes and concerns, but the ones we provide next, are some of the more consensual ones (Wiegers and Beatty, 2013).

The difficulty with defining requirements, arises mostly due to a terminology problem. Different observers might describe a single statement as being a user requirement, software requirement, business requirement, functional requirement, system requirement, product requirement, project requirement, user story, feature, or constraint (Wiegers and Beatty, 2013). Because of the inter-connectedness of requirements with other aspects of systems engineering and project management, it is quite challenging to find a satisfactory scope for a definition of requirements engineering (Hull et al., 2011). A typical definition of requirement can be found in ISO/IEC/IEEE 29148:2011

> A statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines)

It is worth breaking down this definition into its constituents words. A requirement comes mostly in a textual representation (*statement*) even though there are other complementary or alternative forms such visual forms, formal methods and domain specific languages. Requirements may define what is to be built in response to requirements (*product requirements*) but also procedures for using what will be built (*process requirements*). In addition, there may be requirements that stipulate how the product should be developed, usually for quality control purposes. The definition also allures for the existence of many different kinds of requirements, such as *operational, functional, or design characteristic or constraint*, giving rise to different kinds of language, analysis, modelling, process and solution. It states that a requirement should lend itself to a clear, single understanding, common to all parties involved (*unambiguous*). It should also be quantifiable, thus providing a means of measuring and testing the solution against it. Finally, requirements play a multi-dimensional role and come from a multitude of sources. Sommerville and Sawyer (1997) shares a simpler, but nevertheless, useful definition

> Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

This definition makes clear that different types of information are part of requirements domain. Requirements mean different things for different people: for users, they represent external characteristics of the system behaviour, whilst for developers, they are instead linked with internal characteristics. They include both the behaviour of the system under specific conditions and those properties that make it suitable – and maybe even enjoyable – for use by its intended users (Wiegers and Beatty, 2013).

2.1.2 Classification

Wiegers and Beatty (2013) provide a breakdown of different types of information that may be categorised as requirements. Given that the term 'requirement' is extremely overloaded in software engineering, it is useful to give definitions of these information types, and contextualise their use and relevance (see table 2.1).

Business requirements describe why the organization is implementing the system and the business benefits the organization hopes to achieve. The focus is on the business objectives

Table 2.1: Types of requirements information (Wiegers and Beatty, 2013)

Term	Definition
Business requirement	A high-level business objective of the organization that builds a product or of a customer who procures it
Business rule	A policy, guideline, standard, or regulation that defines or constrains some aspect of the business. Not a software requirement in itself, but the origin of several types of software requirements
Constraint	A restriction that is imposed on the choices available to the developer for the design and construction of a product
External interface requirement	A description of a connection between a software system and a user, another software system, or a hardware device
Feature	One or more logically related system capabilities that provide value to a user and are described by a set of functional requirements
Functional requirement	A description of a behaviour that a system will exhibit under specific conditions
Non-functional requirement	A description of a property or characteristic that a system must exhibit or a constraint that it must respect
Quality attribute	A kind of nonfunctional requirement that describes a service or performance characteristic of a product
System requirement	A top-level requirement for a product that contains multiple subsystems, which could be all software or software and hardware
User requirement	A goal or task that specific classes of users must be able to perform with a system, or a desired product attribute

of the organization or the customer who requests the system. Business requirements typically come from the funding sponsor for a project, the acquiring customer, the manager of the actual users, the marketing department, or a product visionary. Business requirements are usually contained within a vision and scope document. Other strategic guiding documents sometimes used for this purpose include a project charter, business case, and market (or marketing) requirements document (Wiegers and Beatty, 2013).

User requirements describe goals or tasks the users must be able to perform with the product that will provide value to someone. The domain of user requirements also includes descriptions of product attributes or characteristics that are important to user satisfaction. Ways to represent user requirements include use cases (Cockburn, 2000), user stories (Cohn, 2004), and event-response tables. Ideally, actual user representatives will provide this information. User requirements describe what the user will be able to do with the system. Some people use the broader term ”stakeholder requirements” to acknowledge the reality that various stakeholders other than direct users will provide requirements. A good set of stakeholder requirements can provide a concise non-technical description of what is being developed at a level that is accessible to senior management.

Functional requirements specify the behaviours the product will exhibit under specific conditions. They describe what the developers must implement to enable users to accomplish their tasks (user requirements), thereby satisfying the business requirements. These are usually documented in a software requirements specification (SRS), which describes as fully as necessary the expected behaviour of the software system. The SRS is used in development, testing,

quality assurance, project management, and related project functions. People call this deliverable by many different names, including business requirements document, functional spec, requirements document, and others.

System requirements describe the requirements for a product that is composed of multiple components or subsystems. A 'system' in this sense, is not just any information system. A system can be all software or it can include both, software and hardware subsystems. People and processes are part of a system too, so certain system functions might be allocated to human beings. The system requirements can form an excellent technical summary of a development project (Hull et al., 2011).

Business rules include corporate policies, government regulations, industry standards, and computational algorithms. Business rules are not themselves software requirements because they have an existence beyond the boundaries of any specific software application. However, they often dictate that the system must contain functionality to comply with the pertinent rules. Sometimes, as with corporate security policies, business rules are the origin of specific quality attributes that are then implemented in functionality. Therefore, you can trace the genesis of certain functional requirements back to a particular business rule.

In addition to functional requirements, the SRS contains an assortment of *non-functional requirements*. *Quality attributes* are also known as quality factors, quality of service requirements, constraints, and the "-ilities". They describe the product's characteristics in various dimensions that are important either to users or to developers and maintainers, such as performance, safety, availability, and portability. Other classes of non-functional requirements describe *external interfaces* between the system and the outside world. These include connections to other software systems, hardware components, and users, as well as communication interfaces. Design and implementation *constraints* impose restrictions on the options available to the developer during construction of the product.

Figure 2.1 depicts the relationships among these types of requirements information, using ovals for requirements and rectangles for documents. In the figure, solid arrows mean 'are stored in'; dotted arrows mean 'are the origin of' or 'influence.'

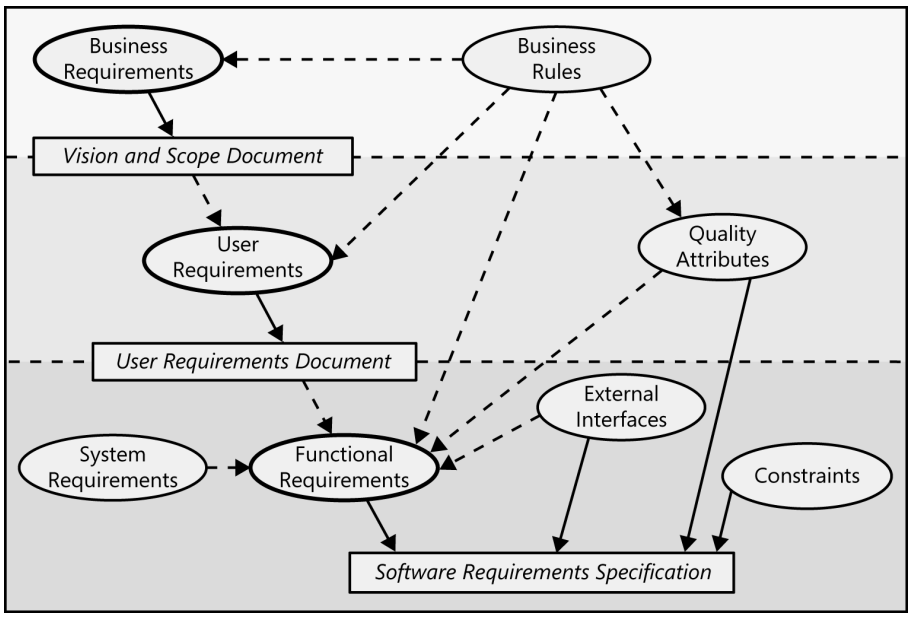

Figure 2.1: Relationships among several types of requirements information (Wiegers and Beatty, 2013, p. 8)

A *feature* consists of one or more logically related system capabilities that provide value to a user, and are described by a set of functional requirements. A feature can encompass multiple user requirements, each of which implies that certain functional requirements must be implemented to allow the user to perform the task described by each user requirement.

We have identified three major requirements deliverables: a vision and scope document, a user requirements document, and a software requirements specification. There is often no need to create three discrete requirements deliverables on each project. It often makes sense to combine some of this information, particularly on small projects. However, we should recognize that these three deliverables contain different information, are developed at different points in the project, possibly by different people and with different purposes and target audiences (Wiegers and Beatty, 2013).

Requirements can also be categorised as either *product* or *project* requirements. Product requirements are those that describe properties of a software system to be built. Projects certainly do have other expectations and deliverables that are not a part of the software the team implements, but that are necessary to the successful completion of the project as a whole. These are project requirements but not product requirements. An SRS houses the product requirements, but it should not include design or implementation details (other than known constraints), project or test plans, or similar information.

Hull et al. (2011) make an important distinction between requirements being defined in either a *problem* or *solution* domain. In the context of requirements existing at different layers of abstraction, those at higher layers, representative of statements of need, usage modelling and stakeholder requirements, pertain to the problem domain, whereas those in lower layers, starting with system requirements, operate in the solution domain. The use of multiple levels of abstraction promotes separation of concerns and allows views of stakeholders, analysts and developers to be taken in consideration. Stakeholder requirements should specify only what they want to achieve and avoid any reference to particular solutions. System requirements, on the other hand, should abstractly specify what the system will do to meet the stakeholder requirements, whilst avoiding any references to any particular design. Finally, subsystem and component requirements, part of architectural designs, will specify how this design meets the system requirements.

2.2 Requirements Engineering

The *Guide to the Software Engineering Body of Knowledge* (*SWEBOK V3.0*, 2014) identifies topics that pertain to software requirements knowledge, which concern the elicitation, analysis, specification, and validation of software requirements as well as their management during the whole life cycle of a software product.

This section defines requirements engineering and breaks it down into its core processes and activities. We will not cover all topics contained in Figure 2.2, but instead focus on the more relevant ones, from the point of view of the work described in this thesis.

The engineering aspect of requirements development and management, should not distract us from the fact that software development involves at least as much communication as it does computing, and yet we sometimes fail to appreciate that requirements engineering and, in particular, requirements elicitation – and much of software and systems project work in general – is primarily a human interaction challenge (Wiegers and Beatty, 2013).

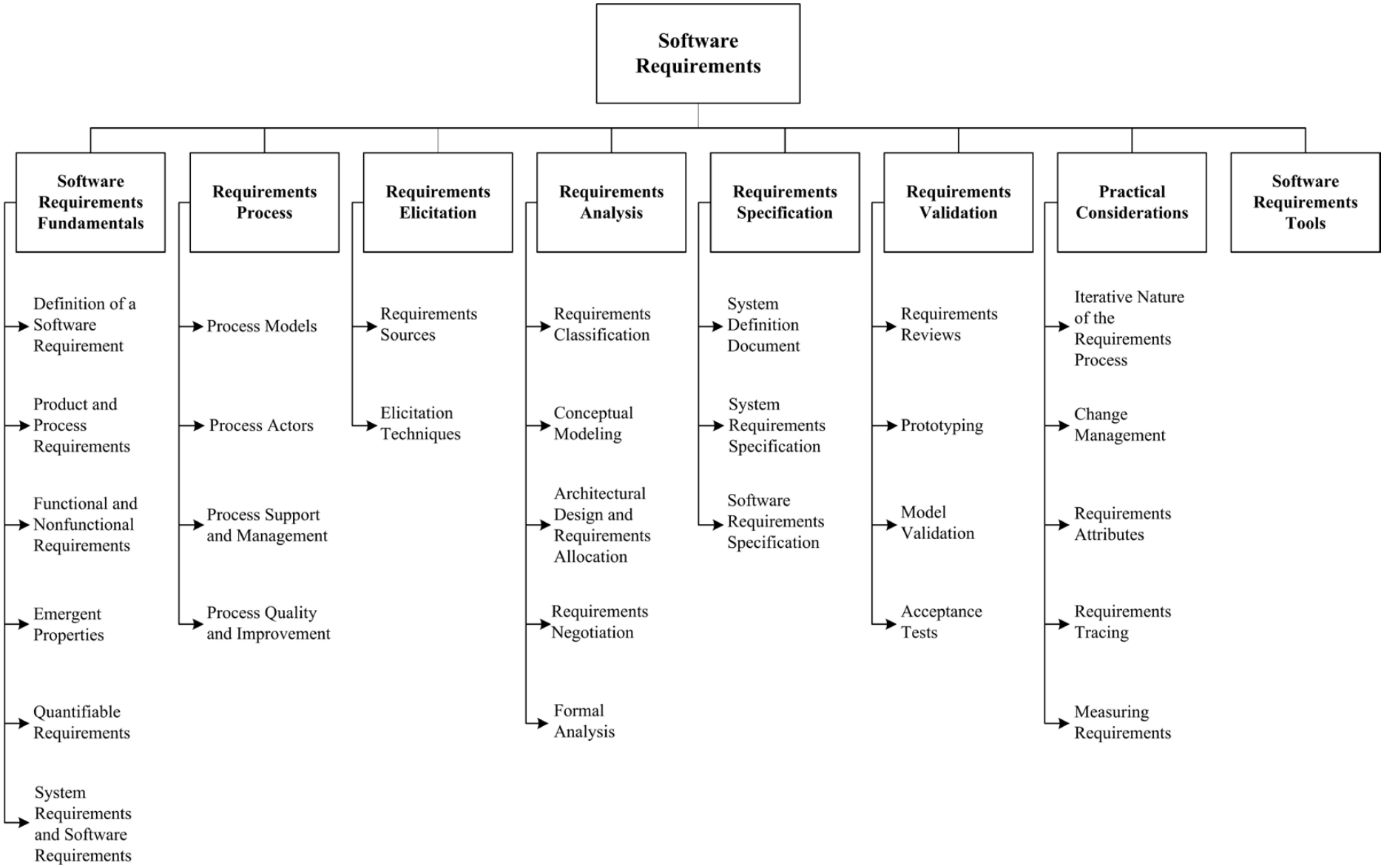

Figure 2.2: Topics for the Software Requirements knowledge area (*SWEBOK V3.0*, 2014)

2.2.1 Definition

The definition in ISO/IEC/IEEE 29148:2011 describes requirements engineering as an *'interdisciplinary function that mediates between the domains of the acquirer and supplier to establish and maintain the requirements to be met by the system, software or service of interest'*. A vital part of the systems engineering process, requirements engineering first defines the problem scope and then links all subsequent development information to it (Hull et al., 2011). One of the most long-standing definition comes from a US Department of Defence software strategy document

> Requirements engineering involves all life-cycle activities devoted to identification of user requirements, analysis of the requirements to derive additional requirements, documentation of the requirements as a specification, and validation of the documented requirements against user needs, as well as processes that support these activities (Department of Defense (DoD), 1991)

A more recent definition emphasizes the goal-oriented nature of requirements engineering, and hints at the importance of understanding and documenting the relationships between requirements and other development artefacts

> Requirements engineering is the branch of software engineering concerned with the real world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families (Zave, 1997)

Hull et al. (2011) argues that both definitions omit the role that requirements play in accepting and verifying the solution. The authors propose an alternative definition

> Requirements engineering is the subset of systems engineering concerned with the discovery, development, trace, analysis, qualification, communication and management of requirements that define the system at successive levels of abstraction (Hull et al., 2011)

They argue the above definition is a better reflection that requirements exist at multiple levels of development, and also list key activities that are considered proper to requirements engineering. Similarly to what we have done for the definition of requirement, it is worth breaking this definition into its constituent parts. *Discovery*, refers to activities related to the elicitation and capture of requirements; *trace* allows setting up links to and from requirements to other artefacts; *qualification* refers to all kinds of testing activities and avoids the often confusing terms *validation* – checking formal expressions of requirements against informal needs – and *verification*, often linked with checks of requirements internal consistency within and between layers of abstraction; *communication* reflects the notion that requirements are part of a human activity, through which all stakeholders agree on what is to be achieved. Finally, the word *abstraction* makes reference to the practice of organizing requirements into layers and of tracing the satisfaction relationship between those layers.

