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) (North, 2006), 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 (Chung et al., 1999) – a structured approach to represent and reason about non-functional requirements – and the goal-oriented requirements language (GRL) (Amyot et al., 2010) 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

1 Introduction **2**
- 1.1 Motivation . . . 2
- 1.2 Aim and limitations of study . . . 4
- 1.3 Significance of the study . . . 4
- 1.4 Overview of Contents . . . 5

2 From requirements to goals **6**
- 2.1 Requirements . . . 6
  - 2.1.1 Definition . . . 6
  - 2.1.2 Classification . . . 7
- 2.2 Requirements Engineering . . . 10
  - 2.2.1 Definition . . . 11
  - 2.2.2 Processes . . . 12
- 2.3 Agile Requirements Engineering . . . 15
  - 2.3.1 Agile development . . . 15
  - 2.3.2 Practices . . . 15
  - 2.3.3 Behaviour driven development (BDD) . . . 15

3 On non-functional requirements **16**
- 3.1 Product or process oriented approaches . . . 16
- 3.2 Requirements as Goals . . . 16
- 3.3 Goal oriented requirements language (GRL) . . . 17

4 Extending behaviour-driven development **18**

5 Conclusion **19**

A Typical chapter contents **21**

List of Figures

2.1 Types of Requirements 10
2.2 Topics for software requirements 11
2.3 Requirements engineering disciplines 12
2.4 Testing activities and requirements 14
2.5 BDD: From business goals to executable specifications 15

List of Tables

2.1 Types of requirements information 8

1 Introduction

This thesis presents an extension to Behaviour Driven Development (BDD) (North, 2006) 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 *'Rethinking the Notion of Non-Functional Requirements'*, 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 Prado 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 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) (North, 2006). 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 taking business goals into 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 however, 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) observe 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 perform 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 to understand the remainder of the thesis and includes: section **??** on requirements engineering and in particular, the approaches taken by agile processes; section 2.3.3 on behaviour-driven development, describing the principles and practices of this popular agile process; section 3 presenting an overview of the research concerning ways to handle non-functional requirements in software engineering and section **??** 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 requirements to goals

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 (BDD) in particular.

In this chapter, we'll reflect on how fast-changing technology and increased competition are placing ever increasing pressure on the development process. We will first review the notions of requirements and requirements engineering, highlighting the most used processes and activities, regardless of the software development method in use.

FiXme: Complete intro

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 aren't proficient at 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) mentions 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 the 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 provided below 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(E)

> 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 requirement (*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. Another definition comes from *Requirements Engineering: A Good Practice Guide* (1997):

> 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 developer 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 operators (Wiegers and Beatty, 2013).

2.1.2 Classification

Wiegers and Beatty (2013) provides 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 provide 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 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. We like to record the business requirements in a vision and scope document. Other strategic guiding documents sometimes used for

Table 2.1: Types of requirements information (extracted from (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

this purpose include a project charter, business case, and market (or marketing) requirements document.

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. Similarly, the system requirements can form an excellent technical summary of a development project (Hull et al., 2011).

Functional requirements specify the behaviours the product will exhibit under specific con-

ditions. 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.

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. In the figure, solid arrows mean 'are stored in'; dotted arrows mean 'are the origin of' or 'influence.'

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, recognize that these three deliverables contain different information, developed at different points in the project, possibly by different people, 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.

