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 that 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 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 and my children Elijah and Indigo, for their endless patience whenever I so often disappeared into the office to research and write this thesis.

My sincere thanks to my mother who always encouraged and supported me through my studies and, more importantly, through all my life.

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 Source Code / Feature

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 that 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.

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

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 requirements 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 such agile approach 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 is deriving from business goals, a sufficient 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 others. 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 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 justifies 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 into 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 goal 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 restrict the validity of our results to any particular taxonomy, as our approach is independent of the non-functional requirements 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;
- Allows BDD to consider all non-functional requirements relevant for a successful product delivery,not just those that are technical;
- 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 bemore suitable for discussion and fostering communication;
- Brings the benefits of executable specifications in BDD to goal formulations.

1.4 Overview of contents

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, an in-depth analysis of 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 also a 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 and chapter 5 are the core of the thesis and contain 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 containing GRL elements and links.

Chapter 4 focuses more on the practical uses of the extended methodology, while chapter 5 exposes the technical details of the changes made to Gherkin and explains in detail the implementation of the translator from Gherkin to GRL.

Chapter 6 contains implications of findings, concluding thoughts and related work, 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’.

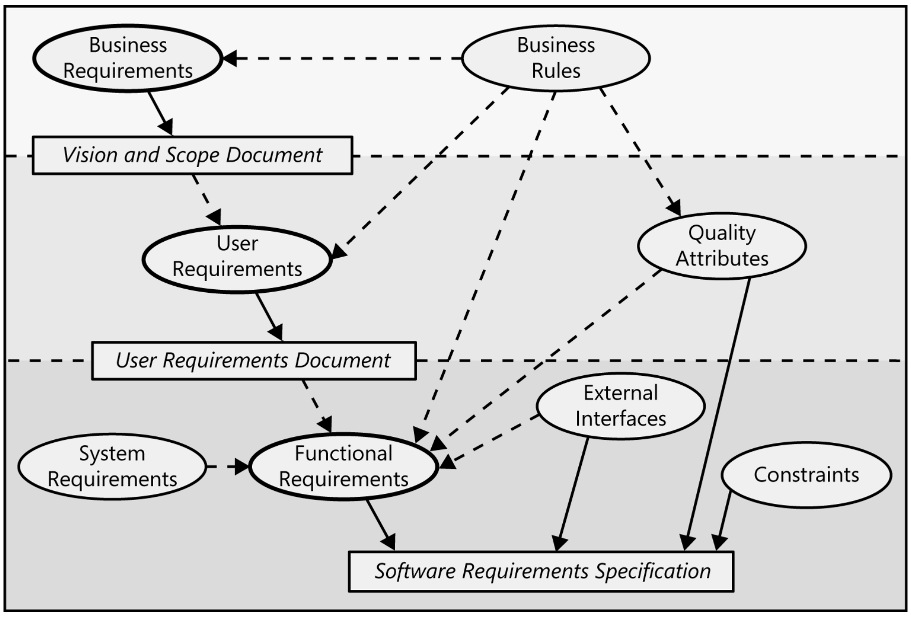


Figure 2.1: Relationships among several types of requirements information (Wiegers and Beatty, 2013, p. 8)

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).

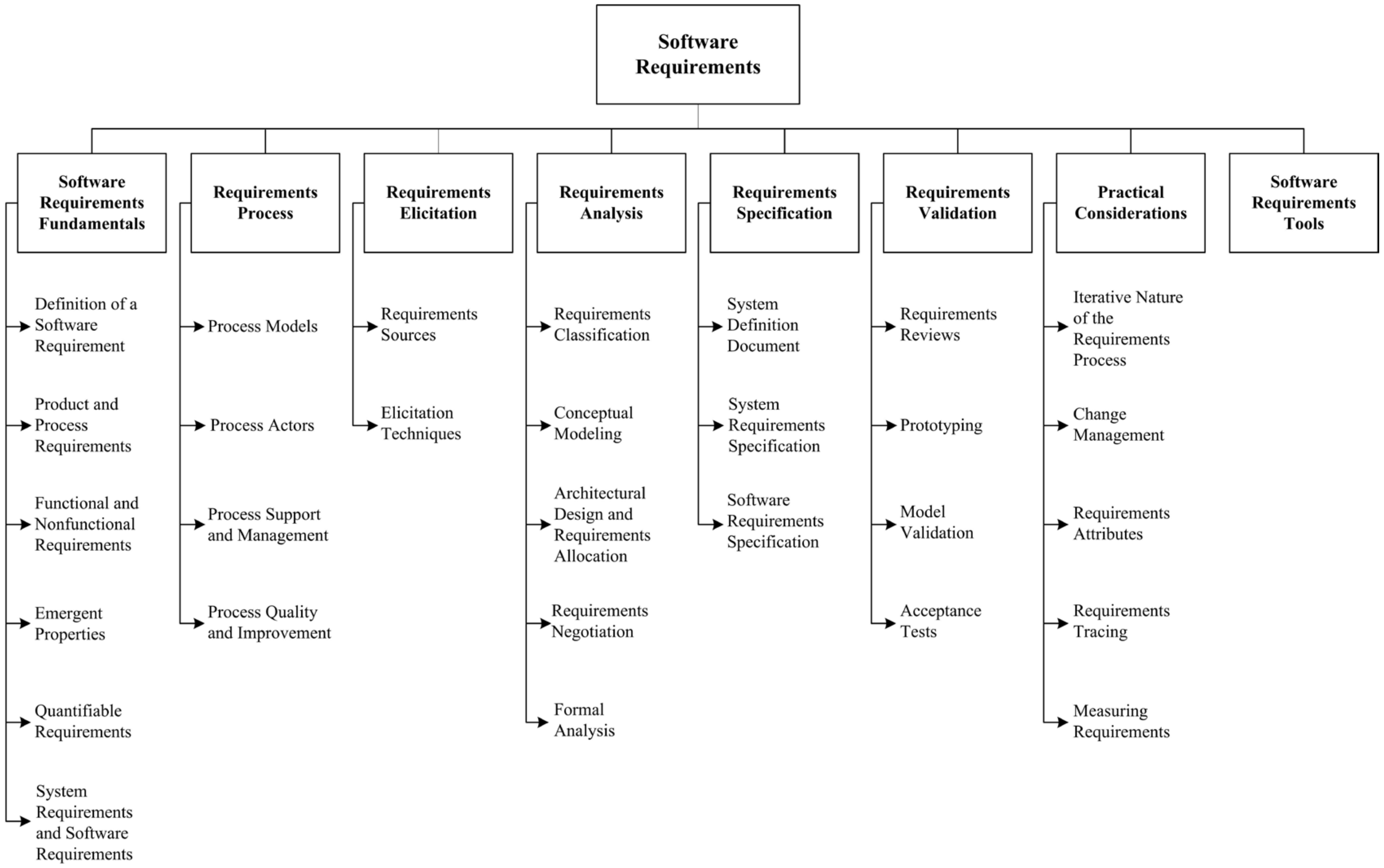

Software Requirements
Software Requirements Fundamentals
Definition of a Software Requirement
Product and Process Requirements
Functional and Nonfunctional Requirements
Emergent Properties
Quantifiable Requirements
System Requirements and Software Requirements
Requirements Process
Process Models
Process Actors
Process Support and Management
Process Quality and Improvement
Requirements Elicitation
Requirements Sources
Elicitation Techniques
Requirements Analysis
Requirements Classification
Conceptual Modeling
Architectural Design and Requirements Allocation
Requirements Negotiation
Formal Analysis
Requirements Specification
System Definition Document
System Requirements Specification
Software Requirements Specification
Requirements Validation
Requirements Reviews
Prototyping
Model Validation
Acceptance Tests
Practical Considerations
Iterative Nature of the Requirements Process
Change Management
Requirements Attributes
Requirements Tracing
Measuring Requirements
Software Requirements Tools

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 *'inter-disciplinary 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 (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

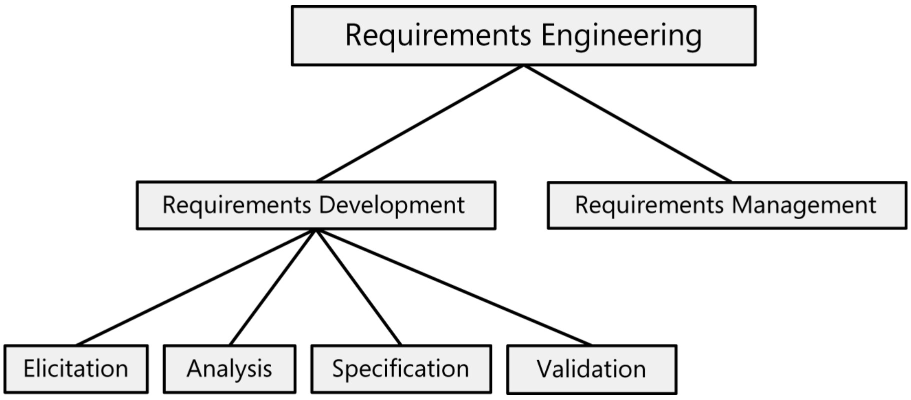


Figure 2.3: Requirements engineering disciplines (Wiegers and Beatty, 2013, p. 15)

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 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) 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.

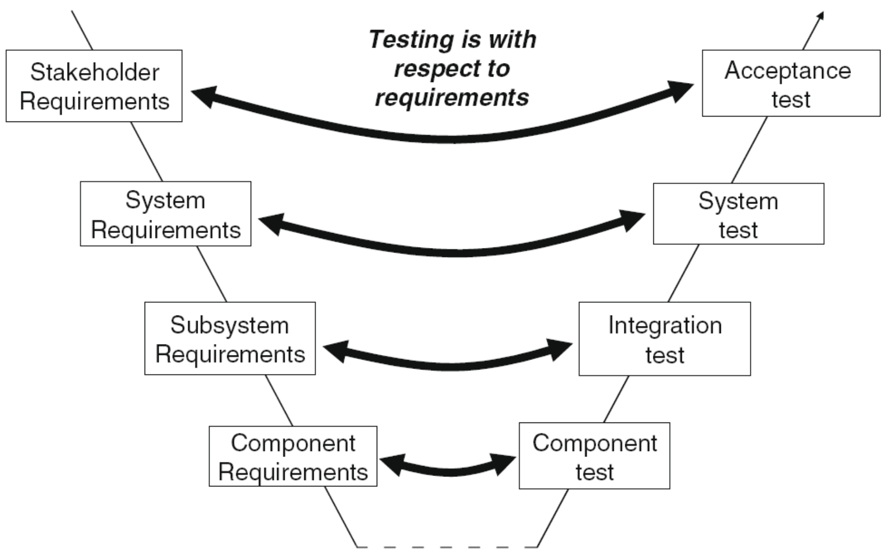


Figure 2.4: Testing activities and requirements (Hull et al., 2011)

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).

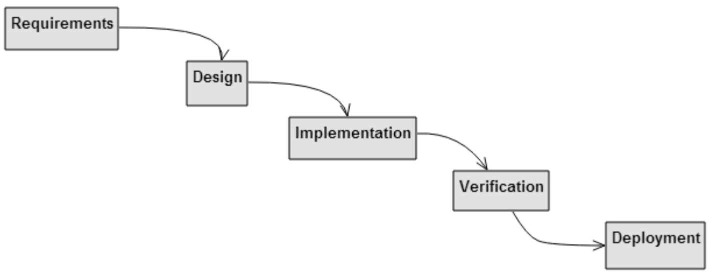


Figure 2.5: Sequential stages in waterfall development (Leffingwell, 2011)

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 development methods (Leffingwell, 2011; Larman, 2003). Instead, we describe an agile requirements artefact model and corresponding agile practices and principles, that compose the

[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

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

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

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

[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

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 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

[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).

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

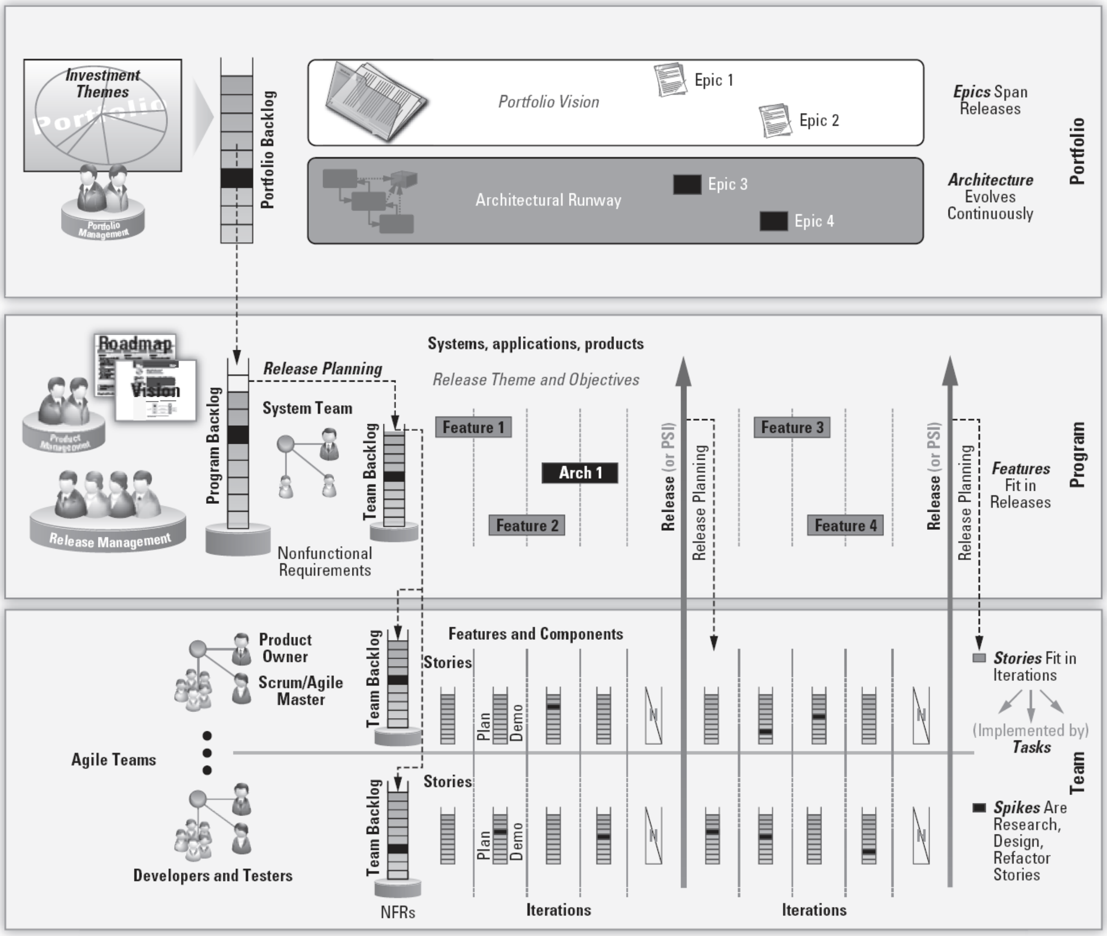


Figure 2.6: Agile development levels: *Team*, *Program* and *Portfolio* (Leffingwell, 2011)