Hull et al. (2011) makes a useful extension to software requirements that applies to complete systems – a collection of components, machine, software and human, which co-operate in an organised way to achieve some desired result. Since components must co-operate, interfaces between components are a vital consideration in system (and requirements) engineering, that is, interfaces between people and machine components, between machine components, and between software components.

2.2.2 Processes

Without loss of generality, we can say that requirements engineering can be split into two main processes, *requirements development* and *requirements management*. Requirements development can be subdivided into elicitation, analysis, specification, and validation (*SWEBOK V3.0*, 2014). Figure 2.3 below shows the domain of requirements engineering split into requirements development, encompassing the activities just mentioned, and also requirements management

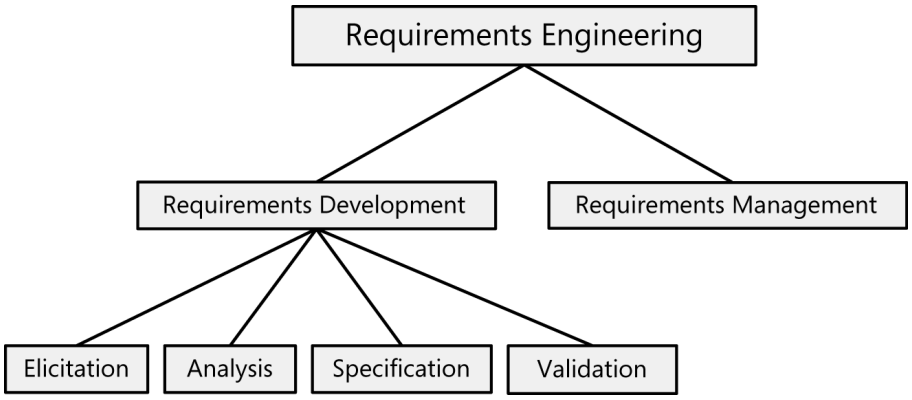

Figure 2.3: Requirements engineering disciplines (Wiegers and Beatty, 2013, p. 15)

Regardless of the development life cycle followed – be it pure waterfall, iterative, incremental, agile, or other – these activities will be present, perhaps at different times in the project and to varying degrees of detail (Wiegers and Beatty, 2013). Following, are the essential actions in each sub-discipline

Requirements elicitation encompasses all of the activities involved with discovering requirements, such as interviews, workshops, document analysis and others. It is a process through which, those who acquire and those who supply a given system, discover, review, articulate, understand, and document the requirements on the system and the life cycle processes (ISO/IEC/IEEE 29148:2011). It typically assumes either a usage-centric or a product-centric approach, although other strategies are also possible. The usage-centric strategy emphasizes understanding and exploring user goals to derive the necessary system functionality. The product-centric approach focuses on defining features that you expect will lead to marketplace or business success (Wiegers and Beatty, 2013).

The same standard defines *requirements analysis* as a process that transforms stakeholder and requirement-driven views of desired services into technical views of products that could deliver those services. The main goal is to obtain a precise understanding of each requirement and representing sets of requirements in appropriate ways. This is done by distinguishing user's goals from functional requirements, determining quality expectations, business rules, suggested solutions, and other information (Wiegers and Beatty, 2013).

Requirements specification involves representing and storing the collected knowledge and information in a persistent and well-organized fashion. The principal activity is translating the collected user needs into written requirements and, optionally visual models, suitable for comprehension, review and use by their intended audiences (Wiegers and Beatty, 2013).

ISO/IEC/IEEE 29148:2011 defines *requirements validation* as a confirmation by examination that requirements (individually and as a set) define the right system as intended by the stakeholders. It confirms that information gathered will enable developers to build a solution that satisfies the business objectives. The central activities are reviewing the documented re-

quirements to correct any problems before the development group accepts them; developing acceptance tests and criteria to confirm that a product based on the requirements would meet customer needs and achieve the business objectives (Wiegers and Beatty, 2013).

Wiegers and Beatty (2013) allude that, from a practical point of view, the goal of *requirements development* is to accumulate a shared understanding of requirements that is good enough to allow construction of the next portion of the product, be that 1 or 100 percent of the entire product, to proceed at an acceptable level of risk. It is in line with what ISO/IEC/IEEE 29148:2011 refers to as a *baseline* set of requirements, that is, *'a specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures'*. The major risk is that, if not performed properly, having to do excessive unplanned rework because of insufficient understanding of the requirements for the next chunk of work before starting design and construction (Wiegers and Beatty, 2013).

ISO/IEC/IEEE 29148:2011 identifies processes and activities within, that are named differently, but that are in essence similar to the ones just described. The principal processes identified are *stakeholder requirements definition* and *requirements analysis* or *system requirements analysis*. These two processes result in a baseline set of requirements, with a nature similar to the mentioned before. The architectural design process includes allocation and decomposition of requirements that triggers the recursive application of the requirements processes, for the definition of system element requirements and the iterative application of the requirements analysis process for derived requirements.

There is a common misconception that requirements engineering is just a single phase that is carried out and completed at the outset of product development (Hull et al., 2011). On the contrary, requirements developed at the outset are still in use at the final stages of development. Figure 2.4 shows different testing activities and their relationship with requirements specified at various levels of abstraction. It is clear that stakeholder requirements are tested as part of acceptance test, a late stage in any software development method, be it traditional or agile. In addition, each testing activity has a separate concern, with acceptance test focusing on validating the product; system test verifying the system and integration and component testing, verifying subsystem and components requirements, respectively.

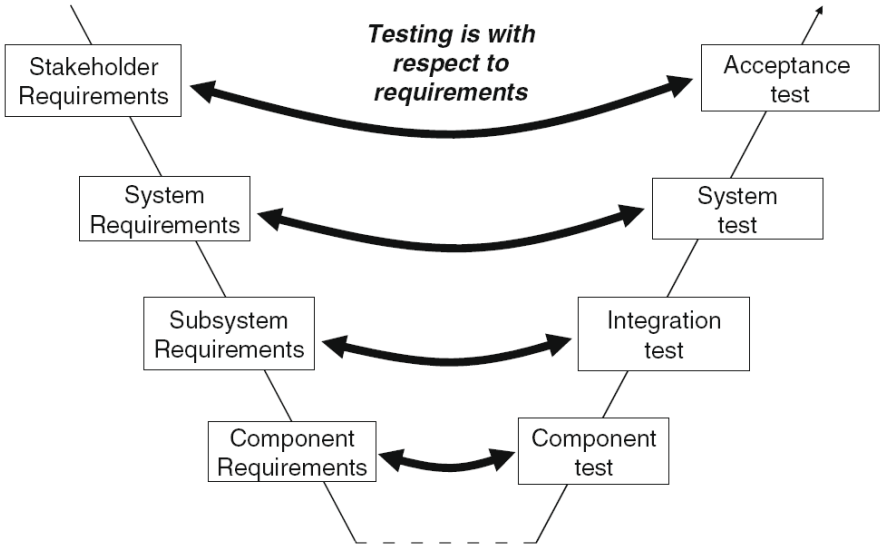

Figure 2.4: Testing activities and requirements (Hull et al., 2011)

If requirements are to play such a central role in systems development, they need to be maintained. Hence requirements engineering connects strongly with change management. Independently of where new or changed requirements come from, the impact of that change on quality, cost and schedule needs to be assessed. This assessment informs decisions to either accept or reject a change; negotiate the cost of change and organise and assign work to development teams (Hull et al., 2011). The purpose of *requirements management* is to anticipate and accommodate requirements changes, so as to minimize their disruptive impact on the project. Core requirements management activities include defining the requirements baseline; evaluating the impact of proposed requirements changes; establishing any relationships and dependencies between requirements and tracking requirements status and change activity throughout the project (Wiegers and Beatty, 2013).

The key concept that enables this kind of impact analysis is requirements tracing, primarily concerned with understanding how high-level requirements – objectives, goals, aims, aspirations, expectations, needs – are turned into low-level requirements. It is therefore primarily concerned with the relationships between layers of information. Requirements tracing allow measuring the impact of change, track progress against a set of requirements and assess benefit against cost of implementation (Hull et al., 2011).

2.3 Agile Requirements Engineering

Traditional software development methods, such as waterfall [1], advocate a simple top-down flow of requirements information (see figure 2.5). In this model, software development occurs in an orderly series of sequential stages. Requirements are agreed to, a design is created, and code follows thereafter. Lastly, the software is tested to verify its conformance to its requirements and design, and deployed to its users upon successful verification (Leffingwell, 2011).

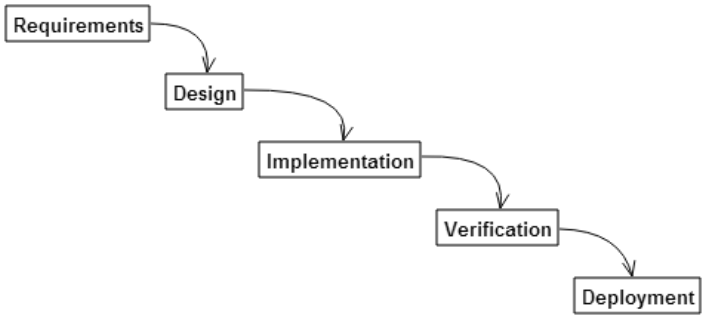

Figure 2.5: Sequential stages in waterfall development (Leffingwell, 2011)

The past decade has seen a movement to more lightweight and increasingly agile methods. Software technology has moved from supporting business operations to becoming a critical component of business strategy (Highsmith, 2000). The move towards agile methods was driven by the same causes that led manufacturers to transition from mass production to lean production techniques, namely a focus on quality, cost reduction and an increase in speed to market.

We will not detail an historical perspective of the evolution from predictive, waterfall-like methods to iterative and incremental processes (e.g RUP [2]) to the more recent agile and lean

[1] Royce (1970) is known to have first described the waterfall process, even though he did not use that term

[2] see *The Rational Unified Process: An Introduction*, Kruchten (2003) for details

development methods (Leffingwell, 2011; Larman, 2003). Instead, we describe an agile requirements artefact model and corresponding agile practices and principles, that compose the agile requirements approach found in current agile methods. In addition, we introduce BDD and explain it in the context of *Specification by Example* (Adzic, 2011) practices and principles.

2.3.1 Agile practices and principles

The agile manifesto [3] declares that: *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value*

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

That is, while there are value in the items on the right, we value the items on the left more

In each statement, the first part (in bold face) indicates a preference, whilst the other represents an item that, although relevant, is of lower priority. The authors of the manifesto chose their words carefully and the use of the word *'uncovering'* and the expression *'by doing it'*, place agile as a continuous incremental learning process carried out by practitioners in the software engineering field.

Agile development reverses the traditional approach of favouring processes and tools over people. In agile, the emphasis is much more on people collaboration and interaction than in following a plan or using a particular set of tools. Similarly, while comprehensive documentation is not a problem in itself, the emphasis should be in working software. Agile methods shift from strict contract negotiation to close collaboration between team members and customers, ensuring delivered software meets customer needs. Finally, agile realises that customer needs are not static and accepts changes to requirements, even if late in the project (Highsmith, 2000).

In agile development approaches we expect cycles and iteration among the business, user, and functional requirements (Wiegers and Beatty, 2013) and where goals are defined for each iteration and are revisited once the iteration is completed (Inayat et al., 2015a). Often, it is impossible or unnecessary to fully specify functional requirements before commencing design and implementation. The essence of agile development is learning just enough about requirements to do thoughtful prioritization and release planning so teams can begin delivering valuable software as quickly as possible (Wiegers and Beatty, 2013).

In response to a somehow fragmented knowledge about the solutions that agile brought to requirements engineering and the new challenges it has raised, Qasaimeh et al. (2008) reflect on the differences of 'traditional' and agile requirements engineering, the practices adopted by the latter and the solutions and challenges presented by adoption of agile requirements. The study compared different agile development methods, analysed their characteristics and classified them based on key requirements for a software development project. The authors analysed some of the most popular agile software methods such as Scrum [4], Extreme

[3] see http://www.agilemanifesto.org/ for details

[4] see *Agile Software Development with Scrum*, Schwaber and Beedle (2001) for details

Programming (XP) [5], Feature Driven Development (FDD) [6], Adaptive Software Development (ASD) [7] and Crystal Methodologies [8].

They concluded that *customer involvement* is a key practice in all agile processes and all analysed methods consider customers an integral part of the development process. Some of the methods advocate the presence of the customer on-site to elicit, prioritize and verify requirements and also during acceptance testing, where most agile processes require tests to be written and executed by customers.