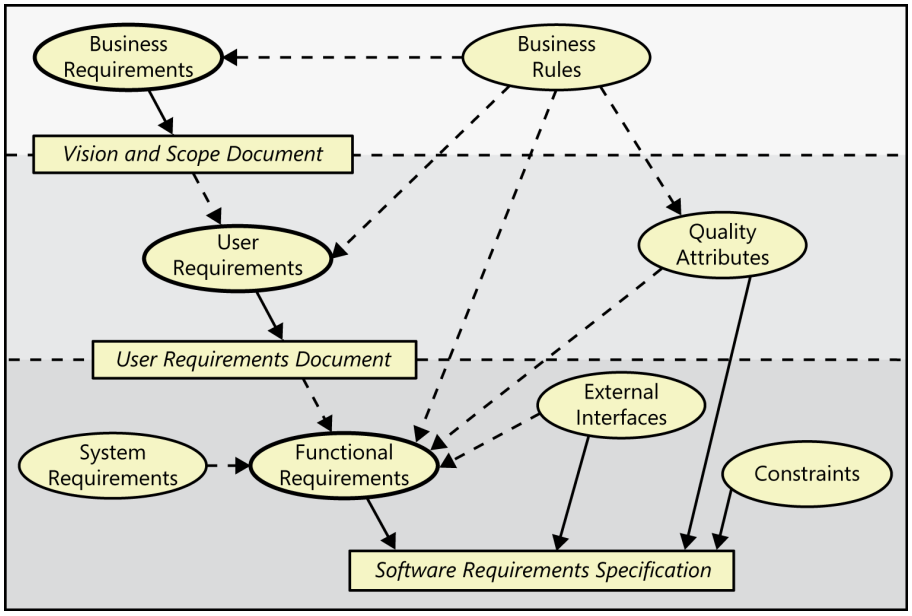

Figure 2.1: Relationships among several types of requirements information (Wiegers and Beatty, 2013, p. 8)

Hull et al. (2011) makes an important distinction between requirements being defined in either *problem* or *solution* domains. 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 stakeholders, analysts and developers views 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 any particular design. Finally, subsystem and component requirements, part of architectural designs, will specify how these 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).

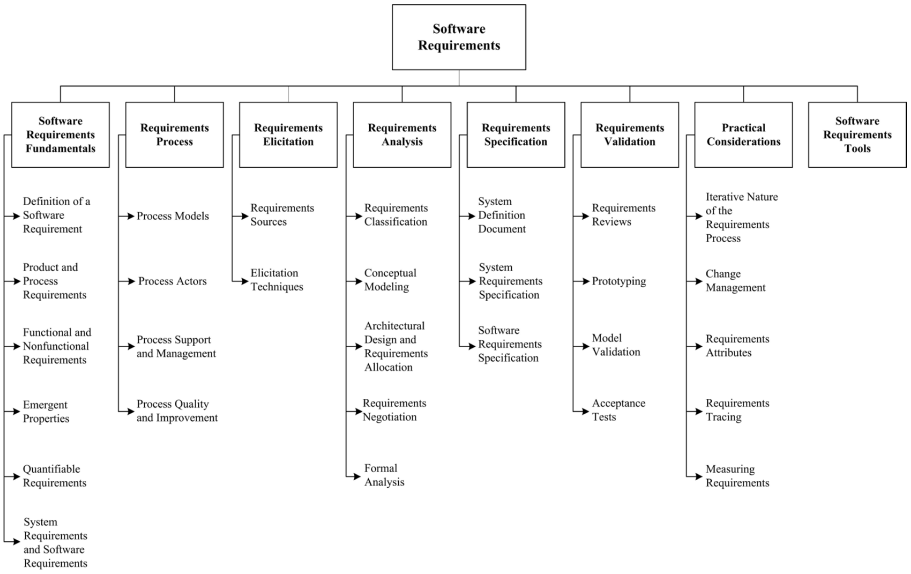

Figure 2.2: Topics for the Software Requirements knowledge area (*SWEBOK V3.0*, 2014)

2.2.1 Definition

The definition in ISO/IEC/IEEE 29148:2011(E) 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 (Director of Defense Research and Engineering, 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 discovering, developing, tracing, analysing, qualifying, communicating and managing requirements that define the system at successive levels of abstraction (Hull et al., 2011)

They defend the above definition reflects better that requirements exist at multiple levels of development and also list key activities that are considered proper to requirements engineering. Similar to what we have done for the definition of requirement, it is worth breaking it down into its constituent parts. *Discovering* refers to activities related to the elicitation and capture of requirements; *tracing* allows setting up links to and from requirements to other artefacts; *qualifying* 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; *communicating* reflects requirements as means of communication, through which all stakeholders can agree on what is to be achieved. Finally, *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 – 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