> ... 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 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

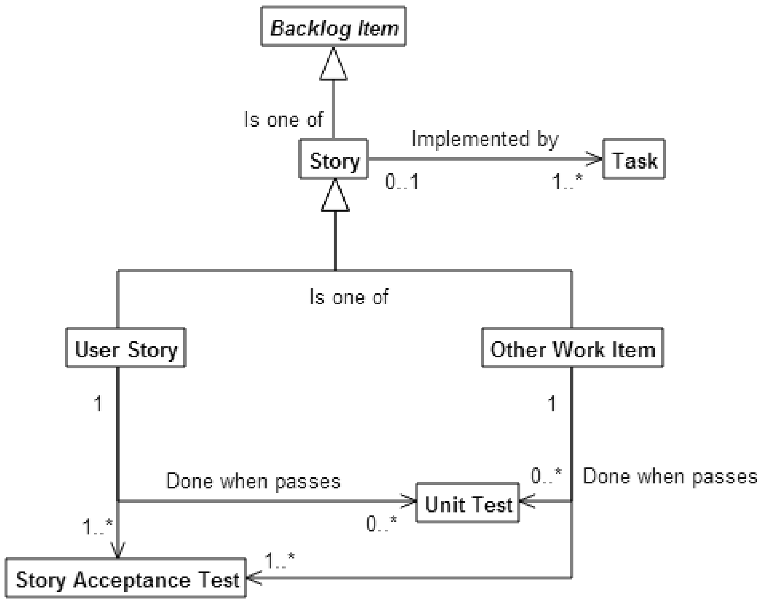


Figure 2.7: Agile requirements meta-model for the *Team*

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.

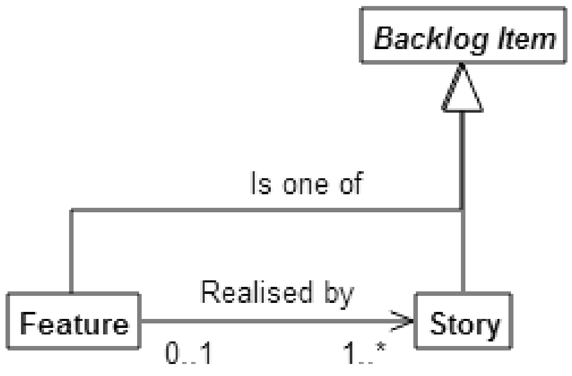


Figure 2.8: Agile requirements meta-model (*Features*)

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*.

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

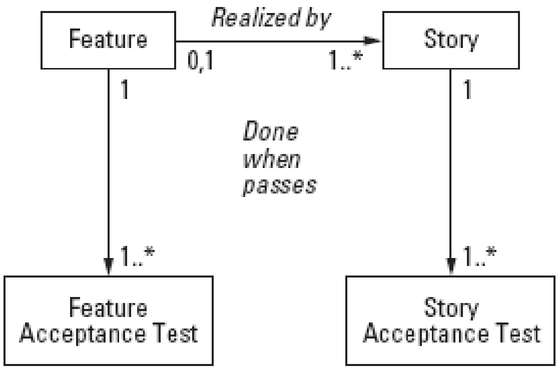


Figure 2.9: Agile requirements meta-model (*Features* and *Acceptance Tests*)

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).

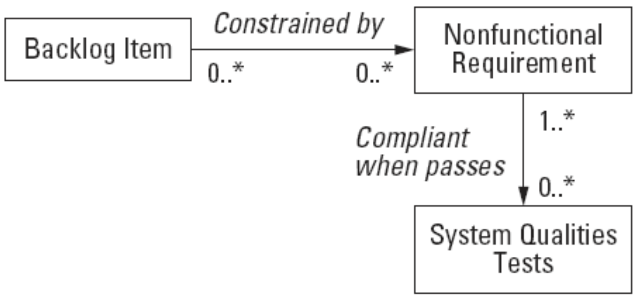


Figure 2.10: Agile requirements meta-model (*Non-functional requirements*)

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).

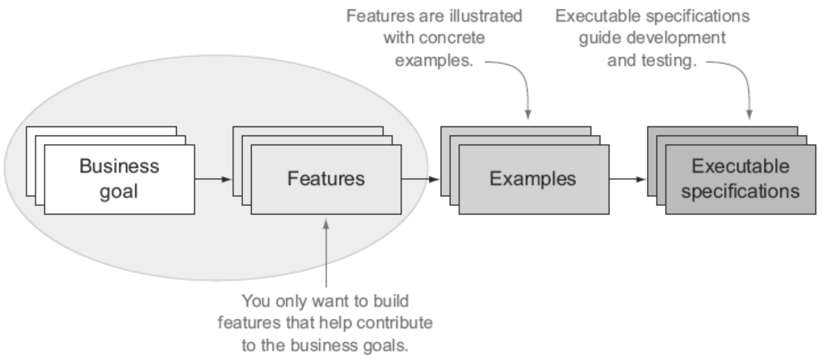

Features are illustrated with concrete examples.
Executable specifications guide development and testing.
Business goal
Features
Examples
Executable specifications
You only want to build features that help contribute to the business goals.

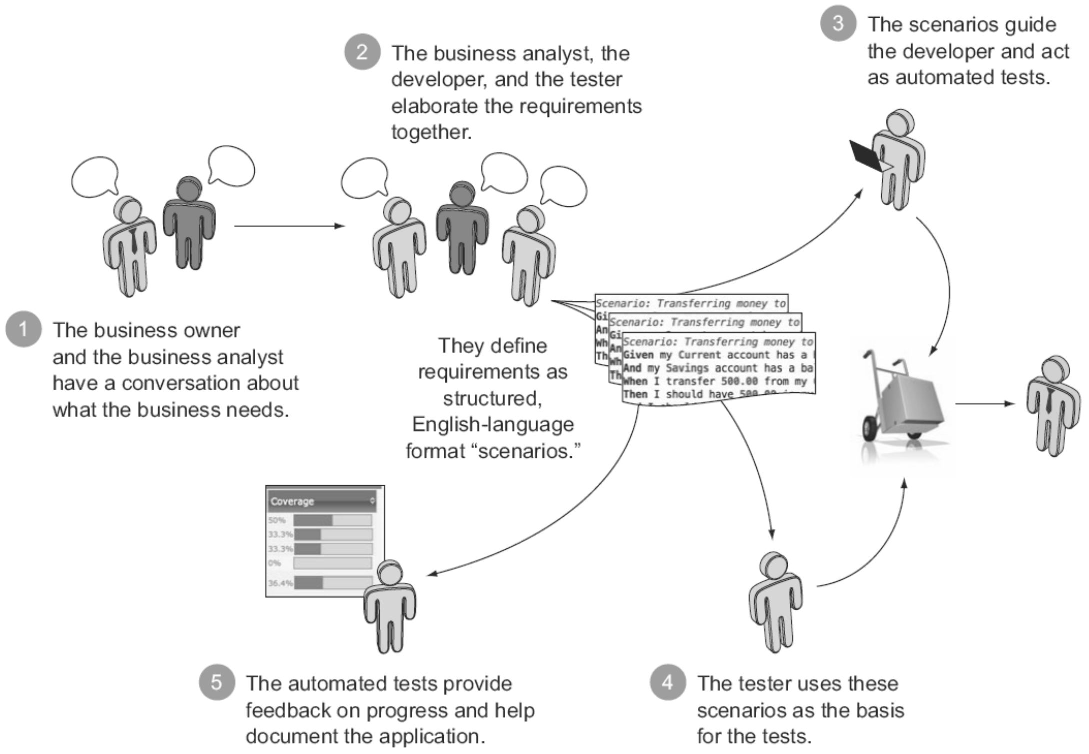
Figure 2.11: BDD: From business goals to executable specifications (Smart, 2014)

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 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


1 The business owner and the business analyst have a conversation about what the business needs.
2 The business analyst, the developer, and the tester elaborate the requirements together.
They define requirements as structured, English-language format "scenarios."
3 The scenarios guide the developer and act as automated tests.
4 The tester uses these scenarios as the basis for the tests.
5 The automated tests provide feedback on progress and help document the application.
Coverage
50%
33.3%
33.3%
0%
36.4%

```
Scenario: Transferring money to
Given my Current account has a
And my Savings account has a ba
When I transfer 500.00 from my
Then I should have 500.00
```

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

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).

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.

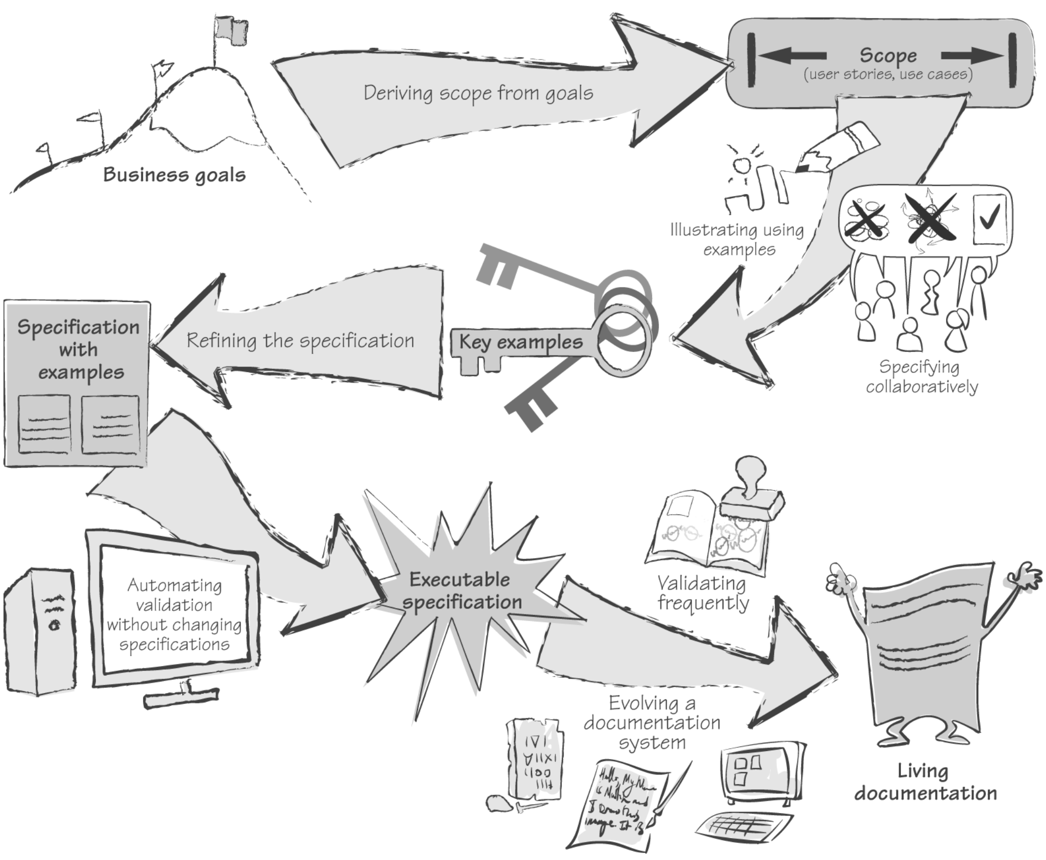

Business goals
Deriving scope from goals
Scope
(user stories, use cases)
Illustrating using examples
Specifying collaboratively
Key examples
Refining the specification
Specification with examples
Automating validation without changing specifications
Executable specification
Validating frequently
Evolving a documentation system
Living documentation

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

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).

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*

```
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
```

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

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-functional requirements 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-functional characteristics are taken into consideration, they 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 view as non-functional characteristics (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

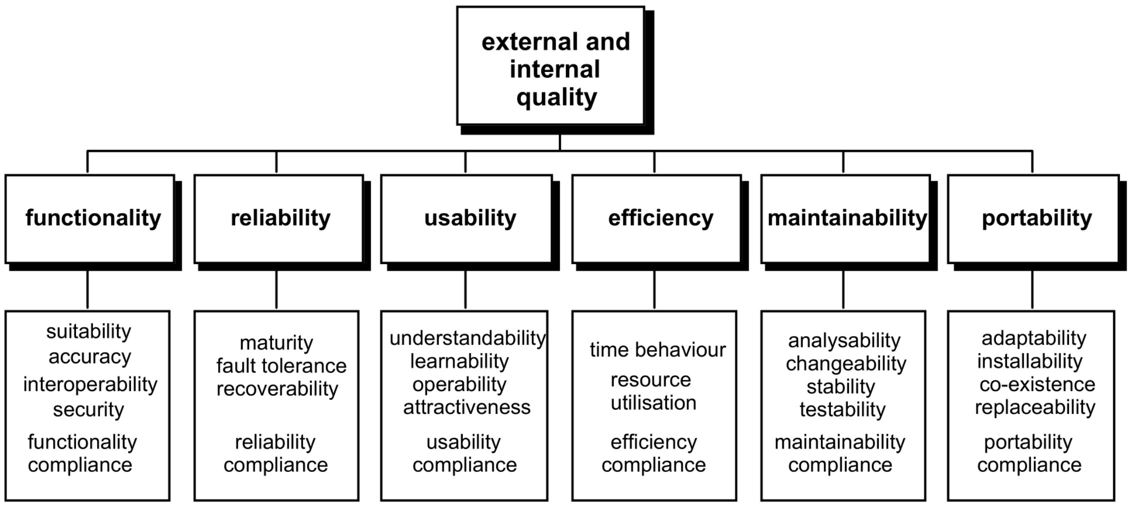


Figure 3.1: Software product quality model in ISO/IEC 9126-1:2001

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).