To reduce *time to market*, most agile processes favour early delivery of software so that customers can use the software and provide feedback early on, improving defect rates and the customers understanding of the expected software features. Agile processes have the ability to quickly *respond to change*, with some processes relying on daily meetings with users, and promoting direct user interaction in determining changes to requirements and deciding what and when changes to requirements are going to be implemented. An informal approach to *documentation* is favoured and agile processes advocate face to face communication and presence of on-site user representatives.

The practices described are not specific to an agile development method, but rather have evolved from multiple uses and empirical studies of commonality across methods. A recent systematic literature review paper (Inayat et al., 2015b) identified the most common agile practices of agile requirements engineering (see Table 2.2).

All agile processes place focus on *verification and validation* of requirements, using testing techniques such as unit, integration and regression testing. Other quality review techniques are also used such as design and code inspections, retrospectives and code quality reviews. They also foster team communication and collaborative work by doing daily 'stand ups' – started in Scrum and now widely accepted as common practice, where all team members stand up around a circle, hence the name 'stand ups', for about 10 to 15 minutes and discuss what they have been doing since the last time they met and any issues or blockers they may be faced with – and use of code standards for facilitating exchange of information among team members.

2.3.2 Agile requirements artefact model

Nowadays, a significant number of organizations have made the transition to agile and that has brought to light common patterns for agile software processes. Leffingwell (2011) introduces the idea of the *Agile Enterprise Big Picture* (see figure 2.6) with the goal of sharing a language for discussion, a set of abstractions, and a visual model that describes agile software development and delivery process mechanisms, the teams and organizational units, and some of the roles key individuals play in the new agile paradigm. The vocabulary introduced is a method-independent set of constructs widely used in the industry and generally accepted in current agile development practices.

Leffingwell describes agile at scale at different levels of detail and with increasing abstraction, from *Team* to *Program* and finally *Portfolio*.

At the *Team* level

> ... agile teams define, build, and test user stories in a series of iterations and releases. In the smallest enterprise, there may be only a few such teams. In larger

[5] see *Extreme Programming Explained: Embrace Change*, Beck (2000) for details

[6] see *A Practical Guide to Feature-Driven Development* Palmer and Felsing (2001) for details

[7] see *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Highsmith (2000) for details

[8] see *Crystal Clear a Human-powered Methodology for Small Teams* and Cockburn (2004) for details

[9] see http://www.agilemanifesto.org/ for details

Table 2.2: List of popular agile practices (Inayat et al., 2015b)

Practice	Description
Acceptance tests	In an agile context, refer to tests created and applied for each of the defined user stories, confirming their correctness and determining the completeness of a user story implementation. In practice, acceptance tests are small notes written at the back of story cards
Code refactoring	To restructure software by applying a series of changes to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
Cross-functional teams	A group of individuals with different functional expertise working towards a common goal. In agile methods, developers, testers, designers, and managers work closely together, helping to bridge communication gaps (Adzic, 2009)
Customer involvement	Agile methods rely on frequent collaboration with an accessible and available on-site customer
Face-to-face communication	Agile processes advocate minimal documentation in the form of user stories and discourage long and complex specification documents, favouring frequent face-to-face communication
Iterative requirements	Unlike in traditional software development methods, requirements emerge over time in agile methods through frequent interaction among stakeholders and gradual detailing of requirements
Prototyping	Promotes quicker feedback and enhances customer anticipation of the product, allowing timely feedback prior to moving to subsequent iterations
Requirements management	Performed by maintaining product backlog/feature lists and index cards
Requirements prioritisation	In agile methods, requirements are prioritised continuously in each development cycle by customers who focus on business value
Retrospectives	At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly [9]
Testing before coding	Commonly known as Test Driven Development, means to write tests prior to writing functional codes for requirements. It promotes feedback in the case of test failures
User Stories	Are created as specifications of the customer requirements. User stories facilitate communication and better overall understanding among stakeholders. A user story describes functionality of a system or software and is composed of a written description used for planning and as a reminder; follow-up conversations that serve to flesh out the details of the story and tests that convey and document details and that can be used to determine when a story is complete (Cohn, 2004).

enterprises, groups, or pods, of agile teams work together to support building up larger functionality into complete products, features, architectural components, subsystems, and so on. The responsibility for managing the backlog of user stories belongs to the team's product owner (Leffingwell, 2011)

At the *Program* level

...the development of larger-scale systems functionality is accomplished via mul-

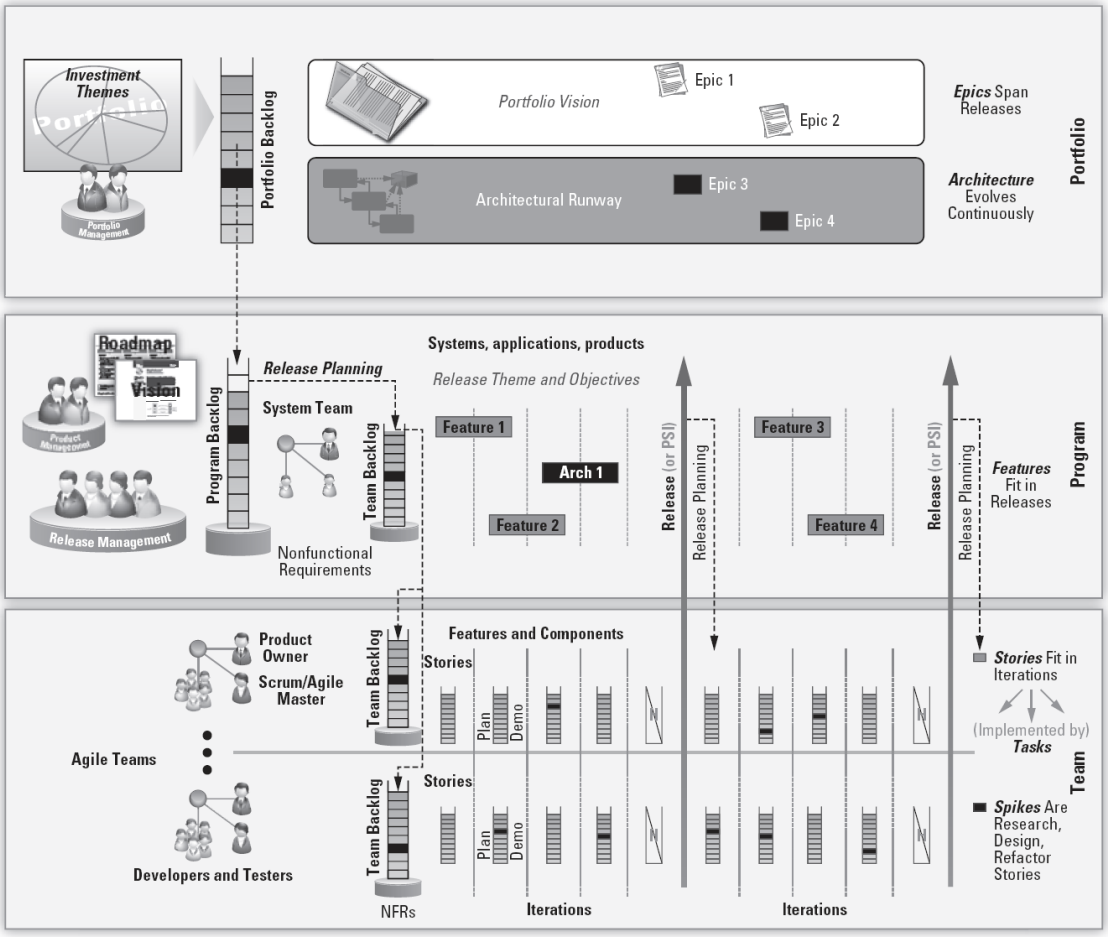

Figure 2.6: Agile development levels: *Team*, *Program* and *Portfolio* (Leffingwell, 2011)

> tiple teams in a synchronized standard cadence of time-boxed iterations and milestones that are date and quality fixed, but scope is variable, producing releases or potentially shippable increments (PSI) at frequent, typically fixed, 60 to 120 day time boundaries. (Leffingwell, 2011)

At the *Portfolio* level

> ... a mix of themes are used to drive the investment priorities for the enterprise. That construct assures that the work being performed is the work necessary for the enterprise to deliver on its chosen business strategy. Investment themes drive the portfolio vision, which is expressed as a series of larger, epic-scale initiatives, which will be allocated to various release trains over time (Leffingwell, 2011)

In the context of this thesis, *Team* and *Program* levels are the most relevant. In addition, together with an agile requirements artefact meta-model (see figure 2.7), we are in possession of an unambiguous and coherent language for doing research in agile requirements engineering. We will delve less into the organisational and team composition aspects of the model, but instead focus on the elements introduced by the meta-model.

The meta-model defines the concept of a *Backlog Item* – an abstract entity representing something that needs to be done, either a user story or another work item – and a *Backlog*, a repository of all the work items the team has identified. This backlog is the one and only definitive source of work for the team. A *Story* is the rough equivalent of a software requirement

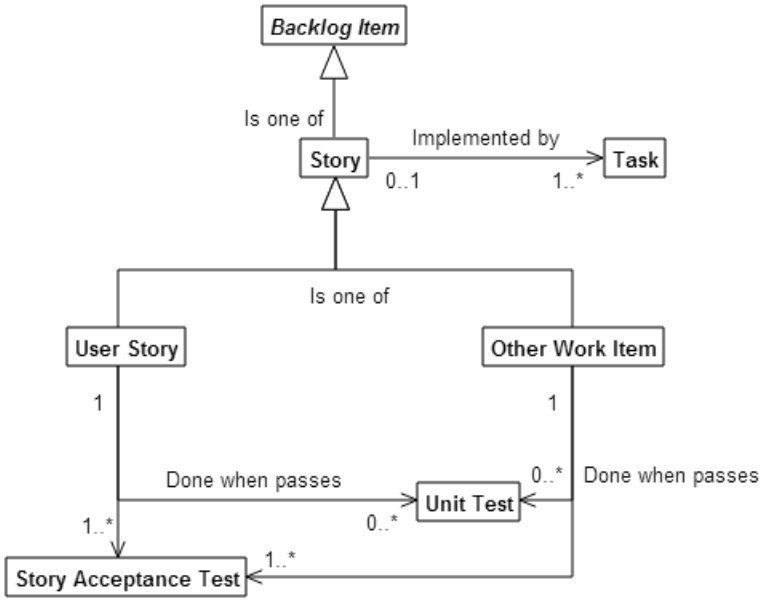

Figure 2.7: Agile requirements meta-model for the *Team*

in agile and, at the *Team* level, we distinguish two types: a *User Story* – used to define the system behaviour and determine value for the user – and *other work items* such as defects, documentation, support and maintenance activities, infra-structure work and so on. Leffingwell (2011) defines *Story* as *'a work item contained in the team's backlog'* and *User Story* as *'a brief statement of intent that describes something the system needs to do for the user'*.

Tasks are used to breakdown a *Story* in work activities required to its implementation. Note that *Tasks* can exist on their own and without an associated *Story*, if the work activity is considered independent and can standalone. Also, a *Story* requires one or more tasks for its implementation and is only complete when it passes one or more *acceptance tests*. A *Story* could also be subject to *unit tests* to confirm that the lowest-level module of an application or module of an application works as intended.

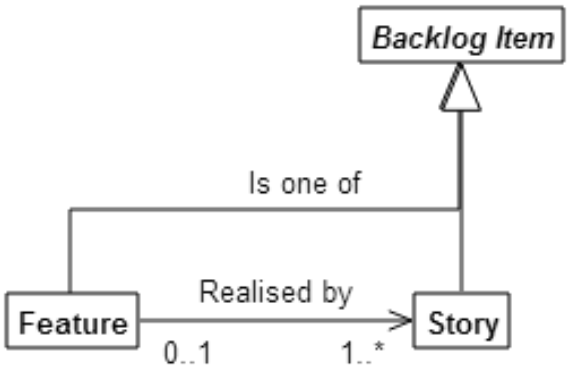

Figure 2.8: Agile requirements meta-model (*Features*)

At *Program* level, the backlog contains a prioritized set of *features* intended to deliver benefits to the users. Leffingwell (2011) defines *Features* as *'services provided by the system that fulfil stakeholder needs'*. *Features* are a kind of backlog item as can be seen in figure 2.8. *Features* are at a higher level of abstraction than *Stories* and sit between needs of users and

software requirements, expressed in agile as *Stories*. A given *Feature* is realised by one or more *Stories*.

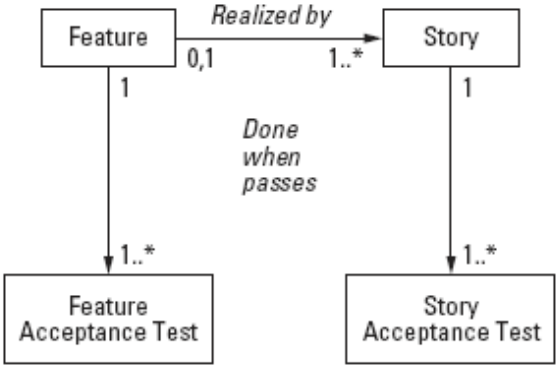

Figure 2.9: Agile requirements meta-model (*Features* and *Acceptance Tests*)

Similar to *Stories*, *Features* also require acceptance tests to ensure all stories that realise it are complete. Note that a *Feature acceptance test* is not a composition of *Story acceptance tests*, as they should focus on other types of tests such as performance, 'what-if' scenarios, etc.

Features and *Stories* are used to specify the functionality of a system but we should not ignore non-functional requirements. In the model in figure 2.10, we see first that some backlog items may be constrained by non-functional requirements, and some may not. We also see that non-functional requirements may not apply to backlog items, meaning that they stand independently and apply to the system as a whole (Leffingwell, 2011).

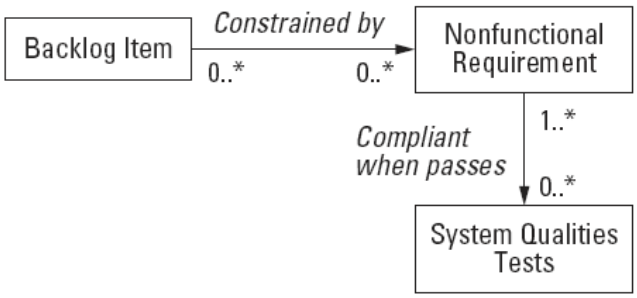

Figure 2.10: Agile requirements meta-model (*Non-functional requirements*)

It is important to note that a non-functional requirement constrains a *Backlog item* which can be either *Features* or *Stories*. This is important from a practical perspective as it is not uncommon to have agile teams starting with *Features* to organise their *backlog*. This is also the case in some agile approaches such as the one described in the next section.