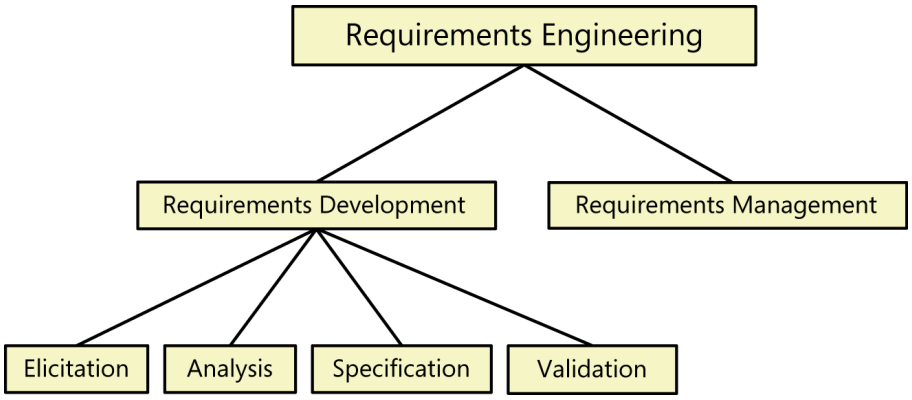

Figure 2.3: Requirements engineering disciplines (Wiegers and Beatty, 2013, p. 15)

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 depth or 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 ('Systems and software engineering - Life cycle processes - Requirements engineering' 2011). It typically takes 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).

ISO/IEC/IEEE 29148:2011(E) defines *requirements analysis* as a process that traansforms 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. Through this process high-level requirements are broken down into an appropriate level of detail, the relative importance of quality attributes is assessed and requirements are allocated to software components defined in the system architecture (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 diagrams suitable for comprehension, review, and use by their intended audiences (Wiegers and Beatty, 2013).

ISO/IEC/IEEE 29148:2011(E) 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 requirements 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) alludes 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 percent 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(E) refers to as *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(E) 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.

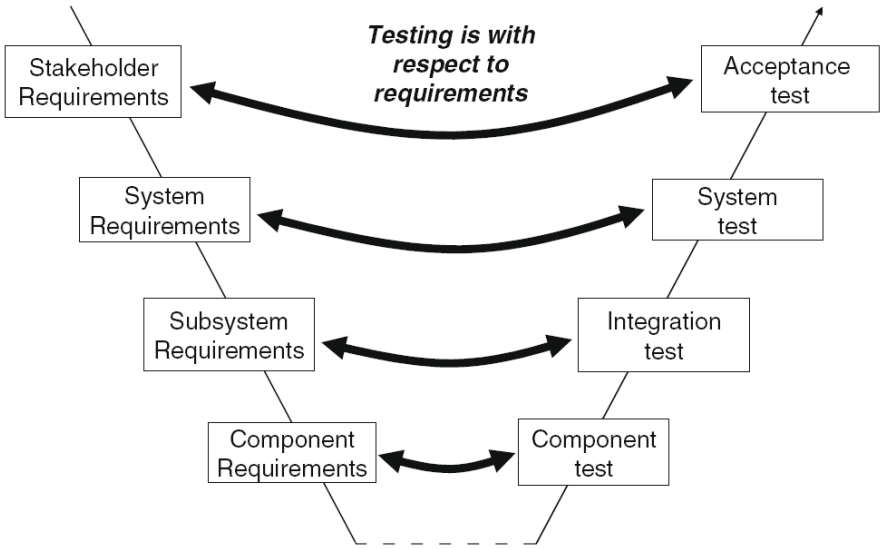

Figure 2.4: Testing activities and requirements

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 the very real changes that you can always expect 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. For example, in a business context the concern is how a business vision determines a set of objectives and how these may be implemented as changes to the organisation and processes. In a systems development context, the focus is in determining how stakeholder requirements are met by system requirements and their breakdown in subsystems and components. 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

2.3.1 Agile development

Traditional software development methods advocate a simple top-down flow of requirements information. In agile development approaches we expect cycles and iteration among the business, user, and functional requirements. An effective change process that includes impact analysis ensures that the right people make informed business decisions about which changes to accept and that the associated costs in time, resources, or feature trade-offs are addressed (Wiegers and Beatty, 2013).