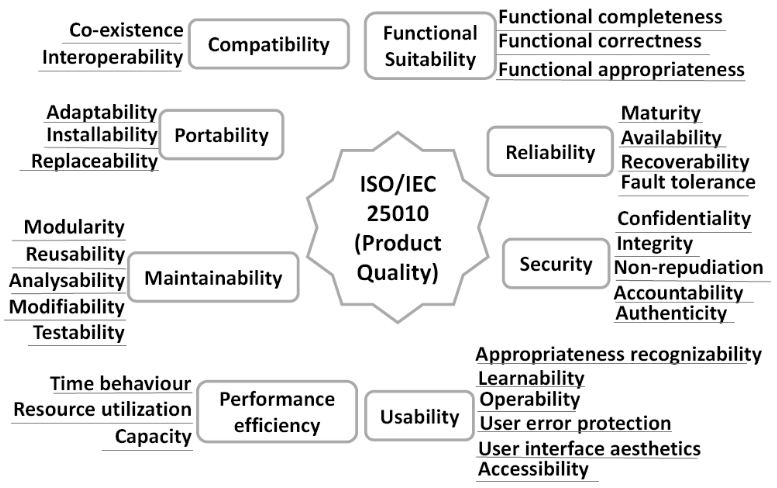


Figure 3.2: ISO 25010 quality model in ISO/IEC 25010:2011

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

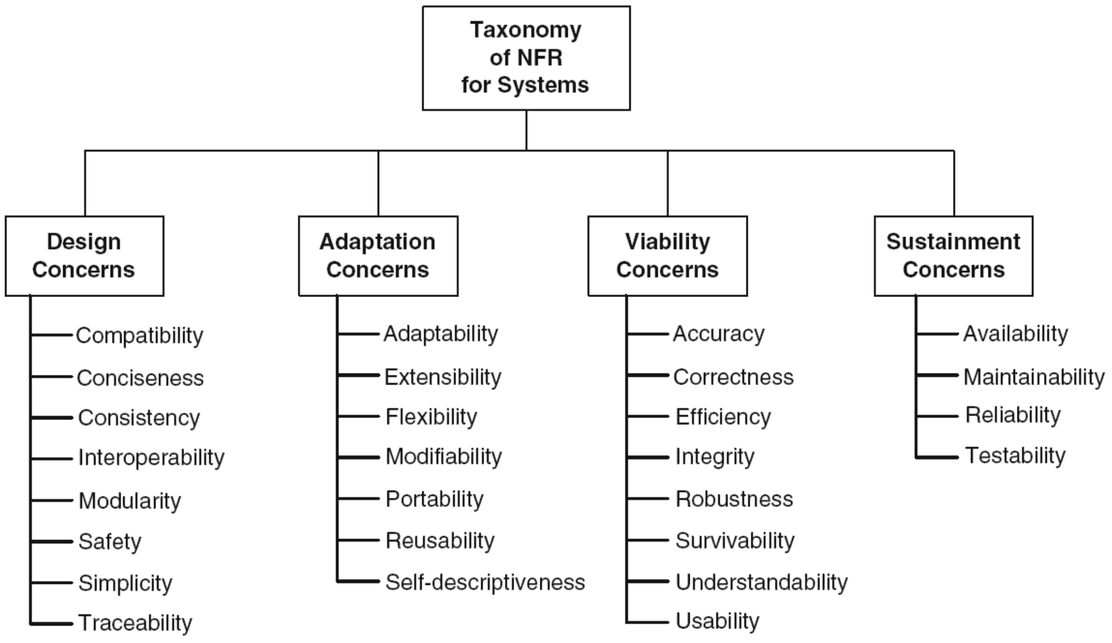


Figure 3.3: Taxonomy of non-functional requirements for systems (Adams, 2015)

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 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 at a higher 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

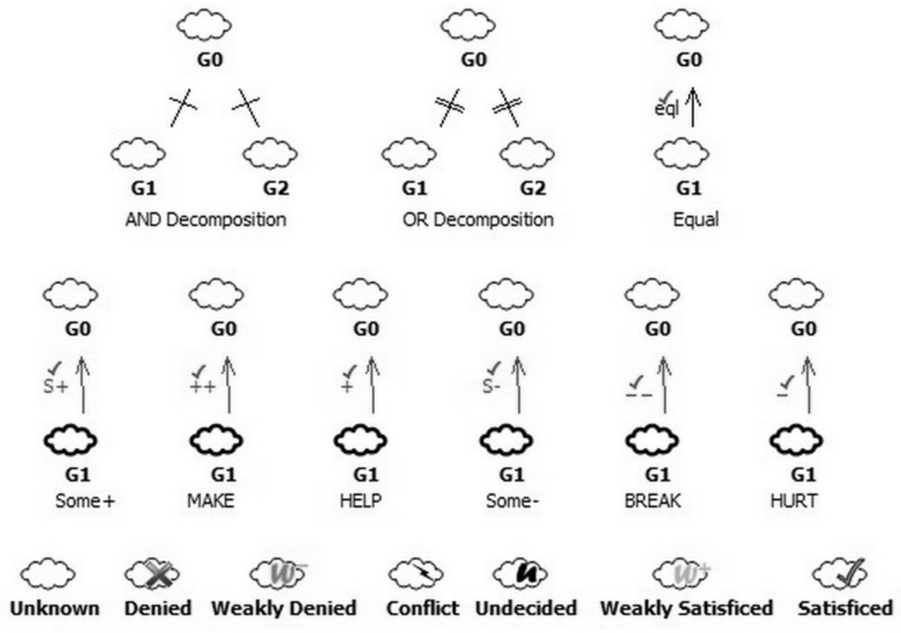


Figure 3.4: Catalogue of visual elements in NFR Framework [2]

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/

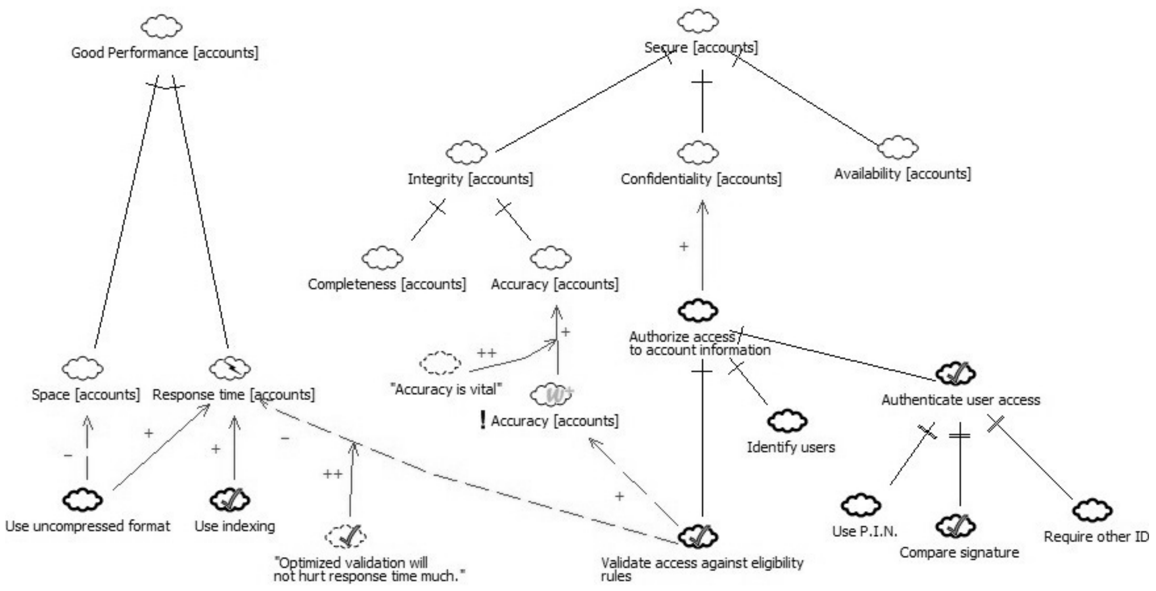


Figure 3.5: Example of a Softgoal Interdependency Graph (SIG) [3]

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 relationships 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).

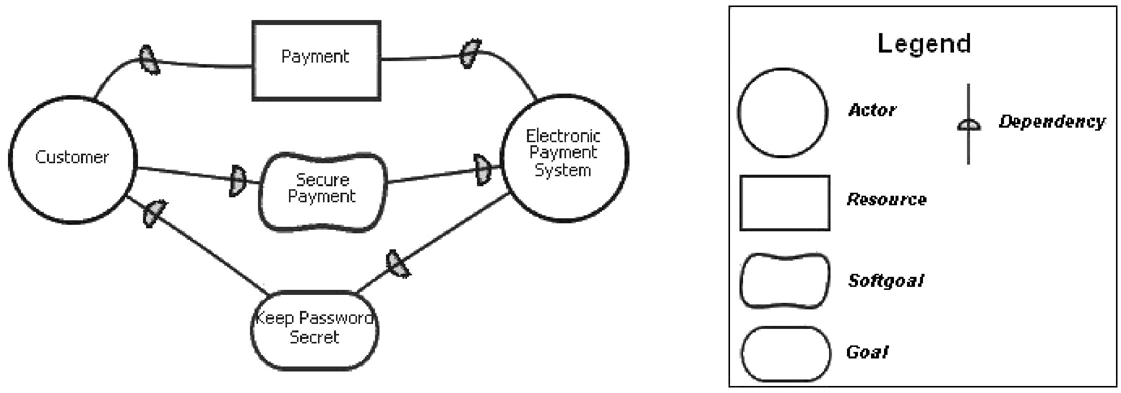


Figure 3.6: Example of an *i** Strategic Dependency (SD) model (Roy, 2007)

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-functional characteristics 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

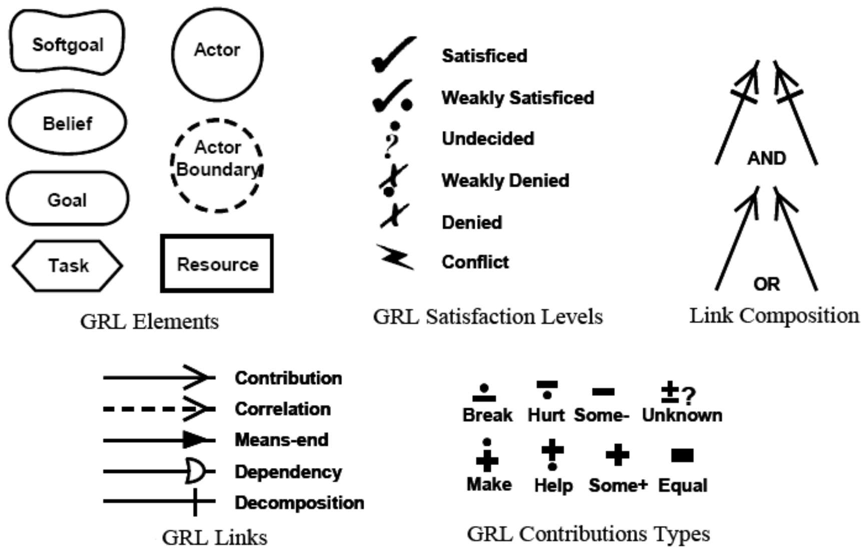


Figure 3.7: Summary of the GRL graphical notation (Roy, 2007)

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.

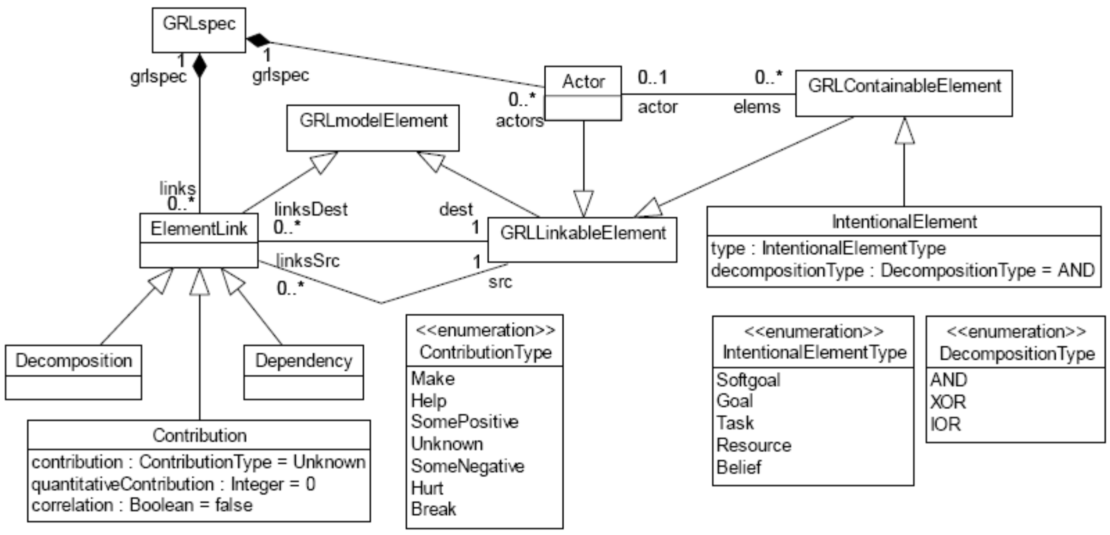


Figure 3.8: GRL Abstract Grammar defined in ITU-T Z.151

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

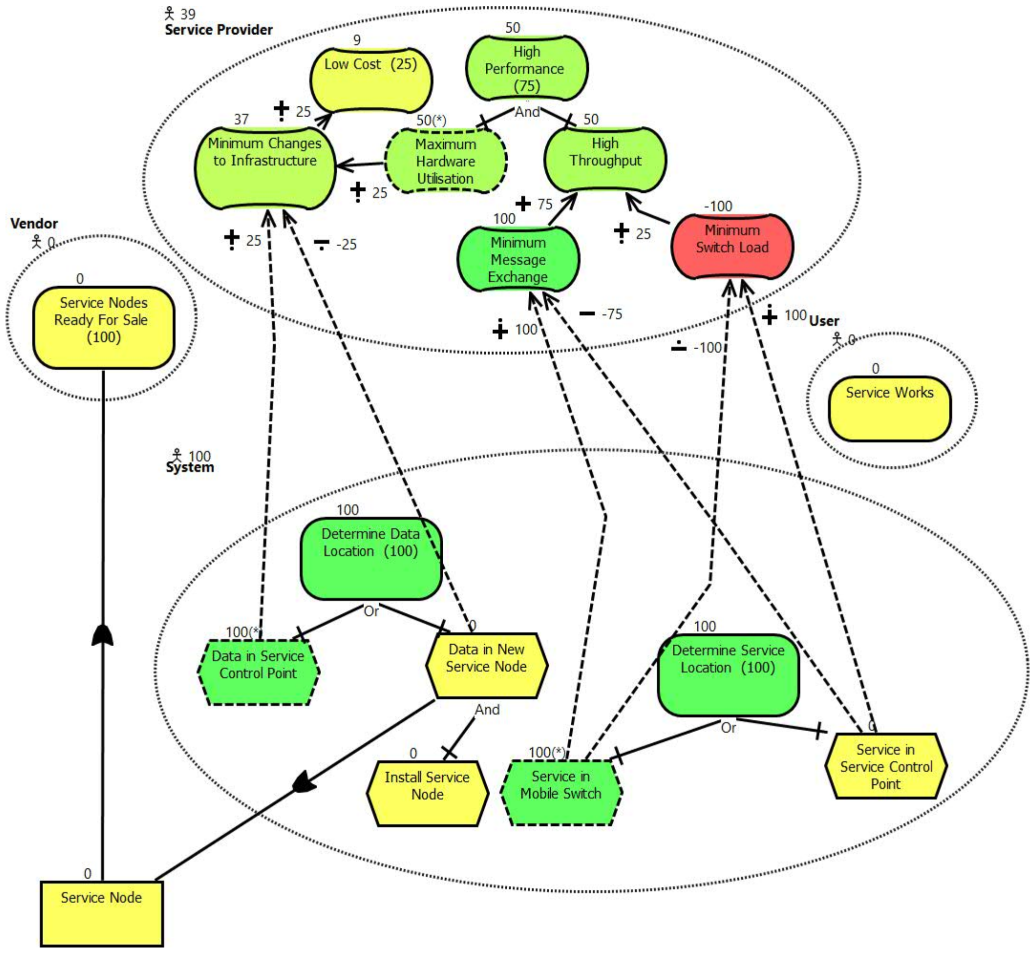


Figure 3.9: Example of a GRL graph model (Amyot and Mussbacher, 2011)

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

GRL defines the following qualitative contribution levels as standard (their equivalent quantitative contribution is also shown)

Table 3.2: GRL contribution levels in jUCMNav

Contribution Type	**Value**
Make	100
Some Positive	75
Help	25
Unknown	0
Hurt	-25
Some Negative	-75
Break	-100

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 following chapters, we will introduce goal-oriented ideas and concepts in behaviour-driven development (BDD) and in doing so, offer a treatment of non-functional requirements as a first-class concept in BDD.