2.3.3 Behaviour driven development (BDD)

Hull et al. (2011) advocates the use of modelling techniques as a mechanism of fostering understanding and communication of ideas associated with system development. A good model is one which is easily communicated. They need to be used for communication within a development team, and also to an organisation as a whole including the stakeholders. The uses of a model can be diverse and cover a wide spectrum. It might be to model the activities of an

entire organisation or to model a specific functional requirement of a system. It is this latter use that receives the attention of Behaviour Driven Development which is, in its essence, an approach to derive the functionality of a system or component from business goals using concrete examples (see figure 2.11).

In the foreword to *BDD in Action: Behavior-driven development for the whole software lifecycle*, Dan North the creator of BDD states the following *'... (BDD) was a response to a triple conundrum: programmers didn't want to write tests; testers didn't want programmers writing tests; and business stakeholders didn't see any value in anything that wasn't production code'*.

BDD applies at all levels of software development, from high-level requirements discovery and specification to detailed low-level coding, whilst promoting the discovery of requirements and automation of high-level acceptance criteria, build and verification of the design and implementation, and production of accurate and up-to-date technical and functional documentation. (Smart, 2014).

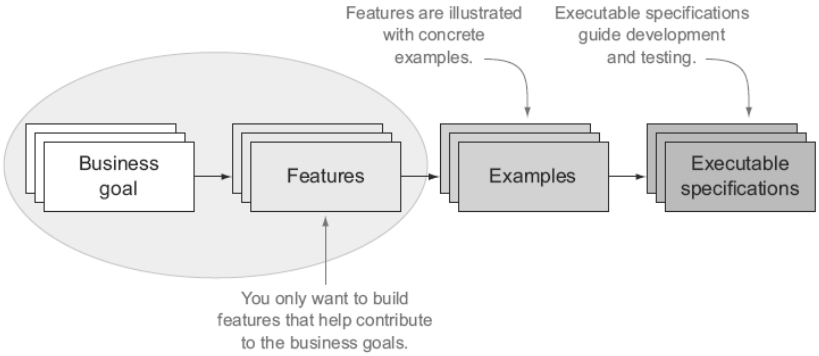

Figure 2.11: BDD: From business goals to executable specifications (Smart, 2014)

BDD helps teams focus their efforts on identifying, understanding, and building valuable features that matter to businesses, and it makes sure that these features are well designed and well implemented (Smart, 2014). BDD practitioners use conversations around concrete examples of system behaviour to help understand how features will provide value to the business. Furthermore, it encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand. Tools exist that can help turn these requirements into automated tests that help guide the developer, verify the feature, and document what the application does (Smart, 2014; Wynne and Hellesoy, 2012). Figure 2.11 depicts the typical flow of information in BDD, from business goals to features, followed by identification of concrete examples, all part of a specification that gets tested and verified against initial business goals stated.

Some authors and practitioners do not consider BDD a software development methodology in its own right, but as a set of methods and techniques grouped under the same label, which incorporates, builds on, and enhances ideas from many agile and iterative methodologies (Smart, 2014).

It is important to note that BDD was developed as a mechanism for fostering collaboration, improving communication and requirements discovery through examples. In traditional development methods, requirements follow a sequential set of activities where typically a business owner informs an analyst of his needs and goals, who in turn will write a requirements

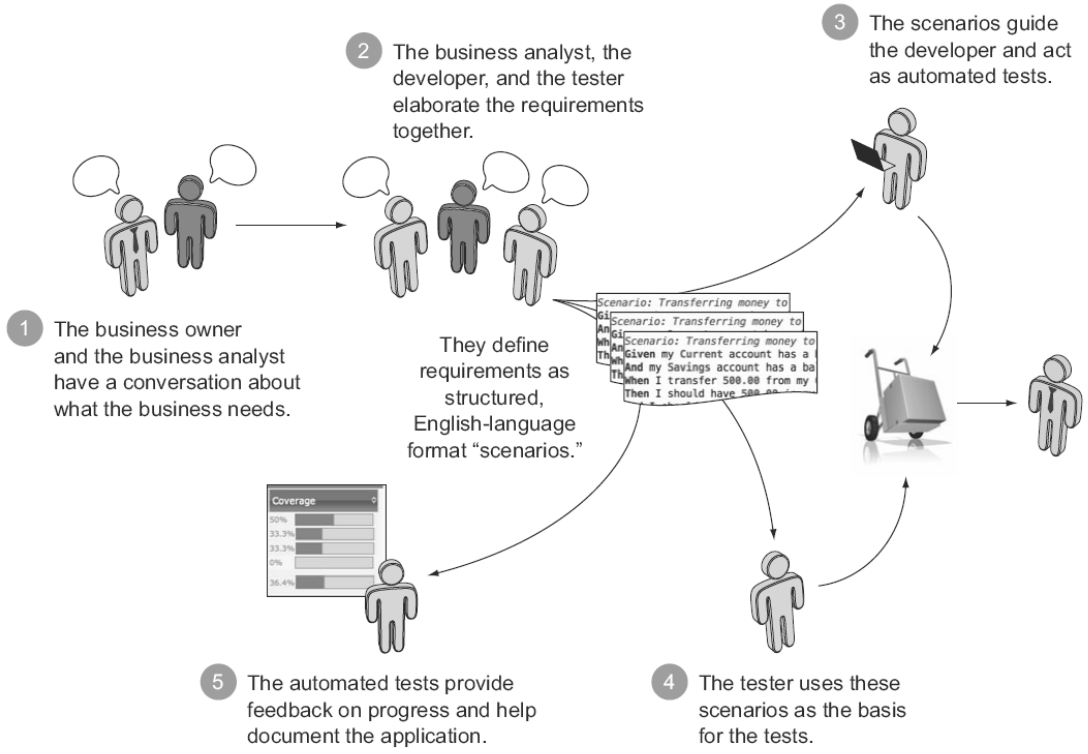

Figure 2.12: Improved communication in BDD (Smart, 2014)

document that a developer translates into software. Simultaneously, or upon development is complete, a tester translates the same requirements document into test cases which, if executed with success, lead to user and technical documentation being produced and the software being deployed to end-users. In BDD the focus is on collaboration, with all interested parties sharing a structured specification that is used to simultaneously, specify features to be implemented and tests to be executed.

BDD can also be seen as an instance of *Specification by Example* (Adzic, 2011) as they both share a common set of principles and practices. *Specification by example* introduces a consistent and coherent language for patterns, ideas and artefacts used for teams who derive *executable specifications* and *living documentation* from *business goals*. The practices of *Specification by Example* do not form a fully fledged software development methodology but rather supplement other methodologies – both iterative and flow based – to provide rigour in specifications and testing, enhance communication between various stakeholders and members of the software development team (Adzic, 2011).

Instead of relying on users to provide requirements, teams *derive scope from goals*, taking customer's business goals and defining the scope in terms of the set of features that achieve those goals. This is done collaboratively with business users and team members, to improve communication and reduce unnecessary rework. *Specifying collaboratively* they are able to harness the knowledge and experience of all team members. It also creates a collective ownership of specifications, making everyone more engaged in the delivery process (Adzic, 2011).

Teams *illustrate specifications using examples*. The team works with business users to identify key examples describing expected functionality, flushing out functional gaps and inconsistencies and ensuring that everyone involved has a shared understanding of what needs to be delivered, avoiding rework that results from misinterpretation and translation (Adzic, 2011).

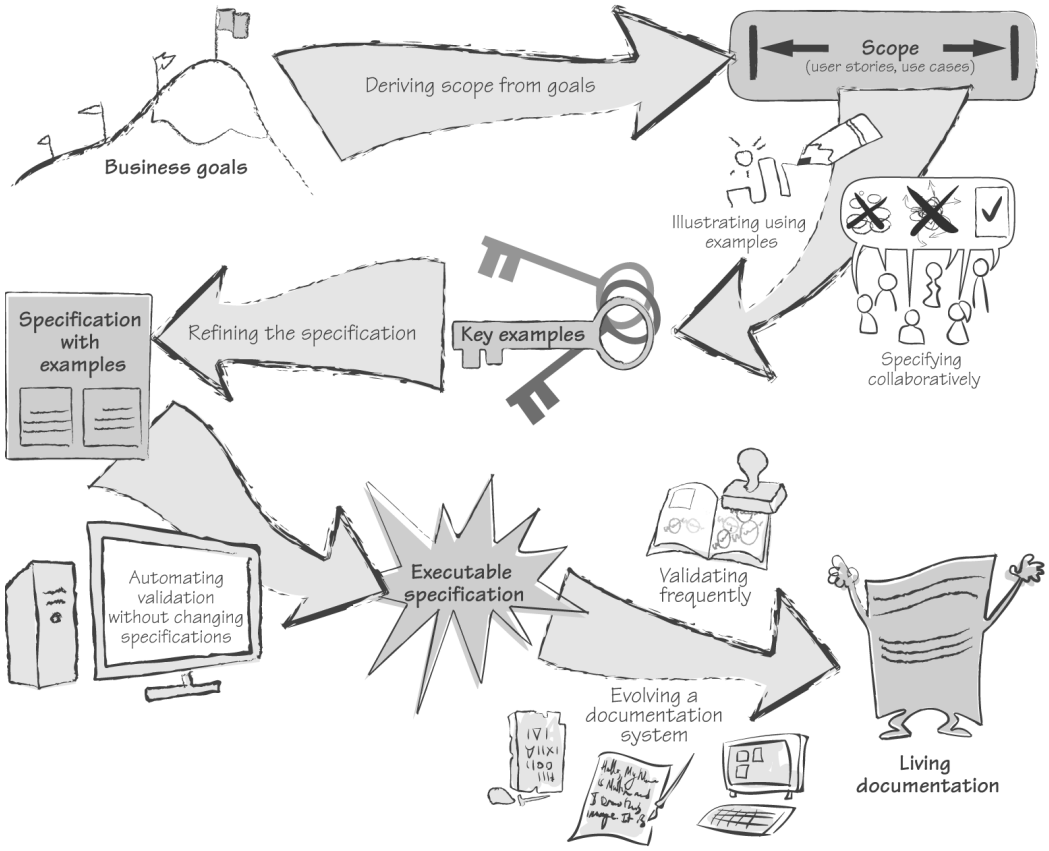

Figure 2.13: Process patterns of *Specification by Example* (Adzic, 2011)

Key examples must be concise to be useful. By *refining the specification*, successful teams remove extraneous information and create a concrete and precise context for development and testing. They define the target with the right amount of detail to implement and verify it. They identify what the software is supposed to do, not how it does it (Adzic, 2011).

Once a team agrees on *specifications with examples* and refines them, the team can use them as a target for implementation and a means to validate the product. To get the most out of key examples, successful teams automate validation without changing the information. As they *automate validation without changing specifications*, the key examples are always comprehensible and accessible to all team members. An automated *Specification with examples* that is comprehensible and accessible to all team members becomes an *executable specification*. We can use it as a target for development and easily check if the system does what was agreed on, and we can use that same document to get clarification from business users (Adzic, 2011).

Validating frequently executable specifications against the system ensures we can discover any differences between the system and the specifications and keep both synchronised in face of changes to requirements or implementation. Not only do teams validate frequently, they also ensure that specifications are actual, current and consistent, effectively turning them into *living documentation*.

In BDD, *executable specifications* are expressed using *Gherkin* (Wynne and Hellesoy, 2012), a domain specific language used to specify desired features of a system or application (see feature 2.1 for an example of a specification in Gherkin).

```
Feature: Feedback when entering invalid credit card details

  In user testing we've seen a lot of people who made mistakes
  entering their credit card. We need to be as helpful as possible
  here to avoid losing users at this crucial stage of the
  transaction.

  Background:
    Given I have chosen some items to buy
    And I am about to enter my credit card details

  Scenario: Credit card number too short
    When I enter a card number that's only 15 digits long
    And all the other details are correct
    And I submit the form
    Then the form should be redisplayed
    And I should see a message advising me of the correct number of digits

  Scenario: Expiry date invalid
    When I enter a card expiry date that's in the past
    And all the other details are correct
    And I submit the form
    Then the form should be redisplayed
    And I should see a message telling me the expiry date must be wrong
```

Feature 2.1: Example of a feature file in Gherkin (Wynne and Hellesoy, 2012)

Gherkin specifications are defined from plain-language text files called *feature files*, containing scenarios to implement and test, representing concrete examples of the features being specified. Each scenario is a list of steps for a BDD tool such as Cucumber to work through (Wynne and Hellesoy, 2012). These tools parse feature files turning each scenario step into source code, typically a method call in one of the tool's supported languages. Feature 2.1 displays a feature file, specified in *Gherkin*, for a credit card application containing two scenarios with concrete examples of credit card validation rules. In Gherkin, we use the *Feature* keyword to name (text after *Feature* keyword, excluding colon character) and describe (all remaining text until the next keyword) a feature. Gherkin uses the *Background* keyword to specify a set of steps that are common to every scenario and within each scenario *Given* is used to set up the context where the scenario happens, *When* to interact with the system somehow and *Then* to check that the outcome of that interaction was what we expected. *And* and *But* keywords are used to make specifications more readable and take the same role as the keyword in the previous line.

2.3.4 Summary

In this chapter we have provided definitions of requirements and presented them within the context of requirements engineering. We followed with an analysis of core activities of requirements engineering and described how they manifest in the context of agile development methods. Finally, we introduced BDD in context with *Specification by Example* and gave an example a feature specification in Gherkin.

It is important to state that in BDD, and other agile approaches in general, handling of non-functional requirements is ill defined. Customers or users talking about what they want the system to do normally do not think about resources, maintainability, portability or performance (Paetsch et al., 2003). In the next chapter we will explore further the notion of non-functional requirements, present some well known classification schemes, and methods used for their elicitation and analysis with a focus on goal-oriented approaches and GRL (Amyot et al., 2010), in particular.

3 On non-functional requirements

In the previous chapter we have followed a generic approach to requirements engineering, without focusing on any particular requirement type. In this chapter, we take a different view and focus on the concept of non-functionality in requirements engineering.