Often, it's impossible or unnecessary to fully specify the functional requirements before commencing design and implementation. In those cases, you can take an iterative or incremental approach, implementing one portion of the requirements at a time and obtaining customer feedback before moving on to the next cycle. This is the essence of agile development, learning just enough about requirements to do thoughtful prioritization and release planning so the team can begin delivering valuable software as quickly as possible. This isn't an excuse to write code before contemplating requirements for that next increment, though. Iterating on code is more expensive than iterating on concepts (Wiegers and Beatty, 2013).

2.3.2 Practices

In agile approaches 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, safety or performance Paetsch et al., 2003.

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 this latter use that is at the core of Behaviour Driven Development (BDD).

North (2006) first described BDD in a blog post

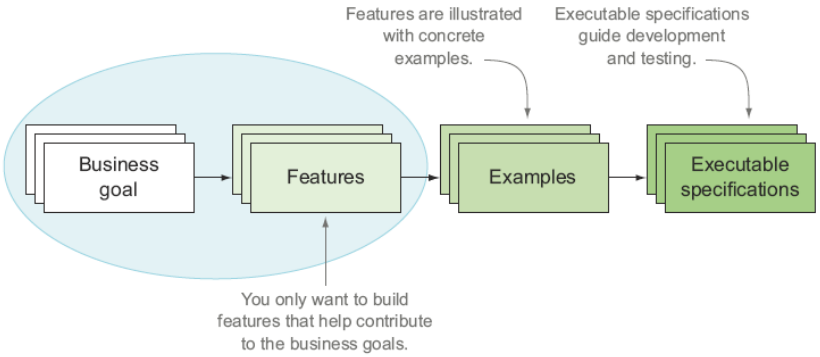

Figure 2.5: BDD: From business goals to executable specifications

3 On non-functional requirements

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'. That adjective says what the requirements are not, but it doesn't say what they are. We are sympathetic to the problem, but we lack a perfect solution (Wiegers and Beatty, 2013).

Other-than-functional requirements might specify not what the system does, but rather how well it does those things. They could describe important characteristics or properties of the system. These include the system's availability, usability, security, performance, and many other characteristics. Some people consider non-functional requirements to be synonymous with quality attributes, but that is overly restrictive. For example, design and implementation constraints are also non-functional requirements, as are external interface requirements (Wiegers and Beatty, 2013).

Still other non-functional requirements address the environment in which the system operates, such as platform, portability, compatibility, and constraints. Many products are also affected by compliance, regulatory, and certification requirements. There could be localization requirements for products that must take into account the cultures, languages, laws, currencies, terminology, spelling, and other characteristics of users. Though such requirements are specified in non-functional terms, the business analyst typically will derive numerous bits of functionality to ensure that the system possesses all the desired behaviours and properties (Wiegers and Beatty, 2013).

In this thesis, rather than worry about precisely what to call these sorts of information, best practice is to ensure that they are part of requirements elicitation and analysis activities. A product can be delivered with all the desired functionality but that users hate because it doesn't match their (often unstated) quality expectations (Wiegers and Beatty, 2013).

3.1 Product or process oriented approaches

3.2 Requirements as Goals

The use of goals in requirements engine 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).

In addition, the requirements engineer needs to explore alternatives and evaluate their feasibility and desirability with respect to business goals. 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).

3.3 Goal oriented requirements language (GRL)

Goals denote the objectives a system must meet. Eliciting high level goals early in the development process is crucial. However, goal-oriented requirements elicitation [15] 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).

A goal is an objective the system under consideration should achieve. Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve (Van Lamsweerde, 2001). Goals also cover different types of concerns: 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.

4 Extending behaviour-driven development

- Compare with Advanced Traceability in (Hull et al., 2011)

5 Conclusion

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 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

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

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'. Chalmers University of Technology.

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.

Chung, L. and J. C. Prado 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.

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

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.

Director of Defense Research, U. S. O. of the and Engineering (1991). *Department of Defense Software Technology Strategy: Draft*. U.S. Department of Defense.

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

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: *in Proceedings of the Third World Congress for Software Quality (3WCSQ'05)*, pp. 55–64.

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.

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

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–.

Smart, J. F. (2014). *BDD in Action: Behavior-driven development for the whole software lifecycle*. 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 P. Sawyer (1997). *Requirements Engineering: A Good Practice Guide*. 1st. New York, NY, USA: John Wiley & Sons, Inc.

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

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.

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