4 Extending BDD - A goal oriented approach

In previous chapters we have presented definitions of requirements and requirements engineering, paying particular attention to the practices, concerning requirements elicitation, analysis, specification and validation found in most agile development methods, and BDD in particular.

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

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.

This focus on the needs of stakeholders, even if logical and beneficial, has lead to other goals, such as cross-cutting concerns or quality concerns of stakeholders, being neglected or addressed later in the software life-cycle. In fact, incorporating these goal types, such as non-functional requirements, into the different phases of the software life-cycle is a very hard task. Researchers face many challenges including their great diversity, subjective nature, difficulty in formalisation, incorporating these requirements into models used for specifying functional requirements and resolving conflicts among non-functional requirements (Laleau and Matoussi, 2008). Behaviour-driven development is one such model, traditionally used for specifying functional requirements and where their non-functional counterparts are rarely given the same relevance.

In this chapter we present, through data publicly available, an application of our extension to BDD to a subset of the requirements of a case study dataset, including the treatment of non-functional requirements as first class citizens alongside functional requirements. Moreover, our approach is independent of the particular taxonomy chosen and remains faithful to the principles of agile development discussed in earlier chapters.

The next chapter will focus on the changes made to Gherkin to support our methodology and also, detail a translator from feature files specified in Gherkin to GRL.

4.1 Methodology

We may be overusing the word methodology, given that some authors consider BDD not to be a methodology in its own right (Smart, 2014). The extension we are proposing to BDD does not make it a full blown methodology (e.g. we are mostly silent regarding management of requirements using BDD) but it is useful to refer to our collection of methods and techniques using one word other than BDD, hence the choice for *methodology*.

There are not that many references in the literature for approaches to handling non-functional requirements in agile methods and the ones that exist, are consistent with each other. Davies (2010) defends that agile teams should take responsibility for eliciting non-functional requirements and they should do it early-on by discussing with users the capabilities of the system, like quality of service and longevity, at the start of the project. The author is of the opinion, there are three approaches to describe these required capabilities in agile methods. Some teams like to use the usual story format, as in this way, all requirements look the same, can be understood by everyone including non-technical individuals, and therefore have the same visibility as other stories. Other teams keep it simple and simply factor non-functional requirements into acceptance criteria for affected user stories. Finally, last approach is to introduce 'technical stories' to cover anything that needs to be built or restructured but is too difficult to explain to business people. These story cards are typically free format and expressed in a language that only the development team understands (Davies, 2010). A team at Connextra is credited with the following story card format [1]

As a <*Actor* >, **I want** <*Goal* >, **So that** <*Benefit* >

Cohn (2015) advises the use of the story format and gives examples where, in his view, non-functional characteristics can be expressed using the template above

As a CTO
I want the system to use our existing orders database rather than create a new one
So that we don't have one more database to maintain

or

As a user
I want the site to be available 99.999 percent of the time I try to access it
So that I don't get frustrated and find another site to use

Trying to write non-functional requirements in this template is a good exercise, as it helps make sure we understand who wants what and why, as the above examples demonstrate. Cohn (2015) points out though that we should pay attention not to get obsessed with this format and instead specify it in whatever way feels natural, if using the template leads to confusingly formed sentences.

In our research, we found the user story format appropriate to write both functional and non-functional requirements. Our extension to Gherkin requires writing requirements in user story format, as we need to differentiate the goal from the stakeholder/role it intends to benefit. We do not believe this to be a significant drawback, but it does represent a breaking change with existing feature files. A refactoring to the extended Gherkin parser described in chapter 5 could relax this constraint and allow existing features to be parsed successfully.

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 or user-friendliness. In view of mathematical functions, in the form of $\textit{function} : \textit{Input} \rightarrow \textit{Output}$, just about anything that addresses characteristics of a function, a function's input or output or relationships between the two, can be considered as non-functional requirement.

[1] see http://agilecoach.typepad.com/photos/connextra_user_story_2001/connextrastorycard.html for examples

Leffingwell (2011) identifies, together with functional and non-functional requirements, design constraints as another class of requirements (see table 4.1). Design constraints typically originate from one of three sources: some necessary restriction of design options, conditions imposed on the development process itself, and regulations and imposed standards.

Table 4.1: Three types of requirements (Leffingwell, 2011)

Requirement Type	**Description**
Functional requirements	Express how the system interacts with its users-its inputs, its outputs, and the functions and features it provides
Non-functional requirements	Criteria used to judge the operation or qualities of a system
Design constraints	Restrictions on the design of a system, or the process by which a system is developed, but that must be fulfilled to meet technical, business, or contractual obligations

Our approach integrates all of the above requirements in a coherent manner, where both non-functional requirements and design constraints are treated as softgoals that can be specified alongside functional requirements. In extending BDD, our research aimed at integrating goal-reasoning concepts, while being shaped by the philosophy and principles of agile development. In other words, we extend BDD by

- following a lightweight approach
- requiring minimal changes to Gherkin
- preventing subversion of the spirit behind *Specification by Example*
- respecting the strengths of BDD and GRL
- following a separation of concerns approach when integrating Gherkin and GRL

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, and more so when documenting the results of technical research. 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 derived from non-functional requirements.

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

- acquiring or accessing knowledge of 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,
- deducing 'operationalisations' (possible design alternatives for meeting NFRs in the target system)
- dealing with ambiguities, trade-offs and priorities, and interdependencies among NFRs and operationalizations
- selecting operationalisations
- supporting decisions with design rationale
- evaluating the impact of decisions.

Again, these steps are not followed in sequence and we do not intend to introduce each, one at a time. It will be clear from the presentation, what each step consists of and when it is being applied.

In presenting our results and describing our methodology, we will make use of case study data contained on the *PROMISE* (Predictor Models in Software Engineering) requirements dataset (*PROMISE*, 2015). The *PROMISE* dataset consists of 625 requirements collected from 15 software development projects. Among them, 255 items are marked as functional requirements and the remaining 370 non-functional requirements items are classified into 11 categories, such as security, performance and usability.

One of the projects included in the dataset, is a *meeting scheduler*. Functionally, the *meeting scheduler* is required to create meetings, send invitations, book conference rooms, book room equipment and so on. The *meeting scheduler* example includes 27 functional requirements and 47 non-functional requirements, covering different aspects of the system, such as interoperability, usability, security, user friendliness, etc. The dataset also specifies for each non-functional requirement if it traces to a functional requirement, that is if that functional requirement is somehow constrained by the non-functional requirement. That mapping is presented in Appendix A, table A.3 (the absence of a particular non-functional requirement identifier from the columns of the table, implies there is no dependency between that non-functional and any of the functional requirements). The entire list of original functional and non-functional requirements of the *meeting scheduler* case study can also be found in Appendix A. Note that to facilitate tracing to the original *PROMISE* dataset we are retaining the requirements identifiers used therein.

We were unsatisfied with the choice of taxonomy used in the dataset, as the data contains no guidance or indication is this respect. Also, some of the categorisations used were arguably inconsistent. For example, functional requirement 48 is originally impacted by non-functional requirements 138, 140, 142 and 162 and all of these have been categorised as Operational, whilst in our view 138 and 140 are more pertinent to interoperability constraints and 142 is more of a design constraint, rather than a specific non-functional requirement.

We also took into consideration an ontological approach and analysis done to this case study dataset (Li et al., 2014). Their classification includes three basic categories of requirements, 'functional requirement (FR)', 'quality requirement (QR)', and 'constraints over function (CF)'. These would be modelled by functional goals, quality goals and function constraints in their conceptual model. They have also identified an additional category, 'Domain Assumption (DA)' that, in their words, can operationalise any other goal.

The authors analysed the 47 non-functional requirements within the dataset, and identified 21 QRs, 9 FRs, 14 Qrs+FRs, 2 CF+QR, and 1 DA. In the interest of space, we have decided to ignore any non-functional requirement that has been identified as FR, given that treating these would have no other consequence than growing our list of functional requirements to address. Therefore, and for the above reason, requirements 140 and 142 were removed from our list of non-functional requirements to take into consideration.

In this context, we opted by using the categorization in ISO/IEC 25010:2011, and reproduced again in figure 4.1 for easy referencing.

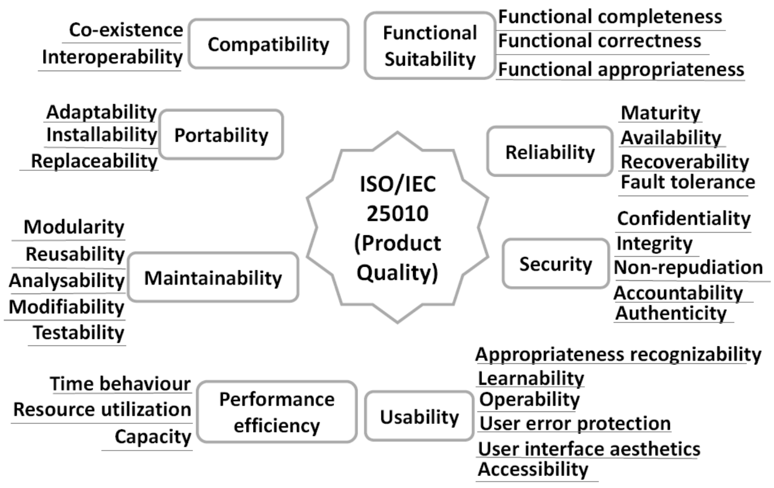


Figure 4.1: ISO 25010 quality model in ISO/IEC 25010:2011

In what follows, we use 10 non-functional requirements (the ones that impact one or more of the functional requirements), with the meaning intended by Leffingwell, that is, requirements that define criteria used to judge the operation or qualities of a system (Leffingwell, 2011). Other requirements, listed originally as non-functional requirements, fall in the category of design constraints, such as requirements with identifiers 139, *'The product will function alongside server software on any operating system where the Java runtime can be installed'* or 141, *'The product must make use of web/application server technology. Open source examples include Apache web server Tomcat and the JBoss application server'*.

In the interest of space and keeping discussion relevant, we will address all three types of requirements, that is, functional, non-functional and design constraints, but will devote more attention to the requirements contained in the mapping table A.3. The reasons for this will become clear when we describe our extended format for feature files, as in our methodology, design constraints are addressed no differently than non-functional requirements with a global reach, one that causes them to have an impact on all other goals.

Before describing how we model this set of requirements using our extended BDD methodology, we need to surface some constraints that we were faced with and that shaped our approach. We had to assume that the list of requirements was consistent with original goals of stakeholders, whoever they may be. This was the first hurdle we had to overcome, and that was identifying the different stakeholders of the system. Note that it was not our intention to identify all possible stakeholders, but rather only the ones that could be derived from the dataset.

We note also, that we did not have access to stakeholders and could not confirm assumptions, refine requirements or deal with ambiguities, or had access to business users from whom we could obtain further knowledge about the domain. This is not to say that we cannot use our methodology with confidence, it is just that we need to be appreciative of the constrained context in which our research took place.

In other words, our focus is on proving the feasibility (or not, for that matter) of our approach and to present the benefits, challenges and conclusions of our research. It is not to design the best possible *meeting scheduler* given a set of elicited requirements. That would just not be possible without a significant amount of guess work.

Chapter 5 describes in detail current Gherkin implementation and changes we introduced to the parser to support our methodology. Therein, we describe the current Gherkin class model. At the centre of the model are *Features*, consisting of a name, a description, an optional list of tags and comments, and a list of *Scenario Definitions* with an optional *Background*. A name is synonymous with title, and description is a block of text in no particular format. *Keyword* is filled by the parser and represents what lexeme was parsed (e.g. for a feature, one can use 'Feature', 'Business Need' or 'Ability') when processing each line of a feature file. *Background* adds context to the scenarios in a single feature and is similar to a scenario containing a number of *Steps*. *Tags* can be used to group features and scenarios together. Every *scenario* consists of a list of *Steps*, which must start with one of the keywords *Given*, *When*, *Then*, *But* or *And*. These *Steps*, are the interface to external tools to support testing and automated test execution approaches.

In the next section, we will introduce our changes to Gherkin by using a subset of requirements from the *meeting scheduler* case study.