It is consensual that a system's utility is determined by both its functional and non-functional characteristics, such as usability, flexibility and performance (Chung and Leite, 2009).

The perception of quality is determined by these two characteristic sets, and therefore, they must be taken into consideration in the development of software systems. However, most of the attention in software engineering in the past has been centred on notations and techniques for defining and providing the functions of a software system (Chung and Leite, 2009). Additionally, in the occasions where non-functionality is taken into consideration, the needed quality characteristics are treated only as technical issues related mostly to the detailed design or testing of an implemented system (Chung and Leite, 2009).

Chung and Leite (2009) noted that real-world problems are more non-functionally oriented than they are functionally oriented, e.g., poor productivity, slow processing, high cost, low quality and unhappy customers. Rather than worry about precisely how to refer to these information types, it is more pertinent to ensure they are part of requirements elicitation and analysis activities. A product can be delivered with the desired functionality but that users hate because it doesn't match their (often unstated) quality expectations (Wiegers and Beatty, 2013).

In the remainder of this chapter, we will highlight issues with current definitions of non-functional requirements. We will also list relevant classification and representation schemes. Finally, we will introduce goal-oriented approaches to handling non-functional requirements, with a focus on goal-oriented requirements language (GRL).

3.1 Definition

For many years, the requirements for a software product have been classified broadly as either functional or non-functional. The functional requirements are evident: they describe the observable behaviour of the system under various conditions. However, many people dislike the term 'non-functional' (Wiegers and Beatty, 2013).

These 'other-than-functional' requirements, could refer to *how* well a system performs its functions, rather than to *what* those functions may be. They could describe important characteristics or properties of a system, such as availability or performance. Likewise, they could be considered as *quality attributes* (ISO/IEC/IEEE 1061:1998), but that view ignores other aspects such as design and implementation constraints or business rules, which we could also associate with non-functionality (Wiegers and Beatty, 2013).

We now provide three selected definitions, not because of their special correctness or attractiveness, but because they are amongst the more popular ones and support the argument that present definitions are inconsistent, ambiguous and confusing at times (Glinz, 2007).

Wiegers and Beatty (2013) defines non-functional requirements as

> ... descriptions of a *property* or *characteristic* that a software system must exhibit or a *constraint* that it must respect, other than an observable system behaviour (Wiegers and Beatty, 2013)

while Jacobson et al. (1999) defines them as

> .. a requirement that specifies system properties, such as environmental and *implementation constraints*, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies *physical constraints* on a functional requirement. (Jacobson et al., 1999)

and Sommerville and Kotonya (1998) as requirements which are

> ... not specifically concerned with the functionality of a system, instead specifying restrictions on the product being developed and the development process, and *external constraints* the product must meet. (Sommerville and Kotonya, 1998)

Glinz (2007) identifies terminological, but also major conceptual issues with current definitions of non-functional requirements. Most definitions use the terms *property* or *characteristic*, *attribute*, *quality* or *constraint* differently, and the meaning of those terms is not always clear. For example, *property* and *characteristic* denote something that the system must have, but that is equally a criteria for inclusion in functional requirements definitions. Also, every non-functional requirement (or functional, for this matter) can be regarded as a quality of a system or as a constraint, because it restricts the space of potential solutions to those that meet this requirement (Glinz, 2007). In addition, there are references to implementation, physical and external constraints and not only there is a lack of clear guidance as to what they intend to restrict, also, these restrictions or constraints are known by different terms in other definitions, such as *'interface requirements'* or *'design constraints'*. Finally, *performance* is treated as a quality or attribute is most definitions, but it deserves a category of its own in some quality models (ISO/IEC/IEEE 29148:2011).

3.2 Classification and representation schemes

The definitions we have just alluded to, refer to concepts that are considered to be part of non-functional requirements. However, there are more detailed classification schemes that are worth our attention.

Roman (1985) provides one such classification scheme based on the notion of constraints and types of constraints. *Interface* constraints, define the way components of a system or application and its environment, interact. *Performance* constraints, cover a broad range of issues dealing with time/space bounds, reliability, security, and survivability. *Operating* constraints, include physical (e.g., size, weight, power, etc.), personnel availability, skill level considerations, accessibility for maintenance, environmental conditions (e.g., temperature, radiation, etc.), and spatial distribution of components. *Life-cycle* constraints, fall into two broad categories: those that pertain to qualities of the design, such as maintainability or portability, and those that limit the development, maintenance, and enhancement process. *Economic* constraints, represent considerations relating to immediate and long term costs. Finally, *political* constraints deal with policy and legal issues. This view of non-functional requirements as constraints of components of a system or application will be explored further in Chapter 4.

Another classification scheme is introduced by ISO/IEC 9126-1:2001 and its interpretation of software quality. Software product quality can be evaluated by measuring internal attributes

(typically static measures of intermediate products), or by measuring external attributes (typically by measuring the behaviour of the code when executed), or by measuring quality in use attributes. The objective is for the product to have the required effect in a particular context of use.

This standard defined a quality model for external and internal quality. It categorises software quality attributes into six characteristics (functionality, reliability, usability, efficiency, maintainability and portability), which are further subdivided into sub-characteristics

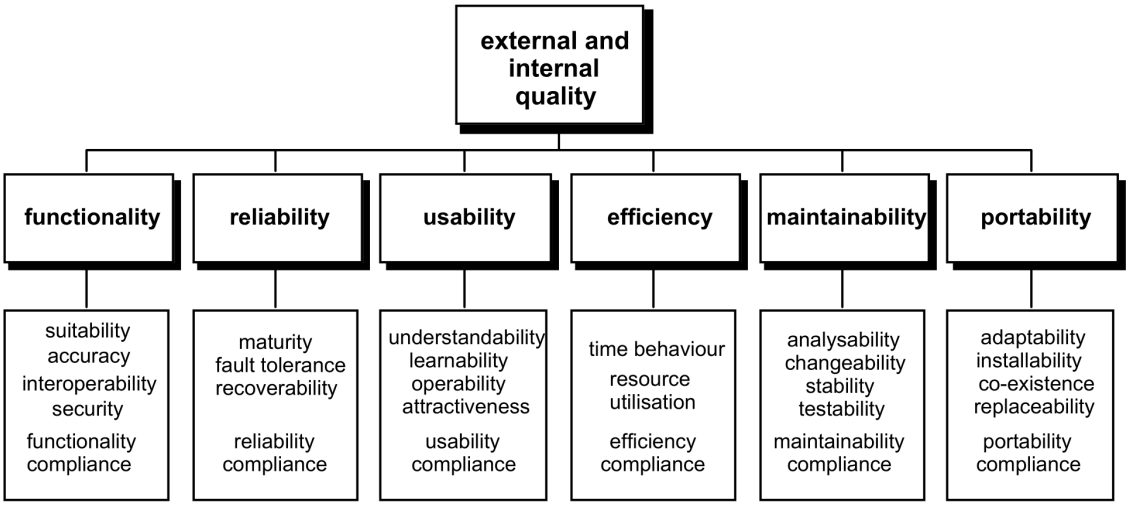

Figure 3.1: Software product quality model in ISO/IEC 9126-1:2001

Interestingly, several issues have been identified with ISO/IEC 9126-1:2001 such as terminology inconsistencies with other ISO standards; ambiguities in the way it is structured in terms of characteristics and sub-characteristics; incomplete set of characteristics and sub-characteristics, etc. (Al-Qutaish, 2009).

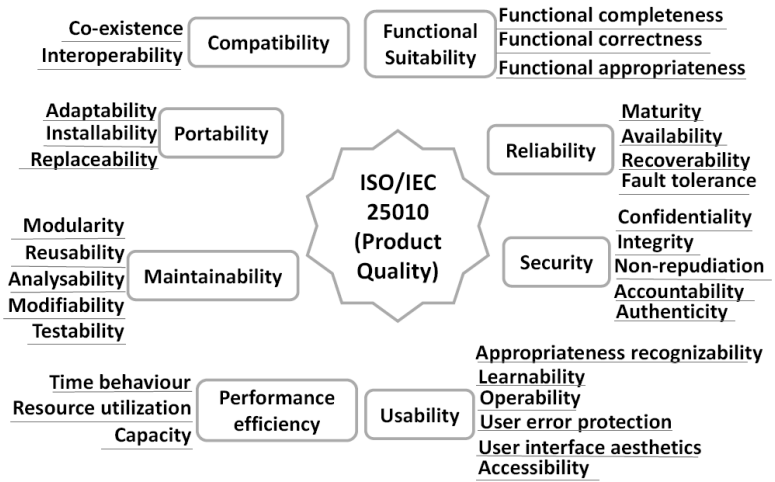

Figure 3.2: ISO 25010 quality model in ISO/IEC 25010:2011

This has lead to a revised standard that extends this quality model with eight product quality characteristics and 31 sub-characteristics (ISO/IEC 25010:2011) (see figure 3.2).

Another classification sheme is FURPS (Grady and Caswell, 1987), an acronym representing a model for classifying software quality attributes or non-functional requirements, based

on five categories: functionality, usability, reliability, performance and supportability (see table 3.1). The original FURPS model was later extended to empathize various specific attributes (Adams, 2015).

Table 3.1: FURPS quality model (Grady and Caswell, 1987)

Category	**Attribute**
Functionality	Feature set
	Capabilities
	Generality
	Security
Usability	Human factors
	Aesthetics
	Consistency
	Documentation
Reliability	Frequency/severity of failure
	Recoverability
	Predictability
	Accuracy
	Mean time to failure
Performance	Speed
	Efficiency
	Resource consumption
	Throughput
	Response time
Supportability	Testability
	Extensibility
	Adaptability
	Maintainability
	Compatibility
	Configurability
	Serviceability
	Installability
	Localizability

Some classification schemes emerge from research work carried out in specific domains. For example, in the context of Service Oriented Architectures (SOA) [1], Becha and Amyot (2012) defined a catalogue of generic (i.e., domain independent) non-functional properties to be considered when service descriptions are developed including reliability, usability and security, but also others less obvious such as price – the fee that the service consumer is expected to pay for invoking a given service – or reputation of service, understood as the opinion of service consumers towards a service.

A more recent classification scheme is provided by Adams (2015). In this work, over 200 non-functional requirements are reduced, using results reported in eight models from the extant literature. The 27 resultant non-functional requirements have been organized in a taxonomy that categorizes the 27 major nonfunctional requirements within four distinct categories: *System Design Concerns*, *System Adaptation Concerns*, *System Viability Concerns* and *System Sustainment Concerns*. Figure 3.3 show the relationship between the four system concerns and the 27 non-functional requirements selected for consideration during a system's life cycle.

[1] https://en.wikipedia.org/wiki/Service-oriented_architecture

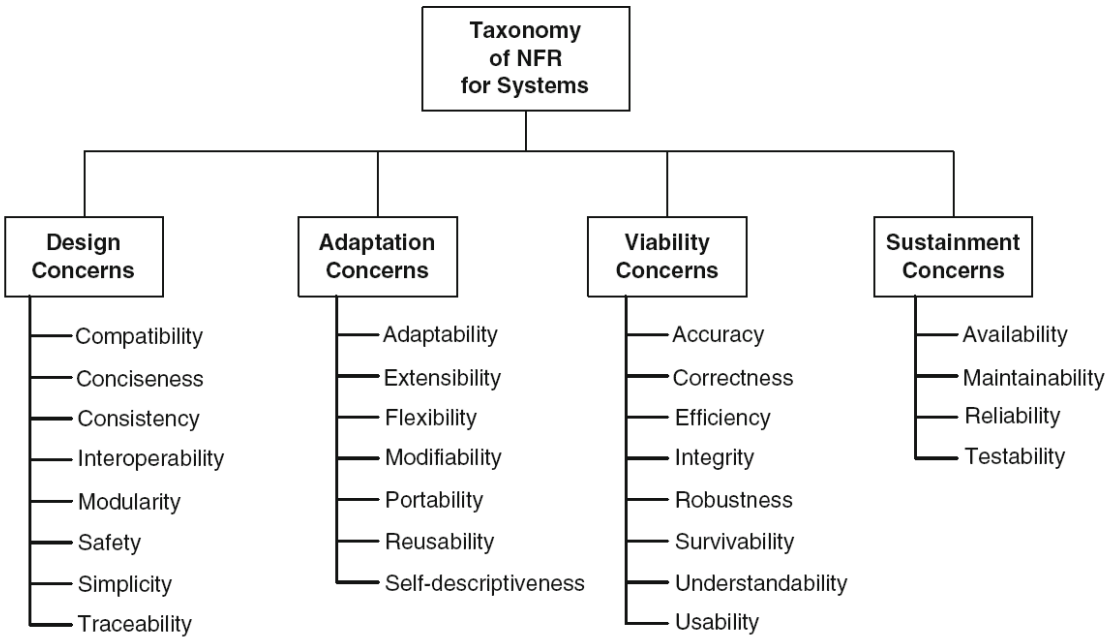

Figure 3.3: Taxonomy of non-functional requirements for systems (Adams, 2015)

There have also been attempts to automate the classification of non-functional requirements. One of such approaches, detects and classifies stakeholders quality concerns across requirements specifications containing scattered and non-categorized requirements, and also across free-form documents such as meeting minutes or notes (Cleland-Huang et al., 2007). A similar approach, proposes a semi-supervised text categorization for the automatic identification and classification of non-functional requirements (Casamayor et al., 2010).

According to Glinz (2007), the categorization and sub-classification of non-functional requirements in current classification schemes, reflect rather divergent concepts. For instance, whilst Roman (1985) presents a classification scheme based on the notion of constraints, Adams (2015) devises another scheme around concerns. Glinz had previously found more classification problems due to mixing three concepts that should better be separated. These are the concepts of *kind* (should a given requirement be regarded as a function, a quality, a constraint, etc.), *representation* and *satisfaction* (hard vs. soft requirements) (Glinz, 2005).

A software practitioner should be aware of the well known classification schemes and their limitations, such as the terminology and categorically inconsistencies mentioned above. More than be concerned with choosing the right scheme, he or she should know what each attribute or quality characteristic means, such as performance, so that the correct intent can be communicated between users, developers and testers who, ultimately, will implement and verify appropriate features in the product or application (Chung and Leite, 2009).

According to Laleau and Matoussi (2008), there are three extreme ways of specifying software requirements: complete formal specifications, informal specifications and hybrid or semi-formal approaches. We proceed by sampling from the literature, several approaches belonging to the above mentioned categories.

No-Fun (Franch, 1998) belongs to the formal category of representation schemes. *NoFun* (acronym for 'NOn-FUNctional'), is a formal language for the description of software quality consisting of a hierarchy of software quality characteristics and attributes formulated as abstract and concrete quality models; component quality descriptions through assignment of values to

component quality basic attributes, and finally, quality requirements stated over components, both context-free (universal quality properties) and context-dependent (quality properties for a given framework software domain, company, project, etc.) (Franch, 1998). In another paper Botella et al., 2001, the framework was applied to the set of quality attributes refinements that are part of ISO/IEC 9126-1 (2001).

Chung and Leite (2009) mention that an informal and common way to represent non-functional requirements is by means of requirements sentences, which are commonly listed separately under different sections of a requirements document and refers to ISO/IEC/IEEE 830:1998, superseded by ISO/IEC/IEEE 29148:2011, as an example.

There are also informal but structured approaches around requirements sentences, such as the *Volere* requirements process (Robertson and Robertson, 1999), that is comprised of an identification number, NFR type, use case related to it, description, rationale, originator, fit criterion, customer satisfaction, customer dissatisfaction, priority, conflicts, supporting material, and history (Chung and Leite, 2009).

In spite of all of the above ways by which non-functional requirements have been classified and represented, goal-oriented approaches were the first to treat non-functional requirements in more depth, dealing with their representation but also conflict detection amongst multiple non-functional requirements.

3.3 Goal-oriented approaches

The use of goals in requirements engineering has received increasing attention over the past few years. Such recognition has led to a whole stream of research on goal modelling, goal specification, and goal-based reasoning for multiple purposes, such as requirements elaboration, verification or conflict management, and under multiple forms, from informal to qualitative to formal (Van Lamsweerde, 2001).

A goal captures, at different levels of abstraction, the various objectives the system under consideration should achieve (Van Lamsweerde, 2001). Goals also cover functional concerns associated with the services to be provided, and non-functional ones, associated with quality of service, such as safety, security, accuracy, performance, and so forth.

The *NFR Framework* (Chung et al., 1999) has probably been the first requirements model to address the lack of a proper treatment of quality characteristics by addressing both functional and non-functional requirements as a whole, and non-functionality at a high level of abstraction for both the problem and the solution (Chung and Leite, 2009).

In the *NFR Framework*, non-functional requirements are treated as softgoals, i.e., goals that need to be addressed not absolutely but in a good enough sense. Reflecting the sense of 'good enough', the *NFR Framework* introduces the notion of satisficing, and, with this notion, a softgoal is said to satisfice (instead of satisfy) another softgoal. Softgoals are related through relationships which represent the influence or interdependency of one softgoal on another. A qualitative analysis method is included in the framework for deciding the status of softgoals, given that other, related softgoals are satisfied or have been found to be unsatisfiable (Chung et al., 1999).

Figure 3.4 shows in graphical form the most important concepts of the *NFR Framework*. Softgoals, which are "soft" in nature, are shown as clouds and can be decomposed in other more specific soft-goals though decomposition links. We say that softgoals are refined downwards into subgoals, and subgoals contribute upwards to parent softgoals. There are two main types of

[2] sourced from http://www.utdallas.edu/ supakkul/NFR-modeling/label-evaluation

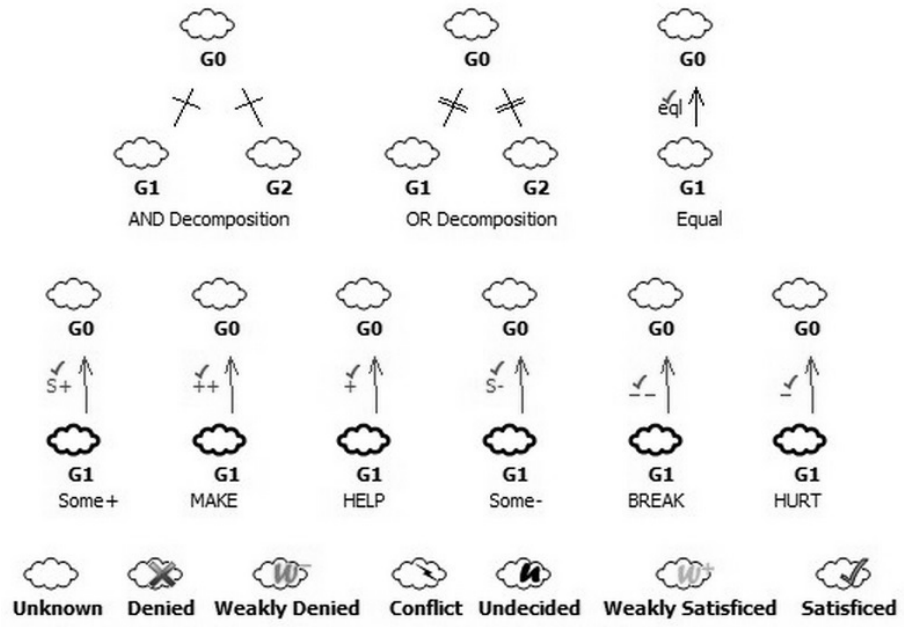

Figure 3.4: Catalogue of visual elements in NFR Framework [2]

decompositions *AND* and *OR*, with the former used when several subgoals are needed together to meet a higher softgoal, and the latter when one subgoal alone is sufficient.

When the non-functional requirements have been sufficiently refined, the next step is to identify *operationalisations* (tick dark clouds), which are possible development techniques or specific solutions for achieving these non-functional requirements. In the *NFR Framework*, each softgoal or contribution is associated with a *label*, indicating the degree to which it is satisficed or denied.

The *NFR Framework* offers several different types of contributions whereby a softgoal satisfices, or denies, another softgoal - *MAKE*, *HELP*, *HURT* and *BREAK* are the prominent ones. While *MAKE* and *BREAK* respectively reflect our level of confidence in one softgoal fully satisficing or denying another, *HELP* and *HURT* respectively reflect our level of confidence in one softgoal partially satisficing or denying another (Chung and Leite, 2009).

The NFR Framework uses non-functional requirements such as security, accuracy, performance and cost to drive the overall design process. The framework, offers a structure for representing and recording the design and reasoning process in graphs, called softgoal interdependency graph (SIG) (Chung et al., 1999).

Main requirements are shown as *softgoals* at the top of a graph. Softgoals are connected by *interdependency links*, which are shown as lines, often with arrowheads. Softgoals have associated *labels* (values representing the degree to which a softgoal is achieved) which are used to support the reasoning process during design (Chung et al., 1999).

Figure 3.5 shows a SIG for a hypothetical credit card application where security of account information, and good performance in the storing and updating of that information have been elicited as non-functional requirements to attain.

These two non-functional requirements, softgoals in the *NFR Framework* terminology, are represented as clouds at the top of the figure. Softgoals have a type (in this case, security and performance) and a topic, which refers to an entity from the domain or subject matter,

[3] sourced from http://www.utdallas.edu/ supakkul/NFRs-catalog/

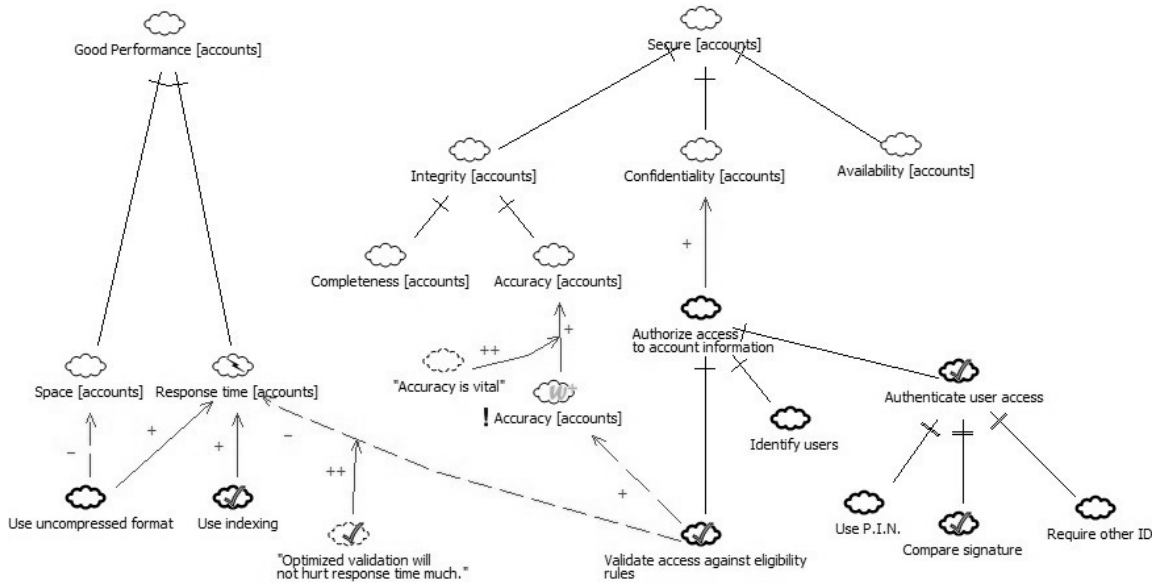

Figure 3.5: Example of a Softgoal Interdependency Graph (SIG) [3]

in this case bank accounts. In the figure, secure accounts softgoal is decomposed into sub-softgoals integrity, confidentiality and availability. In this case, the developer or designer has taken the view that authorizing access to account information would help attain confidentiality of accounts softgoal. This authorization access, is then refined into an *OR* decomposition of operationalisations, namely 'Use pin', 'Compare signature' or 'Require other ID'.

Faced with those options, a decision is made to compare signatures by labelling the 'compare signature' operationalisation as satisficed, which in turn will do the same to 'Authenticate user access'.

The *NFR Framework*, rationalizes the development process by providing techniques for justifying design decisions made. These design decisions may positively or negatively affect one or more non-functional requirements, establishing interdependencies that allow inferring the degree to which those non-functional requirements are satisficed or denied (Chung et al., 1999).

Handling non-functional requirements is complex, as it is difficult to define a non-functional term completely unambiguously and hard to explore a complete list of possible solutions and choose the best, or optimal solution (Chung and Leite, 2009). Therefore, the *NFR Framework* takes a more lightweight and qualitative approach towards non-functional requirements, taking a view that softgoals are idealizations and, as such, without all its defining properties necessarily established.

Horkoff (2012) suggests that models focusing on stakeholder goals are particularly suitable for elicitation and analysis in early requirements engineering, as they can show the underlying motivations for systems, capture non-functional success criteria, and show the effects of high-level design alternatives on the attainment of goals. We call this type of model, including agents with interdependent goals, agent-goal models, of which *i** and GRL are some of the most prominent examples.

The *i** framework incorporates concepts from the *NFR framework*, including softgoals, *AND/OR* decompositions, and contribution links, as well as (hard) goals, resources, and dependencies between actors (agents) (Horkoff and Yu, 2013). *i** consists of two main modelling components. The *Strategic Dependency (SD)* model is used to describe the dependency re-

lationships among various actors in an organizational context. The *Strategic Rationale (SR)* model is used to describe stakeholder interests and concerns, and how they might be addressed by various configurations of systems and environments (Yu, 1997).

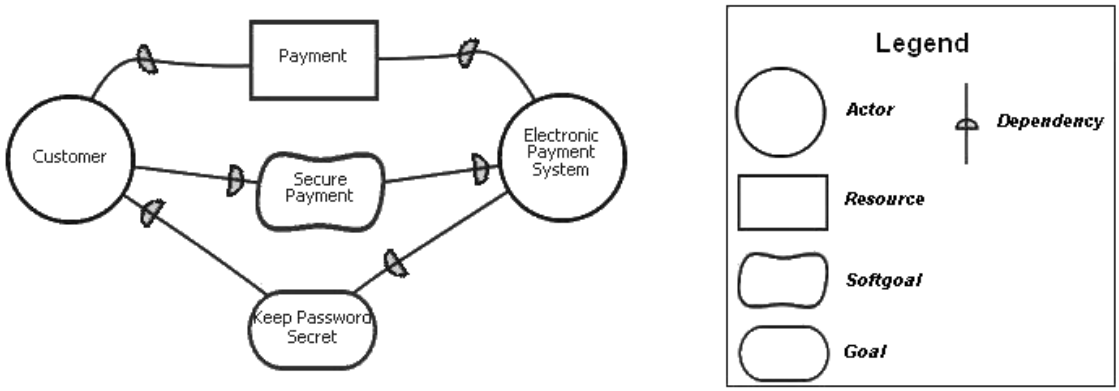

Figure 3.6: Example of an *i** Strategic Dependency (SD) model (Roy, 2007)

Figure 3.6 shows a simple SD model with two actors represented by circles, a Customer and an Electronic Payment System, and their inter-dependencies. In the example, Payment represents a resource to be provided by the Customer, Secure Payment is a softgoal to be achieved by the System, and Keep Password Secret is a goal to be achieved by the Customer (Roy, 2007).

The central concept in *i** is that of the intentional actor. Organizational actors are viewed as having intentional properties such as goals, beliefs, abilities, and commitments. Actors depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. By depending on others, an actor may be able to achieve goals that are difficult or imp ossible to achieve on its own (Yu, 1997).

The concepts of *actors* and *intentional elements* in *i** have influenced other goal-oriented languages, and in particular GRL, a visual modelling notation for intentions, business goals, and non-functional requirements of many stakeholders, for alternatives that have to be considered, for decisions that were made, and for rationales that helped make these decisions (Amyot and Mussbacher, 2011). We describe GRL in depth in the next section.

3.3.1 Goal-oriented Requirements Language (GRL)

The Goal-oriented Requirements Language (GRL) (Amyot et al., 2010), inherits the concept of softgoal from the *NFR Framework*, and adopts non-functionality and related attributes, as first class modelling concepts (Chung and Leite, 2009). GRL captures business or system goals, alternative means of achieving goals (either objectively or subjectively), and rational for contributions and decisions (Amyot, 2003). The language was developed with the following capability targets (Amyot, 2003)

- to enable reasoning about feature interactions and trade-offs early in the design process
- to support the specification, analysis and management of goals and non-functional requirements
- to model the relationship between goals and system requirements
- to capture reusable analysis and design knowledge about non-functional requirements

GRL is part of User Requirements Notation (URN) (ITU-T Z.151), an ITU-T approved standard, with two complementary notations: GRL for modelling actors and their intentions, goals and non-functional requirements and Use Case Maps (UCM) notation for describing scenarios and architectures (Amyot, 2003). ITU-T Z.151 focuses on the definition of an abstract syntax, a concrete graphical syntax, and an interchange format for URN. In this section we will only cover what pertains to GRL as UCM is out of scope.