4.2 Meeting scheduler case study

Based on the set of functional requirements (see table A.1) we devised the following list of stakeholders

Procurement Manager
Represents the concerns and interests of those that select, purchase and buy software

Privileged User
Represents a user that due to reasons of seniority or others, has privileged access to a service or application

Meeting Organiser Represents the concerns and interests of those users that want to organise meetings

Facilities Manager
Represents those individuals that are responsible for the security, maintenance and services of work facilities to ensure that they meet the needs of the organisation and its employees

When addressing each requirement we determined which stakeholder does the requirement pertain to. While there may be multiple stakeholders impacted by the requirement, in our approach only one of them will have the requirement assigned to. This is a current limitation of our approach, even though for most practical purposes, and without loss of generality, each goal usually pertains to at most one stakeholder.

In order to describe changes to Gherkin we will make use of a subset of requirements from the *meeting scheduler* case study. This subset of requirements is listed below

Table 4.2: Exemplar subset of meeting scheduler requirements

Requirement ID	**Requirement Description**
R 48	The product shall record meeting entries
R 63	The product shall record updated meeting agendas
R 67	The product shall record different meeting types
R 73	The product shall have an intuitive user interface

The reason these requirements are of particular interest is because, requirements 63 and 67 are impacted by 48, while 73 is a global non-functional requirement, in the sense that constrains all other functional requirements, and these variations cover all of our extensions to BDD.

Starting with requirement 48, it can be argued that recording meetings is of particular interest to the *Meeting Organiser* role. We then introduce our first change to Gherkin, which is being able to specify requirements in user story format.

```
Feature:    R48 - The product shall record meeting entries

            As a Meeting Organiser
            I want to record meeting entries
            So that I can always access a persisted record of each meeting
```

Source Code / Feature 4.1: Changes to Gherkin – User story format

We introduce the concepts of an *Actor*, in this case *Meeting Organiser*, representing a stakeholder or system; a *Goal*, in this case *record meeting entries*, representing an actor's high-level desire or need and a *Benefit*, in this case *I can always access a persisted record of each meeting*, which represents the anticipated benefit for an actor if the specified goal is achieved.

Actors are extremely relevant, as they represent stakeholders that usually have different, and at times conflicting concerns, and not only those need to be addressed and conflicts resolved, but also the actor's satisfaction level, needs to be assessed.

Goals equate very closely with functional requirements and it will be by satisfying them (and also *Sofgoals*, representing non-functional requirements and addressed later) that an actor's satisfaction level can be evaluated. *Benefits* are used to document the perceived benefit of achieving a goal, and help justify modelling decisions in GRL.

Actors and *Goals* are GRL concepts (see chapter 3, section 3.3.1 for details on these concepts) that will be used by a Gherkin to GRL translator that will be described in chapter 5. *Benefits* will not be translated to GRL as we did not find a suitableGRL element to map them to.

In order to address conflicts amongst *Goals*, we extend the user story format with one or more references to other goals that may be impacted by a given goal (see listing 4.2). The keyword used is *Which may impact* followed by a *Goal* that needs to be defined in another feature file.

```
Feature:    R48 - The product shall record meeting entries

            As a Meeting Organiser
            I want to record meeting entries
            So that I can always access a persisted record of each meeting
            Which may impact record a meeting agenda
            Which may impact record different meeting types
```

Source Code / Feature 4.2: Changes to Gherkin – Impacted goals

The careful reader will have already identified *record a meeting agenda* and *record different meeting types* as the goals for requirements 63 and 67. To our knowledge, this is the first time that, in BDD practices, goals of stakeholders have been collectively addressed and potential conflicts or dependencies identified and taken into consideration. In GRL this will correspond to a *Dependency* link.

If we refer to table A.3, that maps functional to non-functional requirements that impact them, we note that requirement 48 is constrained by an *Interoperability* quality characteristic. That leads us to the next change, that is, attaching constraints to behaviours of systems or applications (see listing 4.3).

```
Feature:    R48 - The product shall record meeting entries

            As a Meeting Organiser
            I want to record meeting entries
            So that I can always access a persisted record of each meeting
            Which may impact record different meeting types
            Which may impact record a meeting agenda

Constrained by:
            | Interoperability | Help |
```

Source Code / Feature 4.3: Changes to Gherkin – Constraints

While the *Feature* keyword is used to name and mark the beginning of a feature file, the keyword *Constrained by* (note that *Without ignoring* may also be used) lists the quality characteristics that constrain this functionality, and therefore should not be ignored. The lines immediately below are parsed as a table (using | as a cell separator) to list the non-functional characteristics and the expected *contribution* level for the satisfaction of the goal. In other words, and referring to this feature in particular, we expect Interoperability to *help* (see chapter 3, section 3.3.1 for more on contribution levels) achieve the *record meeting entries* goal. In GRL this will correspond to a *Contribution* link.

Interoperability is appropriate as a grouping term, but is not specific enough to support analysis. If we revert to mapping table A.3 we can see that this requirement is impacted by four non-functional requirements, all related to the *Interoperability* quality characteristic. Table 4.3 lists those requirements

As we mentioned before, 140 and 142 have been considered to be functional requirements (Li et al., 2014) and therefore, will not receive our attention as they would need to be addressed in separate feature files and would complicate this description process further. We are then left with requirements with identifiers 138 and 162.

In BDD, and in the spirit of *Specification by Example*, we clarify requirements details through examples and these, in the context of feature files, are represented by *Scenarios*. Listing 4.4 shows the revised feature file

Note that the two first scenarios are tagged with the tag *NFR* to indicate they are examples of a non-functional characteristic. We also state the *Contribution* level (*Help*) of each particular

Table 4.3: Four non-functional requirements impacting functional requirement 48

Requirement Type	Description
R 138	The product must work with most database management systems (DBMS) on the market whether the DBMS is colocated with the product on the same machine or is located on a different machine on the computer network.
R 140	The product will require collaboration with a database management system (DBMS). The DBMS may be located on the same machine as the product or on a separate machine residing on the same computer network.
R 142	A database management system such as Oracle DB2 MySql or HSQL will need to be integrated with the product
R 162	The product must be able to interface with various database management systems. The product shall communicate successfully with the database management system on 100\% of all transactions.

```
Feature:    R48 - The product shall record meeting entries

            As a Meeting Organiser
            I want to record meeting entries
            So that I can always access a persisted record of each meeting
            Which may impact record different meeting types
            Which may impact record a meeting agenda

Constrained by:
            | Interoperability | Help |

@NFR
Scenario:   Meeting entries can be recorded in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording meeting entries in DBMS are successful
            Which helps Interoperability

Scenario:   Record simplest meeting
            Given the application has been started
            And I choose to add new meeting
            When I specify a meeting name and date
            And I proceed to save the meeting
            Then the meeting should be recorded
```

Source Code / Feature 4.4: Changes to Gherkin – Scenarios

scenario to the attainment of the quality desired, in this case, *Interoperability*. We believe this communicates better what behaviour is expected from the system than taking the individual requirements 138 and 162 on their own. The latter option, made them slightly confusing as they seemed to overlap to some extent and were unrelated, where in fact they constrain the same functionality and contribute to attain the same *Interoperability* quality.

We are also listing a third scenario, *Record simplest meeting*, which is an ordinary BDD example detailing the expected behaviour of this feature. Because we are modelling behaviours using Goals, and in this particular case, the Goal is to *record meeting entries*, this is equivalent of saying that *Record simplest meeting*, if satisficed, contributes to the satisfaction of *record meeting entries*. We have set an assumption, that if no contribution level is specified – as in this case –, then a default contribution *Help* level shall be used.

In GRL, we will use *Tasks* to represent scenarios and the two situations described above are modelled using *Contributions*, from *Task* to *Softgoal* in the first situation, and from *Task* to *Goal* in the second.

The complete feature file is listed below with remaining scenarios included.

```
Feature:    R48 - The product shall record meeting entries

            As a Meeting Organiser
            I want to record meeting entries
            So that I can always access a persisted record of each meeting
            Which may impact record different meeting types
            Which may impact record a meeting agenda

Constrained by:
            | Interoperability | Help |

@NFR
Scenario:   Meeting entries can be recorded in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording meeting entries in DBMS are successful
            Which helps Interoperability

Scenario:   Record simplest meeting
            Given the application has been started
            And I choose to add new meeting
            When I specify a meeting name and date
            And I proceed to save the meeting
            Then the meeting should be recorded

Scenario:   Record meeting for a date in the past fails with a warning to the user
            Given the application has been started
            And I choose to add new meeting
            When I specify a date in the past
            And I proceed to save the meeting
            Then the meeting should not be recorded
            And the user is warned with an appropriate message
            Which helps record meeting entries

Scenario:   Record complex meeting
            Given the application has been started
            And I choose to add new meeting
            When I specify a meeting with 10 attendants
            And I choose an available conference room
            And I request a projector to be available
            And I proceed to save the meeting
            Then the meeting should be recorded
            With some positive contribution to record meeting entries
            Contributing to help Interoperability
```

Source Code / Feature 4.5: Changes to Gherkin – Complete feature

A last point to make regarding this feature, is that it is possible to specify, for the same scenario, contributions to both the feature's goal and one or more quality characteristics (refer to scenario *Record complex meeting*). Note that, on these occasions, the scenario should not be tagged with *NFR* as it is not an example of a quality requirement only, but rather an example of a feature's goal that also happens to contribute to attain a quality characteristic.

The remaining features, corresponding to requirements with identifiers 63 and 67 can be found in listings 4.6 and 4.7.

The last requirement to address is the one with identifier 73, and by doing so, we will introduce our last change to Gherkin and that is allowing the specification of global non-functional

```
Feature:    R63 - The product shall record updated meeting agendas

            As a Meeting Organiser
            I want to record a meeting agenda
            So that I can have access to the details of the meeting

Constrained by:
            | Interoperability | Help |

Scenario:   Record meeting agenda with details in text

Scenario:   Record meeting agenda with details in HTML

@NFR
Scenario:   Meeting agendas can be recorded in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording meeting agendas in DBMS are successful
            Which helps Interoperability
```

Source Code / Feature 4.6: Changes to Gherkin – R63 – The product shall record updated meeting agendas

```
Feature:    R67 - The product shall record different meeting types

            As a Meeting Organiser
            I want to record different meeting types
            So that I can chose the most appropriate type of meeting

Constrained by:
            | Interoperability | Help |

Scenario:   Record one to one meeting

Scenario:   Record team meeting

@NFR
Scenario:   Different Meeting types can be recorded in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording different meeting \
            types in DBMS are successful
            Which helps Interoperability
```

Source Code / Feature 4.7: Changes to Gherkin – R67 – The product shall record different meeting types

requirements or design constraints (see listing 4.8). For easy reference, we list requirement 73 and non-functional requirements it is impacted by, in table 4.4

Table 4.4: Cross-cutting non-functional requirements

Requirement ID	Requirement Description
R 73	The product shall have an intuitive user interface
R 149	The product shall allow for customization of start page and views preferences

Note that this feature, also introduces a new actor, namely *Procurement Manager* as we decided to model this cross-cutting concern as a goal of a stakeholder that carries that overall interest in the application's user interface and learnability characteristics. This learnability quality requirement is not something that was part of the original dataset, but a quality that we derived from our interpretation of the requirements.

```
Feature:  R73 - The product shall have an intuitive user interface

            @GLOBAL
            As a Procurement Manager
            I want the product to have an intuitive interface
            So that all users are able to use the product without difficulty

Constrained by:
            | user interface aesthetics | Help         |
            | Learnability              | SomePositive |

@NFR
Scenario:   Product has an intuitive interface
            Which makes user interface aesthetics
            Contributing to make Learnability
```

Source Code / Feature 4.8: Changes to Gherkin – R73 – The product shall have an intuitive user interface

These cross-cutting concerns (applies also to design constraints), will be distinguished by ordinary non-functional requirements by being tagged with a *GLOBAL* tag, allowing the parser and, more importantly, the translator from Gherkin to GRL to treat them appropriately. In GRL, they will correspond to *Dependency* links between the goal herein introduced (in this case, *product to have an intuitive interface*) and all other goals introduced so far. That is, satisfaction of any other goal, will be dependant on the satisfaction of this global reach goal.

We now turn our attention to GRL and how we can model the features covered so far. In chapter 5 we will describe a translator from Gherkin to GRL. The output of that translator can be imported into jUCMNav [2], a free, Eclipse-based [3] graphical editor and analysis and transformation tool for the User Requirements Notation (URN), including GRL.

This tool has been described in detail before, first in Jean-François Roy's masters thesis (Roy, 2007) and later in a research paper (Amyot et al., 2010), and we refer the interested reader to those two documents, as describing the tool in detail here, is out of scope.

For now, we describe possible uses of the models imported into jUCMNav – more precisely, a GRL catalogue of intentional elements, dependencies, contributions and decompositions – that correspond to the Gherkin features just described, for requirements with identifiers 48, 63, 67 and 73. Figure 4.2 shows a graph of the model that was imported into jUCMNav and that was generated using the translator described in chapter 5.

The model shows two actors (modelled as circles drawn with dashed lines), and within each, their set of goals defined in the story format extension for feature files described earlier. We can also observe dependencies (thick lines with a filled arrow in the middle) pointing from a dependent goal to others it depends upon (e.g. *Record a meeting agenda* depends upon *Record meeting entries*).

Also, all global qualities have incoming dependencies from all other goals (e.g. refer to *Product to have an intuitive interface* in the graph).

Three *Softgoals* are shown, namely *Interoperability*, *Learnability* and *User interface aesthetics* that also correspond to the ones identified earlier. Note that these *Softgoals* have to be

[2] see http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome for details

[3] see https://eclipse.org/ for details

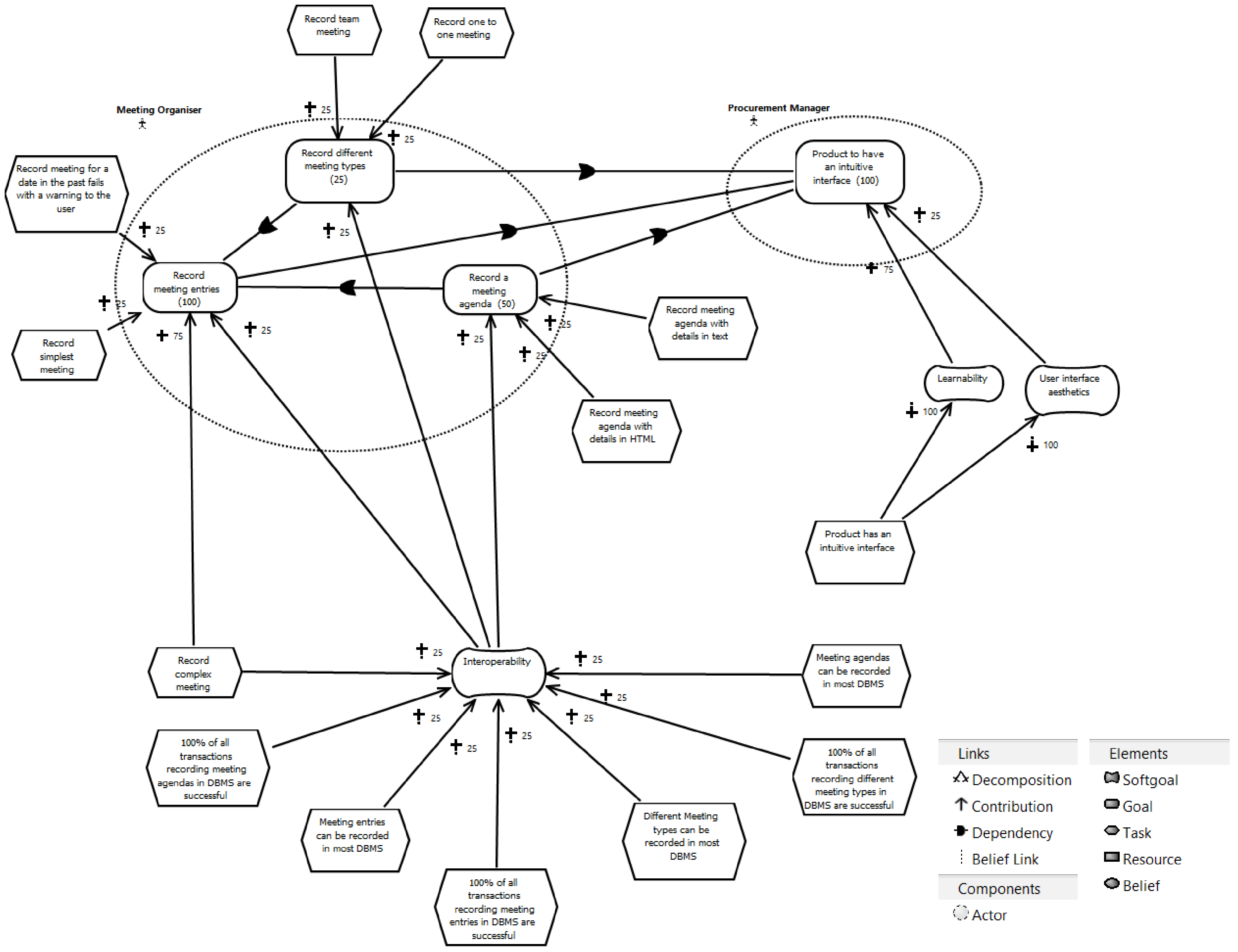

Record team meeting
Record one to one meeting
Meeting Organiser
Procurement Manager
Record meeting for a date in the past fails with a warning to the user
Record different meeting types (25)
Product to have an intuitive interface (100)
Record meeting entries (100)
Record a meeting agenda (50)
Record simplest meeting
Record meeting agenda with details in text
Record meeting agenda with details in HTML
Learnability
User interface aesthetics
Product has an intuitive interface
Record complex meeting
Interoperability
Meeting agendas can be recorded in most DBMS
100% of all transactions recording meeting agendas in DBMS are successful
Meeting entries can be recorded in most DBMS
100% of all transactions recording meeting entries in DBMS are successful
Different Meeting types can be recorded in most DBMS
100% of all transactions recording different meeting types in DBMS are successful
25
75
100
Links
Decomposition
Contribution
Dependency
Belief Link
Components
Actor
Elements
Softgoal
Goal
Task
Resource
Belief

Figure 4.2: Case study GRL Graph – Initial

previously defined, and our translator will ensure these are present by including a set of quality softgoals and decompositions with each run of the translator. We have chosen to base this set in the quality characteristics defined in 25010 (2011) but a different categorisation scheme can be used. This process will be described in more detail in chapter 5.

The hexagons define *Tasks* in GRL corresponding to scenarios defined in the feature files, with some contributing to Goals, other to Softgoals, and others to both (e.g. *Record complex meeting* contributes to *Interoperability* softgoal and to *Record meeting entries* goal).

The *contribution* lines (thick lines with arrows at the end) show a symbol and also a numeric value to depict the intended contribution level). These correspond to the contribution levels described in chapter 3, section 3.3.1.

Using GRL and the jUCMNav tool, we are able to develop analysis procedures by evaluating qualitative or quantitative satisfaction levels of the actors and intentional elements (e.g., goals and tasks) composing the model (Amyot et al., 2010).

Amyot et al. (2010) mentions that the analysis of goal models can be done in very different ways depending on the nature of the model and the purpose of the analysis. We believe that in our methodology, there are at least two types of analysis worth considering.

The first one consists of taking the model as it currently is, and assigning initial satisfaction levels to some of the leaf nodes corresponding to the *Tasks* in the model. This relates to scenarios being satisficed (or not) and triggering the satisfaction levels of goals and actors. In early stages of the software life-cycle, we may be interested in modelling the potential impact of scenarios in goal satisfaction of one or more stakeholders. In later stages, perhaps we have those scenarios already implemented and satisfied and want to assess current satisfaction levels of actors, in order to determine how far we may be from delivering software or application that satisfied its stakeholders in a sufficiently manner.

The second analysis type, considers design options for *Tasks* and/or *Goals* and then checking the impact of those decisions on the satisfaction levels of goals and actors. The process is very similar to the above, but here we are delving into the design domain and considering alternative options to operationalise – with the meaning intended by the *NFR Framework* – goals and tasks.

We will now show instances of these two types of analysis, but before proceeding, we need to specify *importance* levels for each of the goals that are bound to an Actor. For what follows, we set the importance of *Record meeting entries* to *High*, of *Record a meeting agenda* to *Medium* and of *Record different meeting types* to *Low*. The goal *Product to have an intuitive interface* was also set to *High* as it is the only goal for the actor *Procurement Manager*, and that seemed like a logical thing to do. This assignment of importance levels to goals bound to actors, is enforced by jUCMNav, without which, we would not be able to assess the satisfaction level of actors in the model.

We will start by first considering using the model as it currently is. Initial satisfaction levels for some of the intentional elements are provided in a *strategy* and then propagated to the other ones through the various links that connect them. We start by satisfying all *Tasks* corresponding to ‘functional’ scenarios (see figure 4.3), that is, those that are not tagged with *NFR*.

We note that both goals and actors have their satisfaction level untouched (showing zero on top of their respective icon). This is consistent with our expectation, since all goals are dependent on the *Product to have an intuitive interface* which we haven’t yet specified a satisfaction level for, and the current strategy did not cause a change to its satisfaction level. It is interesting to note the impact on both goals and actors of setting the satisfaction of that global constraint to either *Satisfied* (see figure 4.4) or *Weakly Denied* (see figure 4.5). The latter satisfaction level, could be interpreted, for example, as an indication of some users being dissatisfied with the

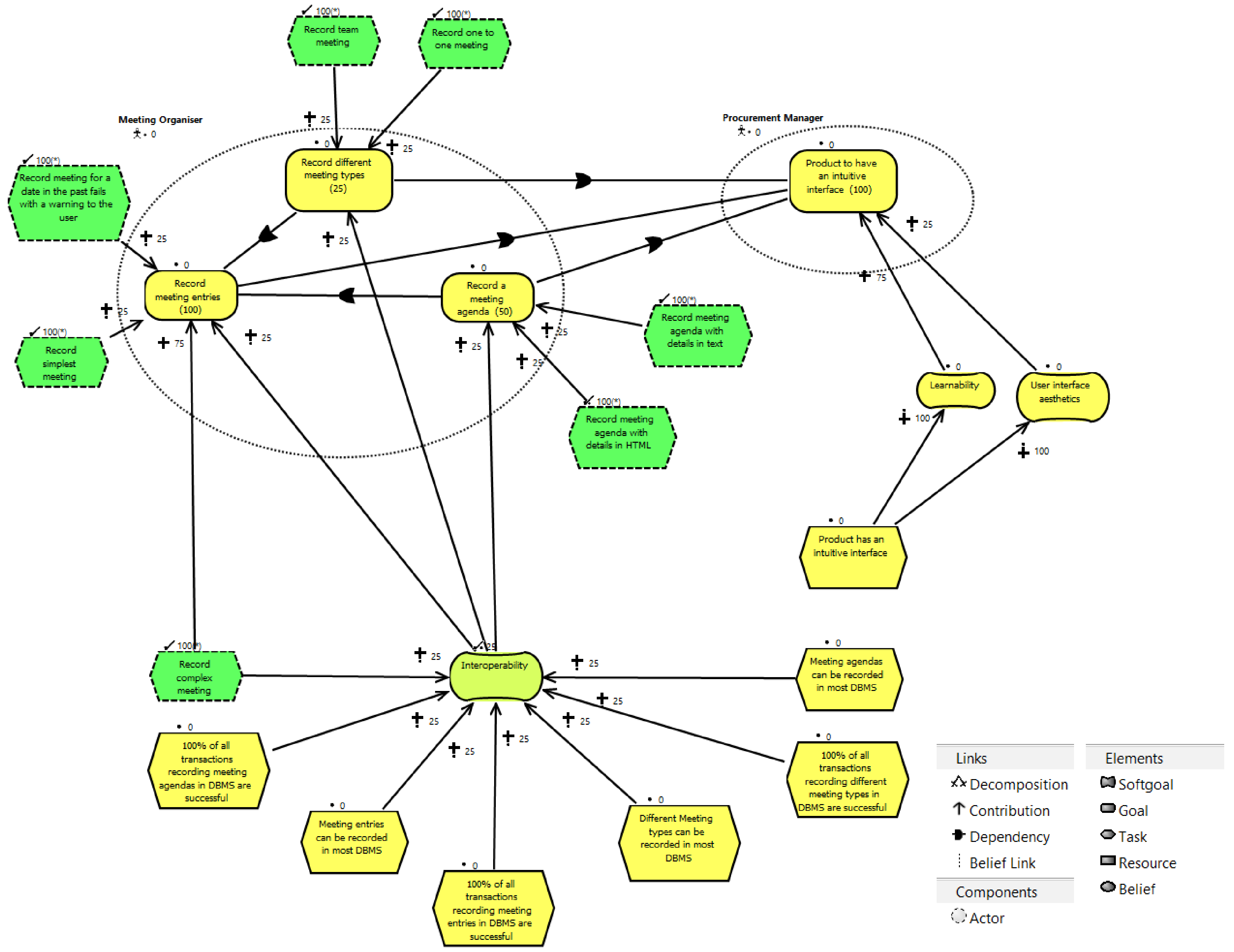

Meeting Organiser
Procurement Manager
Record team meeting
Record one to one meeting
Record meeting for a date in the past fails with a warning to the user
Record different meeting types (25)
Product to have an intuitive interface (100)
Record meeting entries (100)
Record a meeting agenda (50)
Record simplest meeting
Record meeting agenda with details in text
Record meeting agenda with details in HTML
Learnability
User interface aesthetics
Product has an intuitive interface
Record complex meeting
Interoperability
Meeting agendas can be recorded in most DBMS
100% of all transactions recording meeting agendas in DBMS are successful
Meeting entries can be recorded in most DBMS
100% of all transactions recording meeting entries in DBMS are successful
Different Meeting types can be recorded in most DBMS
100% of all transactions recording different meeting types in DBMS are successful
Links
Decomposition
Contribution
Dependency
Belief Link
Components
Actor
Elements
Softgoal
Goal
Task
Resource
Belief

Figure 4.3: Case study GRL Graph – All functional scenarios satisfied

user interface or the team facing implementation issues with the application's user interface.

What can be confirmed from the model is the importance that quality *Product to have an intuitive interface* has on the satisfaction level of goals and actors. When it is weakly denied the satisfaction level of other goals and actors is constrained to the minimum of their own satisfaction levels and the one of the global quality (e.g. -49 in this case). This should be a sign that the agile team should dedicate more attention and resources to improve attainability of that quality (this could be achieved by fixing any issues found or investing time in improving current implementation).

We can also consider design alternatives and perform an assessment of the relative effectiveness of these alternatives at the requirements level (Amyot et al., 2010). Figure 4.6 shows the same model as before but where some design options have been considered. In particular, *High availability* has been considered as a design option to help ensure 100% of all transactions in DBMS [4] are successfull; one of *Use ORM* [5] or *Use standard T-SQL only* were considered as options to attain interoperability with most DBMS and two distinct options were considered to address the constrain specified that the product should have an intuitive interface. The particular strategy chosen is a valid one as it ensures all goals and actors attain a positive satisfaction level. It may not be adequate if say, for example, there are other strategies that provide a better satisfaction level for the *Meeting Organiser* and this has been considered a key stakeholder.

Figure 4.7 shows the same model as before, with the same design options considered but with a different strategy. Here, by using an *external UI library* the satisfaction level of goals and actors is improved, even though *Interoperability* as a quality is *Weakly Denied*. This may be an acceptable strategy, if we are trying to maximise satisfaction levels of actors and goals and attaining *Interoperability* is not as important.

4.3 Summary

In this chapter, and in the context of Behaviour-driven development, we have described a methodology that addresses non-functional requirements in tandem with functional requirements. We have done it, by introducing a goal-oriented approach to the elicitation, analysis and modelling of requirements as aspirational goals of one or more identified stakeholders of an application or system.

We initially described, through a series of small transformational steps, the set of required changes to Gherkin, in order to translate requirements into goals, determine any dependencies and conflicts between goals, and more identify constraints to the satisfaction of those goals.

We have made use of a well known public dataset and, sampled a subset of functional and non-functional requirements to support the description of our methodology. In doing so, we have showed how requirements specified in Gherkin can be later modelled as a GRL graph in a specialised tool such as jUCMNav.

In the next chapter we will describe in detail the implementation changes to the Gherkin parser and also introduce a translator to convert one or more feature files into a GRL catalogue suitable to be imported into jUCMNav, for further modelling, refinement and analysis.