GRL is a visual modelling notation for intentions, business goals, and non-functional requirements of stakeholders, for alternatives that have to be considered, decisions that were made, and rational that helped make these decisions (Amyot and Mussbacher, 2011).

There are four main categories of concepts in GRL: *actors*, *intentional elements*, *indicators* and *links*. Figure 3.7 shows a summary of the graphical notation in GRL

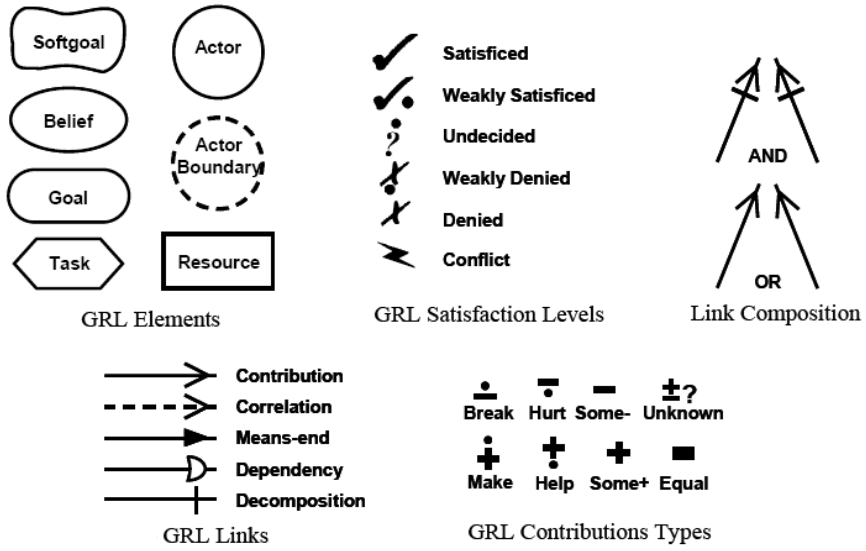

Figure 3.7: Summary of the GRL graphical notation (Roy, 2007)

The intentional elements in GRL are *goals*, *softgoals*, *tasks*, *resources* and *beliefs*. *Intentional* elements are used to present the different alternative behavioural (dynamic) and structural (static) aspects of the system requirements and concentrate on the rational for choosing a particular alternative over the others (Saleh and Al-Zarouni, 2004).

Actors are holders of intentions; they are the active entities in the system or its environment (e.g., stakeholders or other systems) who want goals to be achieved, tasks to be performed, resources to be available and softgoals to be satisfied (ITU-T Z.151). *Indicators* make real-world measurements available for reasoning in the goal model, allowing for a more accurate assessment of the satisfaction of actors (ITU-T Z.151).

Softgoals differentiate themselves from *goals* in that there is no clear, objective measure of satisfaction for a softgoal whereas a goal is quantifiable, often in a binary way. Softgoals are often more related to non-functional requirements, whereas goals are more related to functional requirements (Amyot and Mussbacher, 2011).

Links are used to connect isolated elements in the requirement model. Different types of links depict different structural and intentional relationships (including *decompositions*, *contributions* and *dependencies*) (ITU-T Z.151).

GRL supports the analysis of strategies, which help reach the most appropriate trade-offs among (often conflicting) goals of stakeholders. A *strategy* consists of a set of intentional elements and indicators that are given initial satisfaction values (ITU-T Z.151). It uses qualitative

labels associate to lower-level intentional elements to measure the satisfaction level of higher-level elements. The qualitative satisfaction labels associated to intentional elements goes from *SATISFICED* to *DENIED* (Roy, 2007). These satisfaction values are then propagated to the other intentional elements through their links, enabling a global assessment of the strategy being studied as well as the global satisfaction of the actors involved (ITU-T Z.151).

The GRL notation supports belief elements, which provide justifications of the assessments in the model. Beliefs keep track of the rationales in the graphical models (Roy, 2007).

The URN standard describes the syntax and semantics of the language using an abstract syntax (partially shown in figure 3.8), reserving a concrete grammar, presented as an extension of the abstract grammar, to support a graphical language but which has no implication in the semantics of the language.

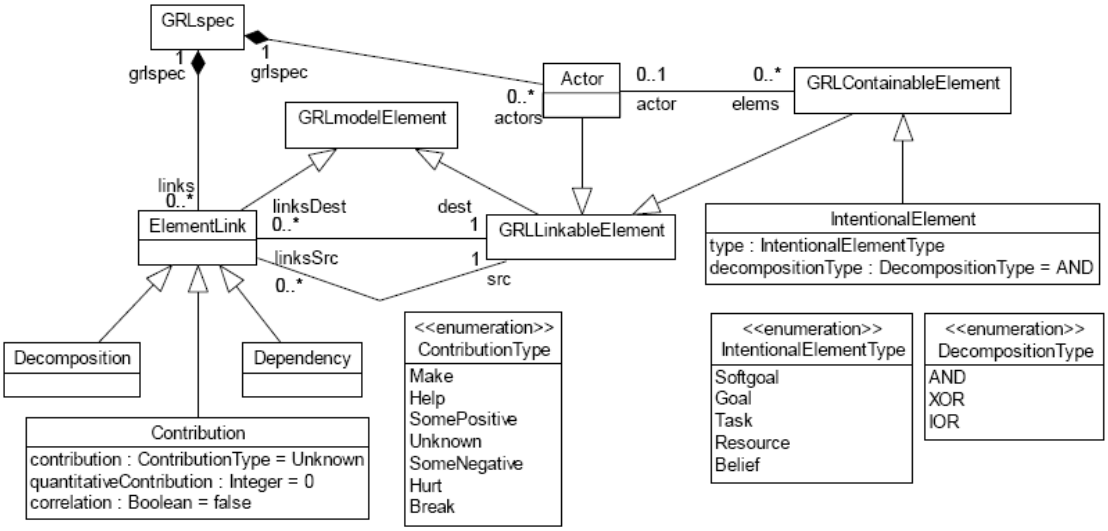

Figure 3.8: GRL Abstract Grammar defined in ITU-T Z.151

A GRL goal graph is a connected graph of intentional elements that optionally reside within an actor. The one shown in figure 3.9 targets the evaluation of an architectural decision about where to put the data and the logic of the authorization service of a wireless system (Amyot and Mussbacher, 2011).

A goal graph, such as the one in figure 3.9, shows the non-functional requirements and business goals of interest to the system and its stakeholders, as well as the alternatives for achieving these high-level elements.

Figure 3.9 shows a particular configuration of alternatives (indicated by a star (*) and a dashed outline), consisting of assigning an initial qualitative satisfaction level of *satisficed* to service control point for data location and mobile switch for service location, and the impact this decision has on the satisfaction levels of stakeholders, namely the system and service provider and their goals. Note that the algorithm used in figure 3.9 combined qualitative and quantitative contributions and the numbers (between -100 and 100) are a quantitative measure of the satisfaction levels of goals and stakeholders. It can be seen that, the strategy chosen resulted in a a satisfaction level for the service provider measured at 39, while the system obtained 100. Accordingly, the goals of the system have been fully satisficed whilst the ones of the service provider have only been partially or in GRL terminology, weakly satisficed. Interestingly, in spite of minimum switch load, a softgoal of the service provider being denied, the impact

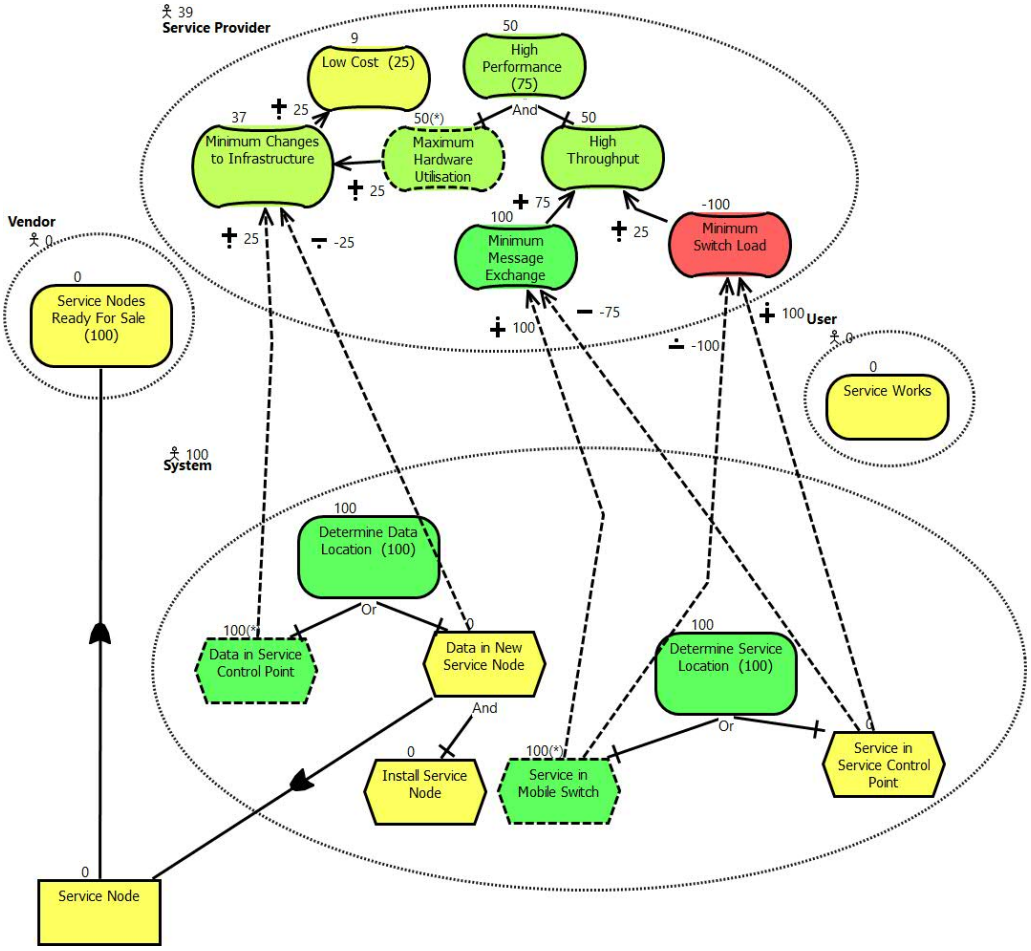

Figure 3.9: GRL Abstract Grammar (Amyot and Mussbacher, 2011)

of that decision on high throughput was low as high throughput had been decomposed in two softgoals and minimum message exchange was satisficed fully.

This evaluation mechanism propagates low-level decisions regarding alternatives to satisfaction ratings of high-level stakeholder goals and softgoals. GRL strategies can be compared with each other to help reach the most appropriate trade-offs among often conflicting goals of stakeholders. Colour coding of the intentional elements also reflect their satisfaction level (the greener, the more satisfied) (Amyot and Mussbacher, 2011).

3.3.2 Summary

In this chapter we have reviewed the notion of non-functional requirements and the multiple classification and representation schemes available. We followed with expanding on the importance of goals and goal-oriented approaches in the elicitation, analysis and modelling of stakeholders requirements, and in particular, quality attributes of the systems and applications that will help stakeholders attain those goals. Finally, we introduced GRL, by describing two other languages from which it spawn from, namely the *NFR Framework* and *i**. While describing GRL, we focused on explaining how it supports reasoning about goals and requirements, while showing the impact of proposed alternative solutions to achieve those goals.

In the next chapter, we will introduce goal-oriented ideas and concepts in behaviour-driven development (BDD) and in doing so, offer a treatment of non-functionality as a first-class concept in BDD.

4 Extending behaviour-driven development

We share the view that goal-oriented analysis complements and strengthens traditional requirements engineering techniques by offering a means for capturing and evaluating alternative ways of meeting business goals (Mylopoulos et al., 2001).

In fact, incorporating non-functional requirements into the different phases of the software life-cycle is a very hard task. Researchers face many challenges including great diversity of non-functional requirements, their subjective nature, formally specifying requirements, incorporating these requirements into models used for specifying functional requirements and resolving conflicts among non-functional requirements (Laleau and Matoussi, 2008).

In this work, we use the approach of Chung and Leite (2009) of considering as non-functional requirements, any '-ilities', '-ities', such as usability or security, along with many other things that do not necessarily end with either of them, such as performance, coherence and user-friendliness, as well as concerns on productivity, time, cost and personal happiness. In view of mathematical functions, in the form of, $function : Input \rightarrow Output$ just about anything that addresses characteristics of a function, a function's input or output or relationships between the two, will be considered as non-functional requirement.

Eliciting high level goals early in the development process is crucial. However, goal-oriented requirements elicitation is an activity that continues as development proceeds, as high-level goals (such as business goals) are refined into lower-level goals (such as technical goals that are eventually operationalised in a system). Eliciting goals focuses the requirements engineer on the problem domain and the needs of the stakeholders, rather than on possible solutions to those problems (Nuseibeh and Easterbrook, 2000). In addition, the requirements engineer needs to explore alternatives and evaluate their feasibility and desirability with respect to business goals.

In extending BDD, our research aimed at integrating a goal-reasoning approach in BDD, while being restricted by constraints imposed by the philosophy and principles of agile development. In other words, we extended BDD by

- taking a lightweight approach
- aiming at minimal changes to Gherkin
- avoiding subverting the spirit behind *Specification by Example*
- respecting the strengths of BDD and GRL and avoiding mixing concepts where no gain was to be made

In what follows, we will be assuming an iterative development approach, even though the description of the steps involved will be sequential. That is the nature of writing. We use a similar approach as Chung et al. (1999) of following a process of developing a target artefact by starting with a source specification and producing constraints upon the target artefact.

This could occur at various phases of development (e.g., requirements specification, conceptual design, or development).

The *NFR Framework* uses non-functional requirements such as security, accuracy, performance and cost to drive the overall design process (Chung et al., 1999). BDD in turn, uses expected behaviours of a system to drive the development process. We assume a hybrid approach, by retaining its behaviour-driven characteristics, but adding constrains provided by non-functional requirements.