[4] Database Management Systems (DBMS)

[5] Object Relational Mapping

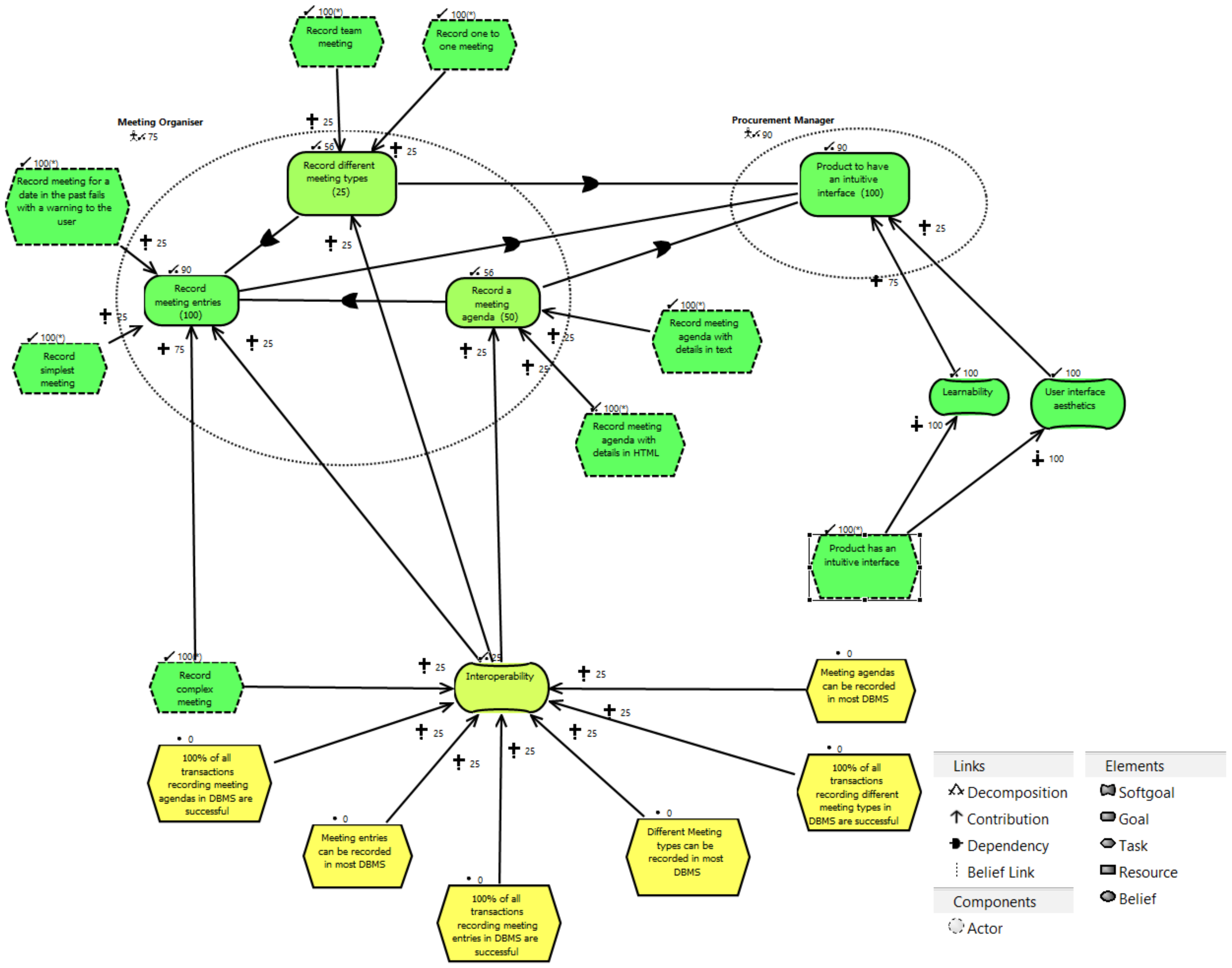

100(*) Record team meeting
100(*) Record one to one meeting
Meeting Organiser 75
Procurement Manager 90
100(*) Record meeting for a date in the past fails with a warning to the user
56 Record different meeting types (25)
90 Product to have an intuitive interface (100)
90 Record meeting entries (100)
56 Record a meeting agenda (50)
100(*) Record meeting agenda with details in text
100(*) Record meeting agenda with details in HTML
100(*) Record simplest meeting
100 Learnability
100 User interface aesthetics
100(*) Product has an intuitive interface
100(*) Record complex meeting
25 Interoperability
0 Meeting agendas can be recorded in most DBMS
0 100% of all transactions recording meeting agendas in DBMS are successful
0 Meeting entries can be recorded in most DBMS
0 100% of all transactions recording meeting entries in DBMS are successful
0 Different Meeting types can be recorded in most DBMS
0 100% of all transactions recording different meeting types in DBMS are successful
25
75
100
Links
Decomposition
Contribution
Dependency
Belief Link
Components
Actor
Elements
Softgoal
Goal
Task
Resource
Belief

Figure 4.4: Case study GRL Graph – All functional scenarios and global constraint satisfied

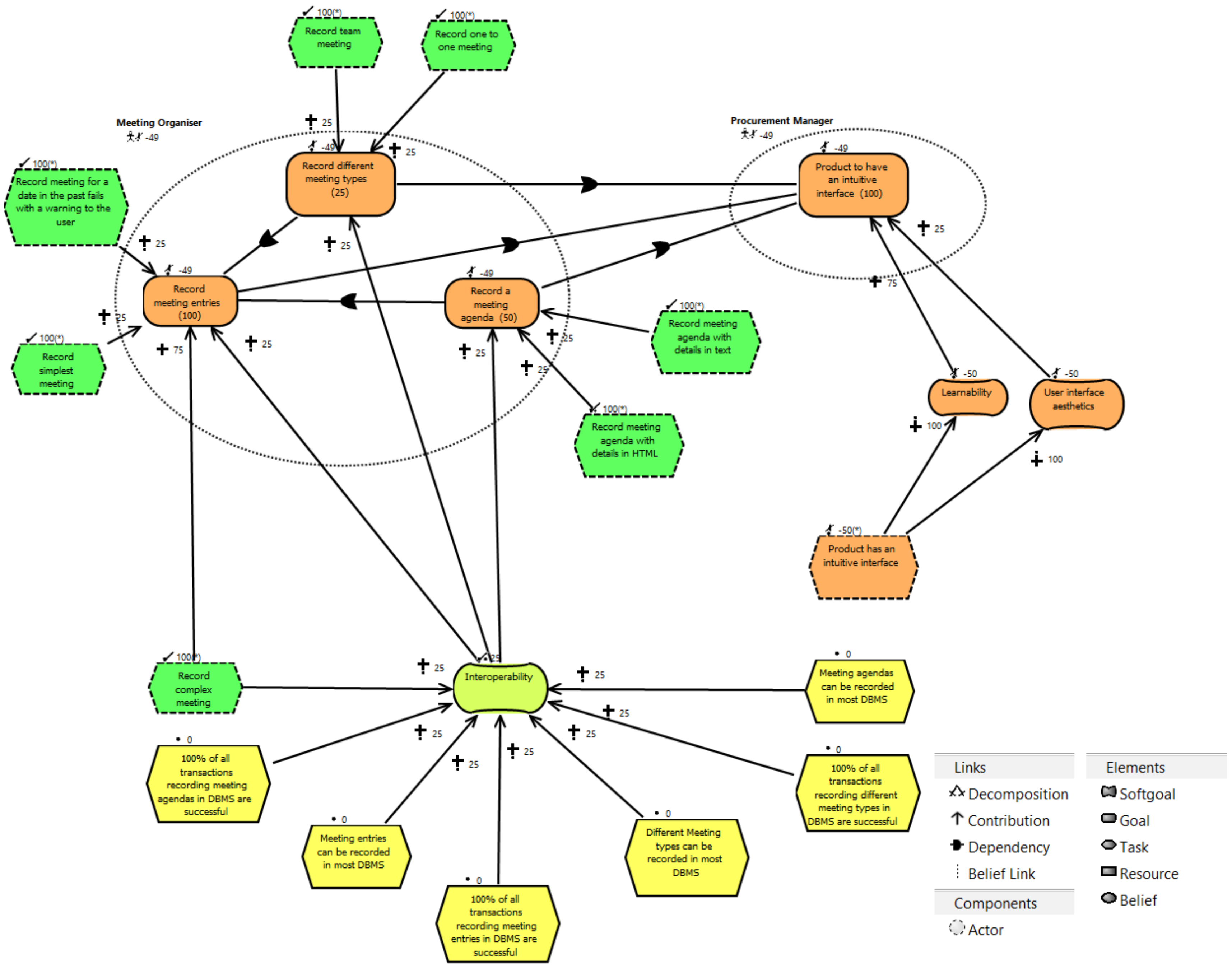

Meeting Organiser -49
Procurement Manager -49
100(*) Record team meeting
100(*) Record one to one meeting
-49 Record different meeting types (25)
100(*) Record meeting for a date in the past fails with a warning to the user
-49 Record meeting entries (100)
100(*) Record simplest meeting
-49 Record a meeting agenda (50)
100(*) Record meeting agenda with details in text
100(*) Record meeting agenda with details in HTML
-49 Product to have an intuitive interface (100)
-50 Learnability
-50 User interface aesthetics
-50(*) Product has an intuitive interface
25 Interoperability
100(*) Record complex meeting
0 Meeting agendas can be recorded in most DBMS
0 100% of all transactions recording different meeting types in DBMS are successful
0 Different Meeting types can be recorded in most DBMS
0 100% of all transactions recording meeting entries in DBMS are successful
0 Meeting entries can be recorded in most DBMS
0 100% of all transactions recording meeting agendas in DBMS are successful
Links
Decomposition
Contribution
Dependency
Belief Link
Components
Actor
Elements
Softgoal
Goal
Task
Resource
Belief

Figure 4.5: Case study GRL Graph – All functional scenarios satisfied, global constraint weakly denied

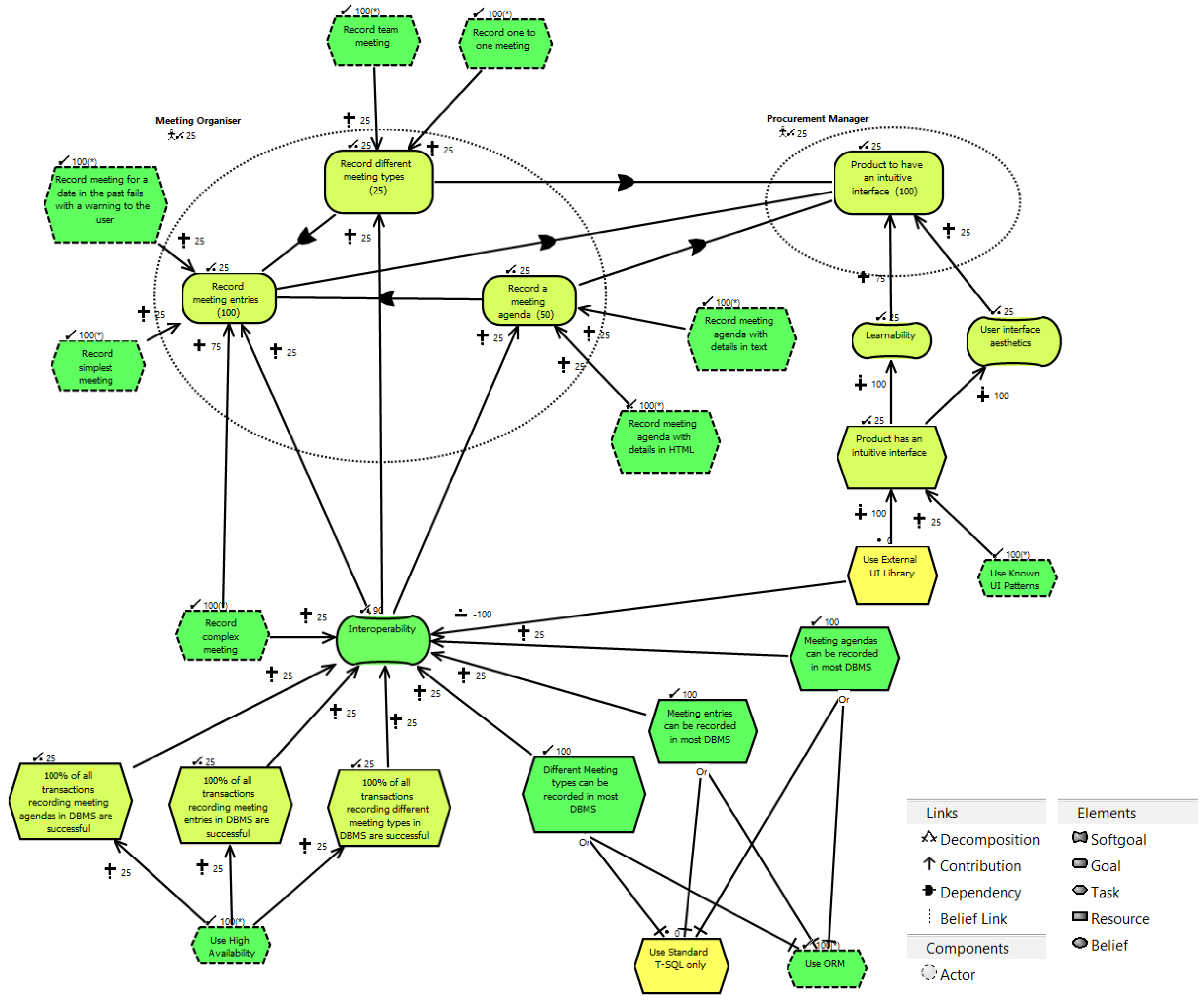

Meeting Organiser
Procurement Manager
Record team meeting
Record one to one meeting
Record different meeting types (25)
Product to have an intuitive interface (100)
Record meeting for a date in the past fails with a warning to the user
Record meeting entries (100)
Record a meeting agenda (50)
Record meeting agenda with details in text
Record meeting agenda with details in HTML
Record simplest meeting
Learnability
User interface aesthetics
Product has an intuitive interface
Use External UI Library
Use Known UI Patterns
Record complex meeting
Interoperability
Meeting agendas can be recorded in most DBMS
Meeting entries can be recorded in most DBMS
Different Meeting types can be recorded in most DBMS
100% of all transactions recording meeting agendas in DBMS are successful
100% of all transactions recording meeting entries in DBMS are successful
100% of all transactions recording different meeting types in DBMS are successful
Use High Availability
Use Standard T-SQL only
Use ORM
Or
Links
Decomposition
Contribution
Dependency
Belief Link
Components
Actor
Elements
Softgoal
Goal
Task
Resource
Belief

Figure 4.6: Case study GRL Graph – Design options considered (Option A)