Similarly to the *NFR Framework*, there are several major steps in the process:

- acquiring or accessing knowledge the particular domain and the system which is being developed, functional requirements for the particular system, and particular kinds of NFRs, and associated development techniques
- identifying particular NFRs for the domain
- decomposing NFRs,
- identifying 'operationalizations' (possible design alternatives for meeting NFRs in the target system)
- dealing with ambiguities, trade-offs and priorities, and interdependencies among NFRs and operationalizations
- selecting operationalizations
- supporting decisions with design rationale
- evaluating the impact of decisions.
- Compare with Advanced Traceability in (Hull et al., 2011)

5 Conclusion

Most conventional approaches to system design are driven by functional requirements. Developers focus their efforts primarily on achieving the desired functionality of a product or system, usually considering non-functional requirements (such as cost and performance) in a non-systematic and often undocumented way (Chung et al., 1999). Furthermore, these software quality attributes are often addressed late and viewed as consequences of the decisions rather than as goals to be achieved.

Out work helps to address some of the most common requirements risks (Wiegers and Beatty, 2013, p. 20).

- Insufficient user involvement
- Inaccurate planning
- Creeping user requirements
- Ambiguous requirements
- Gold plating
- Overlooked stakeholders

Our work is also actual and contributes to major requirements trends in recent in the past decade, including

- The recognition of business analysis as a professional discipline
- The maturing of tools both for managing requirements in a database and for assisting with requirements development activities such as prototyping, modelling, and simulation
- The increased use of agile development methods and the evolution of techniques for handling requirements on agile projects
- The increased use of visual models to represent requirements knowledge

Writing the requirements isn't the hard part. The hard part is determining the requirements. Writing requirements is a matter of clarifying, elaborating, and recording what you've learned. A solid understanding of a product's requirements ensures that your team works on the right problem and devises the best solution to that problem. Without knowing the requirements, you can't tell when the project is done, determine whether it has met its goals, or make trade-off decisions when scope adjustments are necessary. It can cost far more to correct a defect that's found late in the project than to fix it shortly after its creation. Shortcomings in requirements practices pose many risks to project success, where success means delivering a product that satisfies the user's functional and quality expectations at the agreed-upon cost and schedule (Wiegers and Beatty, 2013).

Some of the most common requirements risks are insufficient user involvement; inaccurate planning; creeping user requirements ; ambiguous requirements and overlooked stakeholders. Sound requirements processes emphasize a collaborative approach to product development that involves stakeholders in a partnership throughout the project. Eliciting requirements lets the development team better understand its user community or market, a critical success factor. Emphasizing user tasks instead of superficially attractive features helps the team avoid writing code that no one will ever execute. Customer involvement reduces the expectation gap between what the customer really needs and what the developer delivers. Documented and clear requirements greatly facilitate system testing. All of these increase your chances of delivering high-quality products that satisfy all stakeholders (Wiegers and Beatty, 2013).

A central point for the distinction of functionality and other qualities is that, for software construction, the purpose of the software system needs to be well defined in terms of the functions that the software will perform (Chung and Leite, 2009).

The distinction between functionality and other qualities in the field of requirements engineering has an important benefit: it makes clear to software engineers that requirements are meant to deal with quality attributes and not with just one of them. As the software industry became more mature and different domains were explored by software engineers, it became clearer that it would not be enough just to deal with the description of the desired functionality, but that quality attributes should be carefully thought of early on as well (Chung and Leite, 2009).

Not only non-functional requirements need to be stated up front, but they can help the software engineer make design decisions, while also justifying such decisions. However, in order to take this into consideration, it is necessary that quality attributes not be considered just as a separate set of requirements, but with the consideration of the functionality throughout the development process (Chung and Leite, 2009).

A Typical chapter contents

FiXme: Remove

Start of chapter

General advice Link back to previous parts in particular previous chapter
State the aim of the chapter
Outline how you intend to achieve this aim in the form of an overview of contents

Contents

Discussion or Analysis

- What's important
- What overall themes can be identified
- What can be observed or learned
- What limitations or shortcomings have been identified
- Situate the chapter within the whole thesis

Summary

- Replies to the introduction by briefly identifying the chapter's achievements and sets the scene for the next chapter

End of chapter

General advice Start with a strong summary of the main findings of this chapter with academic references and relate it with current theory.
Relates this chapter results to earlier analysis.
End with a strong lead into next chapter.

Bibliography

1061:1998, I. (1998). 'IEEE Standard for a Software Quality Metrics Methodology'. In: *ISO/IEC/IEEE 1061:1998*.

29148:2011, I. (2011). 'Systems and software engineering – Life cycle processes – Requirements engineering'. In: *ISO/IEC/IEEE 29148:2011*, pp. 1–94.

Adams, K. (2015). *Non-functional Requirements in Systems Analysis and Design*. Topics in Safety, Risk, Reliability and Quality. Springer International Publishing.

Adzic, G. (2011). *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Pubs Co Series. Manning.

Adzic, G. (2009). *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. United Kingdom: Neuri Limited.

Amyot, D. (2003). 'Introduction to the User Requirements Notation: Learning by Example'. In: *Comput. Netw.* 42.3, pp. 285–301.

Amyot, D. and G. Mussbacher (2011). 'User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper)'. In: *Journal of Software* 6.5, pp. 747–768.

Amyot, D. et al. (2010). 'Evaluating Goal Models Within the Goal-oriented Requirement Language'. In: *Int. J. Intell. Syst.* 25.8, pp. 841–877.

Barmi, Z. A. and A. H. Ebrahimi (2011). 'Automated testing of non-functional requirements based on behavioural scripts'. MA thesis. Chalmers University of Technology.

Becha, H. and D. Amyot (2012). 'Non-Functional Properties in Service Oriented Architecture – A Consumer's Perspective'. In: *Journal of Software* 7.3.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Botella, P. et al. (2001). 'Modeling Non-Functional Requirements'. In: *Proceedings of Jornadas de Ingenieria de Requisitos Aplicada JIRA 2001*.

Bourque, P. and R. E. Fairley (2014). *Guide to the Software Engineering Body of Knowledge. SWEBOK V3.0*. 3rd. IEEE Computer Society.

Brooks Jr., F. P. (1987). 'No Silver Bullet Essence and Accidents of Software Engineering'. In: *Computer* 20.4, pp. 10–19.

Casamayor, A., D. Godoy, and M. Campo (2010). 'Identification of Non-functional Requirements in Textual Specifications: A Semi-supervised Learning Approach'. In: *Information and Software Technology* 52.4, pp. 436–445.

Chung, L. and J. C. P. Leite (2009). 'Conceptual Modeling: Foundations and Applications'. In: ed. by A. T. Borgida et al. Berlin, Heidelberg: Springer-Verlag. Chap. On Non-Functional Requirements in Software Engineering, pp. 363–379.

Chung, L. et al. (1999). 'Non-Functional Requirements in Software Engineering'. In: International Series in Software Engineering.

Cleland-Huang, J. et al. (2007). 'Automated Classification of Non-functional Requirements'. In: *Requirements Engineering* 12.2, pp. 103–120.

Cockburn, A. (2000). *Writing Effective Use Cases*. Crystal series for software development. Pearson Education.

Cockburn, A. (2004). *Crystal Clear a Human-powered Methodology for Small Teams*. First. Addison-Wesley Professional.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. The Addison-Wesley signature series. Addison-Wesley.

Cugola, G. and C. Ghezzi (1998). 'Software processes: a retrospective and a path to the future'. In: *Software Process: Improvement and Practice* 4.3, pp. 101–123.

Davis, A. (2013). *Just Enough Requirements Management: Where Software Development Meets Marketing*. Dorset House eBooks. Pearson Education.

Department of Defense (DoD) (1991). *Software Technology Strategy: Draft*. Director of Defense Research and Engineering.

Evans, E. (2004). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Franch, X. (1998). 'Systematic Formulation of Non-Functional Characteristics of Software'. In: *Proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice*. ICRE '98. Washington, DC, USA: IEEE Computer Society, pp. 174–181.

Glinz, M. (2007). 'On Non-Functional Requirements'. In: *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pp. 21–26.

Glinz, M. (2005). 'Rethinking the Notion of Non-Functional Requirements'. In: *Proceedings of the Third World Congress for Software Quality (3WCSQ'05)*, pp. 55–64.

Grady, R. B. and D. L. Caswell (1987). *Software Metrics: Establishing a Company-wide Program*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Grunske, L. (2008). 'Specification Patterns for Probabilistic Quality Properties'. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, pp. 31–40.

Highsmith III, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York, NY, USA: Dorset House Publishing Co., Inc.

Horkoff, J. M. (2012). 'Iterative, Interactive Analysis of Agent-goal Models for Early Requirements Engineering'. AAINR97565. PhD thesis. Toronto, Ont., Canada, Canada.

Horkoff, J. and E. Yu (2013). 'Comparison and Evaluation of Goal-oriented Satisfaction Analysis Techniques'. In: *Requir. Eng.* 18.3, pp. 199–222.

Hull, E., K. Jackson, and J. Dick (2011). *Requirements Engineering*. Springer Science.

'IEEE Recommended Practice for Software Requirements Specifications' (1998). In: *ISO/IEC/IEEE 830:1998*, pp. 1–40.

Inayat, I. et al. (2015a). 'A Reflection on Agile Requirements Engineering: Solutions Brought and Challenges Posed'. In: *Scientific Workshop Proceedings of the XP2015*. XP '15 workshops. Helsinki, Finland: ACM, 6:1–6:7.

Inayat, I. et al. (2015b). 'A systematic literature review on agile requirements engineering practices and challenges'. In: *Computers in Human Behavior* 51, Part B. Computing for Human Learning, Behaviour and Collaboration in the Social and Mobile Networks Era, pp. 915–929.

ISO/IEC 25010 (2011). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. International Organization For Standardization.

ISO/IEC 9126-1 (2001). *Software engineering – Product quality – Part 1: Quality model*. International Organization For Standardization.

ITU-T (2012). 'Recommendation, Z.151 (12/10), User Requirements Notation (URN)–Language definition'. In: *ITU-T Z.151*.

Jacobson, I., G. Booch, and J. Rumbaugh (1999). *The Unified Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Laleau, R. and A. Matoussi (2008). *A Survey of Non-Functional Requirements in Software Development Process*. Tech. rep. TR-LACL-2008-7. LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12).

Larman, C. (2003). *Agile and Iterative Development: A Manager's Guide*. Pearson Education.

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

Liu, L. and E. Yu (2004). 'Designing information systems in social context: a goal and scenario modelling approach'. In: *Information systems* 29.2, pp. 187–203.

Mylopoulos, J., L. Chung, and E. Yu (1999). 'From Object-oriented to Goal-oriented Requirements Analysis'. In: *Commun. ACM* 42.1, pp. 31–37.

Mylopoulos, J. et al. (2001). 'Exploring Alternatives During Requirements Analysis'. In: *IEEE Softw.* 18.1, pp. 92–96.

North, D. (2006). *Introducing BDD*. URL: `http://dannorth.net/introducing-bdd/` (visited on 07/20/2015).

Nuseibeh, B. and S. Easterbrook (2000). 'Requirements Engineering: A Roadmap'. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, pp. 35–46.

Paetsch, F., A. Eberlein, and F. Maurer (2003). 'Requirements Engineering and Agile Software Development'. In: *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. WETICE '03. Washington, DC, USA: IEEE Computer Society, pp. 308–313.

Palmer, S. R. and M. Felsing (2001). *A Practical Guide to Feature-Driven Development*. 1st. Pearson Education.

Qasaimeh, M., H. Mehrfard, and A. Hamou-Lhadj (2008). 'Comparing Agile Software Processes Based on the Software Development Project Requirements'. In: *Proceedings of the 2008 International Conference on Computational Intelligence for Modelling Control & Automation*. CIMCA '08. Washington, DC, USA: IEEE Computer Society, pp. 49–54.

Al-Qutaish, R. (2009). 'An Investigation of the Weaknesses of the ISO 9126 International Standard'. In: *Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on*. Vol. 1, pp. 275–279.

Robertson, S. and J. Robertson (1999). *Mastering the Requirements Process*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Roman, G.-.-C. (1985). 'A Taxonomy of Current Issues in Requirements Engineering'. In: *Computer* 18.4, pp. 14–23.

Roy, J.-F. (2007). 'Requirement engineering with URN: Integrating goals and scenarios'. MA thesis. Ottawa-Carleton Institute for Computer Science.

Royce, W. W. (1970). 'Managing the development of large software systems'. In: *proceedings of IEEE WESCON*. Vol. 26. 8. Los Angeles, pp. 328–388.

Saleh, K. and A. Al-Zarouni (2004). 'Capturing Non-Functional Software Requirements Using the User Requirements Notation'. In: *2004 International Research Conference on Innovation in Information Technology (IIT'04), Dubai*, pp. 222–230.

Schwaber, K. and M. Beedle (2001). *Agile Software Development with Scrum*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Smart, J. F. (2014). *BDD in Action: Behavior-driven development for the whole software life-cycle*. 1st ed. Manning Publications.

Solis, C. and X. Wang (2011). 'A Study of the Characteristics of Behaviour Driven Development'. In: *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. SEAA '11. Washington, DC, USA: IEEE Computer Society, pp. 383–387.

Sommerville, I. and G. Kotonya (1998). *Requirements Engineering: Processes and Techniques*. New York, NY, USA: John Wiley & Sons, Inc.

Sommerville, I. and P. Sawyer (1997). *Requirements Engineering: A Good Practice Guide*. 1st. New York, NY, USA: John Wiley & Sons, Inc.

Van Lamsweerde, A. (2001). 'Goal-Oriented Requirements Engineering: A Guided Tour'. In: *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. RE '01. Washington, DC, USA: IEEE Computer Society, pp. 249–262.

Wiegers, K. and J. Beatty (2013). *Software Requirements*. 3rd. Developer Best Practices. Microsoft Press.

Wynne, M. and A. Hellesoy (2012). *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. The pragmatic programmers. Pragmatic Bookshelf.

Yu, E. (1997). 'Towards modelling and reasoning support for early-phase requirements engineering'. In: *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pp. 226–235.

Zave, P. (1997). 'Classification of Research Efforts in Requirements Engineering'. In: *ACM Comput. Surv.* 29.4, pp. 315–321.