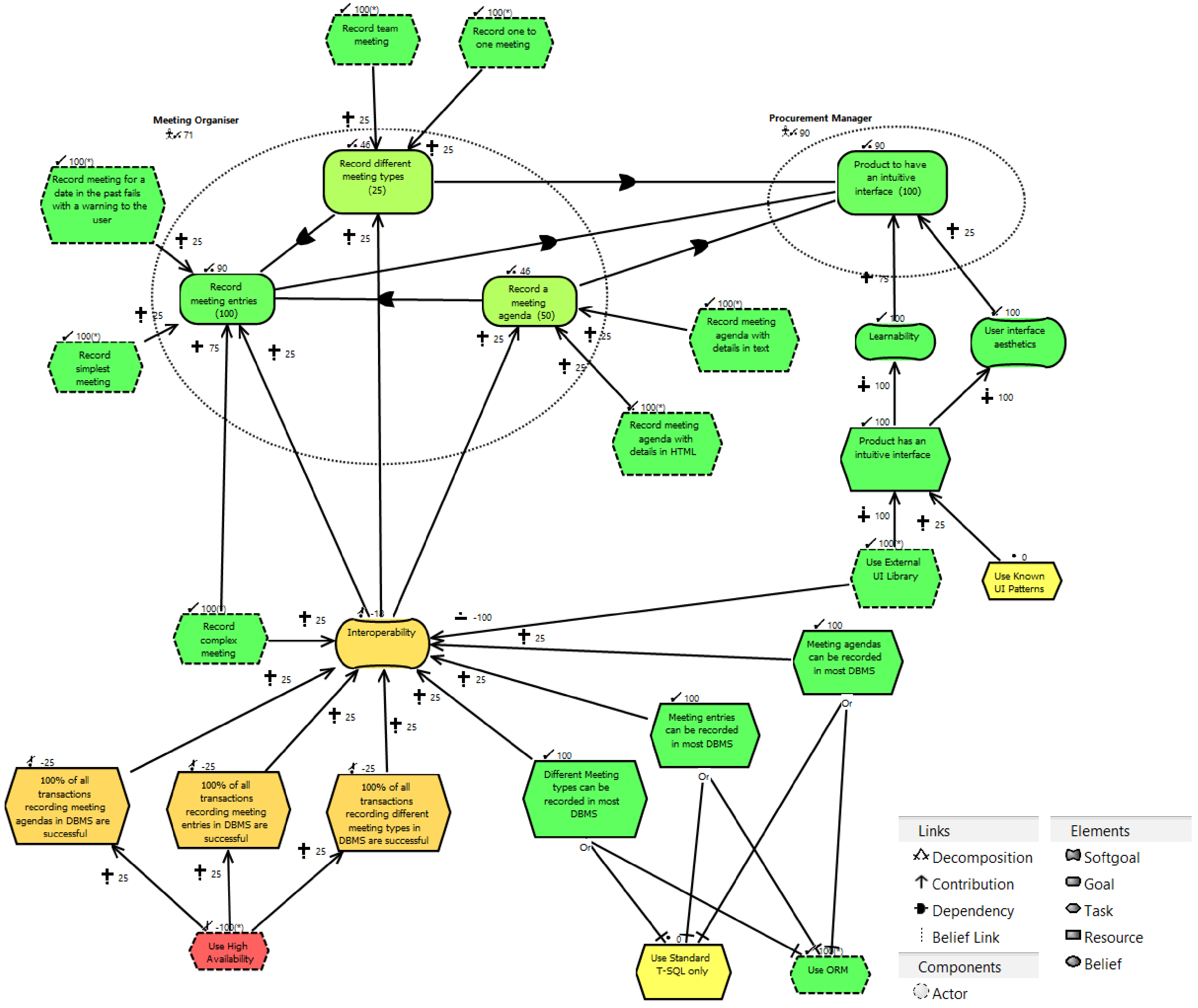

Meeting Organiser
71
Procurement Manager
90
Record team meeting
100(*)
Record one to one meeting
100(*)
Record different meeting types (25)
46
Record meeting for a date in the past fails with a warning to the user
100(*)
Record simplest meeting
100(*)
Record meeting entries (100)
90
Record a meeting agenda (50)
46
Record meeting agenda with details in text
100(*)
Record meeting agenda with details in HTML
100(*)
Product to have an intuitive interface (100)
90
Learnability
100
User interface aesthetics
100
Product has an intuitive interface
100
Use External UI Library
100(*)
Use Known UI Patterns
0
Record complex meeting
100(*)
Interoperability
-18
-100
Meeting agendas can be recorded in most DBMS
100
Meeting entries can be recorded in most DBMS
100
Different Meeting types can be recorded in most DBMS
100
Or
100% of all transactions recording meeting agendas in DBMS are successful
-25
100% of all transactions recording meeting entries in DBMS are successful
-25
100% of all transactions recording different meeting types in DBMS are successful
-25
Use High Availability
-100(*)
Use Standard T-SQL only
0
Use ORM
100(*)
25
75
Links
Decomposition
Contribution
Dependency
Belief Link
Components
Actor
Elements
Softgoal
Goal
Task
Resource
Belief

Figure 4.7: Case study GRL Graph – Design options considered (Option B)

5 From Gherkin to GRL

In the previous chapter we have exemplified how to use the proposed extension to BDD, handling non-functional requirements as first-class citizens alongside functional requirements. In order to achieve this undertaking, we had to alter the Gherkin parser to allow additional constructs.

In this chapter, we are detailing the changes made to the Gherkin parser by showing relevant code extracts accompanied with explanations in order to better describe those changes. We will be using version 3 of the Gherkin parser [1], a version that responds to issues with previous versions [2] (e.g. difficulty in adding support for new languages; difficult to build, so releases are very infrequent).

We will also detail a translator from Gherkin to an XML-based interchange format to be imported in a GRL tool (jUCMNav) and that allows goal modelling and analysis, and supports evaluation of satisfaction levels of actors and goals.

5.1 Gherkin Extended

When using the word Gherkin we can be referring to the language, the parser or the compiler. Gherkin is a parser and compiler for the Gherkin language – a domain specific language for the specification and documentation of requirements, focusing on the expected behaviours of a system or application which are exemplified using scenarios. In that follows the intended meaning of the use of the word Gherkin should be clear from the context of use, and if not we will clarify its intended meaning.

Gherkin (both parser and compiler), is implemented in multiple platforms and target languages, including *.NET / C#*, *JVM / Java*, *Javascript / Browser or Node.js*, etc. Note that, in what follows, we are using the *.NET / C#* version of the Gherkin parser. The following diagram provides an overview of the architecture of Gherkin

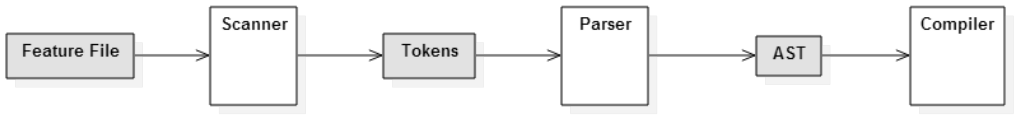


Figure 5.1: Gherkin Overview [3]

The scanner reads a feature file and creates a token for each line. The tokens are passed to the parser, which outputs an AST – a hierarchical syntactic structure of the source program commonly generated by parsers and parser generators – to be used by back-end phases of the compilation process. If the scanner sees a *# language header*, it will reconfigure itself

[1] see https://github.com/cucumber/gherkin3 for details
[2] see discussion in https://groups.google.com/forum/#!msg/cukes/YLKsqbBMBoI/DYhfFx8GBegJ for details
[3] sourced and replicated from https://github.com/cucumber/gherkin3

dynamically to look for Gherkin keywords for the associated language. The parser is generated by the *Berp* [4] parser generator as part of the build process. Berp takes a grammar file (*gherkin.berp*) and a template file (*gherkin-X.razor*) as input and outputs a parser in language X (see figure 5.2).

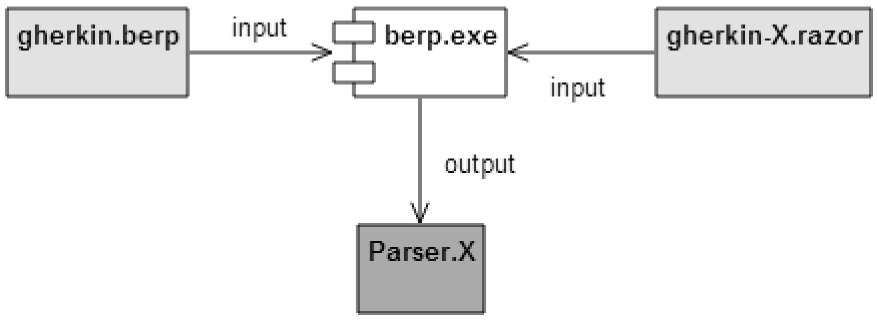


Figure 5.2: Berp Overview [5]

The keywords are defined in a file named *gherkin-languages.json*. Listing below shows a section of that file containing English keywords only, including new ones added as part of this research and that are shown within shaded area below. The new keywords are *asA*, *iWant*, *soThat*, *qualityAttribute*, *qualityReason*, *scenarioContribution* and *whichMayImpact*.

```
{ "en": {
        "and":                  [ "* ", "And " ],
        "background":           [ "Background" ],
        "but":                  [ "* ", "But " ],
        "examples":             [ "Examples", "Scenarios" ],
        "feature":              [ "Feature", "Business Need", "Ability" ],
        "given":                [ "* ", "Given " ],
        "name":                 "English",
        "native":               "English",
        "scenario":             [ "Scenario" ],
        "scenarioOutline":      [ "Scenario Outline", "Scenario Template" ],
        "then":                 [ "* ", "Then " ],
        "when":                 [ "* ", "When " ],
        "asA":                  [ "As a " ],
        "iWant":                [ "I want to ", "I want the " ],
        "soThat":               [ "So that " ],
        "qualityAttribute":     [ "Without ignoring" , "Constrained by"],
        "scenarioContribution": [
            "Breaking the ", "Which breaks ", "Contributing to break ",
            "Helping the ", "Which helps ", "Contributing to help ",
            "Hurting the ", "Which hurts ", "Contributing to hurt ",
            "Making the ", "Which makes ", "Contributing to make ",
            "With some positive contribution to ",
            "With some negative contribution to "
                                ],
        "whichMayImpact":       [ "Which may impact " ]
       }
}
```

Source Code / Feature 5.1: Extended Gherkin – English keywords

It is important to note that Gherkin has been localised for several languages, and in our research we only extended the English language keywords, but it would be relatively easy

[4] see https://github.com/gasparnagy/berp for details and source code
[5] sourced and replicated from https://github.com/cucumber/gherkin3

to add support for more languages. Gherkin uses *JSON* (Javascript Object Notation) [6] – a lightweight data-interchange format – to represent keywords in all supported languages.

As we mentioned before, the parser generates an AST that can be used in back-end stages of compilers or as an intermediate representation that other tools can use. We will make use of this representation when we describe the translator from Gherkin to GRL. The following class diagram describes the AST generated by the original parser, that is, the one without our changes.

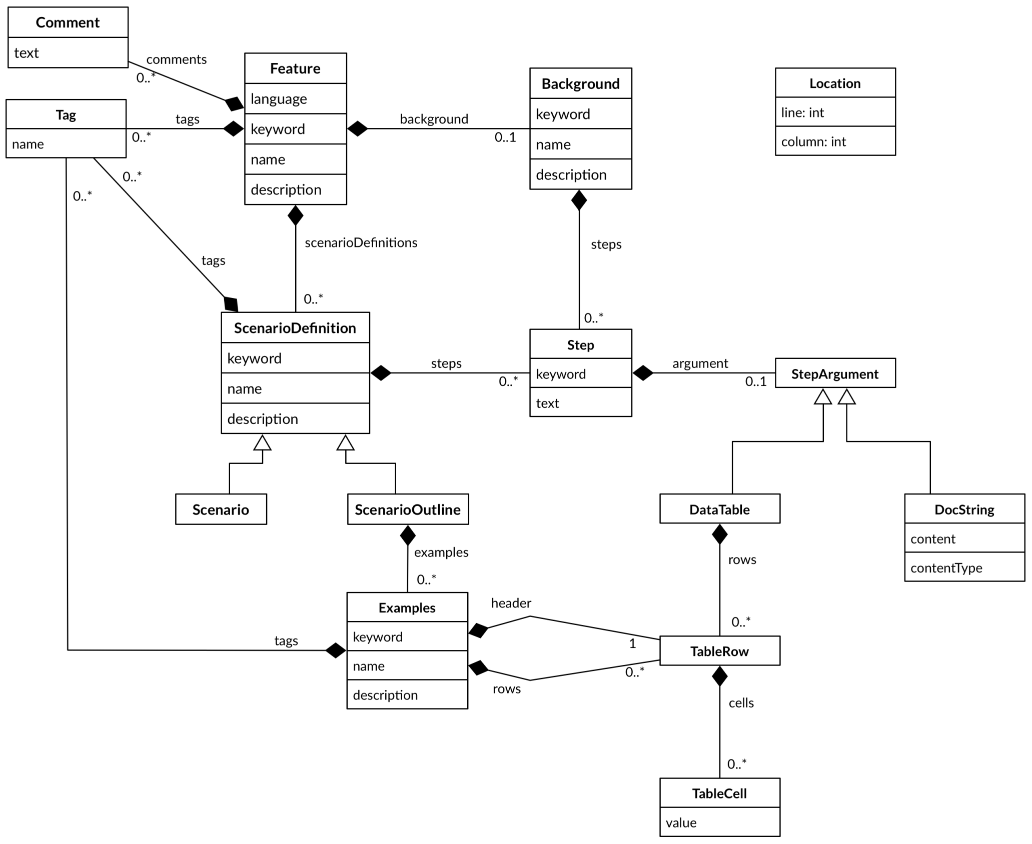


Figure 5.3: Original Gherkin class model [7]

In the diagram, every class represents a node in the AST. Every node has a *Location* that describes the line number and column number in the input file. A requirement of the implementation is that the AST must have a JSON representation (used for testing).

At the centre of the model are *Features*, consisting of a name, a description, an optional list of tags and comments, and a list of *Scenario Definitions* with an optional *Background*. A name is synonymous with title, and description is a block of text in no particular format. *Keyword* is filled by the parser and represents what lexeme was parsed (e.g. for a feature, one can use 'Feature', 'Business Need' or 'Ability') when processing each line of a feature file. *Background* adds context to the scenarios in a single feature and is similar to a scenario containing a number of *Steps*. *Tags* can be used to group features and scenarios together. Every *Scenario* consists of a list of *Steps*, which must start with one of the keywords *Given*, *When*, *Then*, *But* or *And*. These *Steps*, are the interface to external tools to support testing and automated test execution approaches.

[6] see http://json.org/ for details

[7] sourced from https://github.com/cucumber/gherkin3

In order to introduce our changes, we present now a revised class model for the AST returned by the parser after applying the changes to Gherkin discussed next.

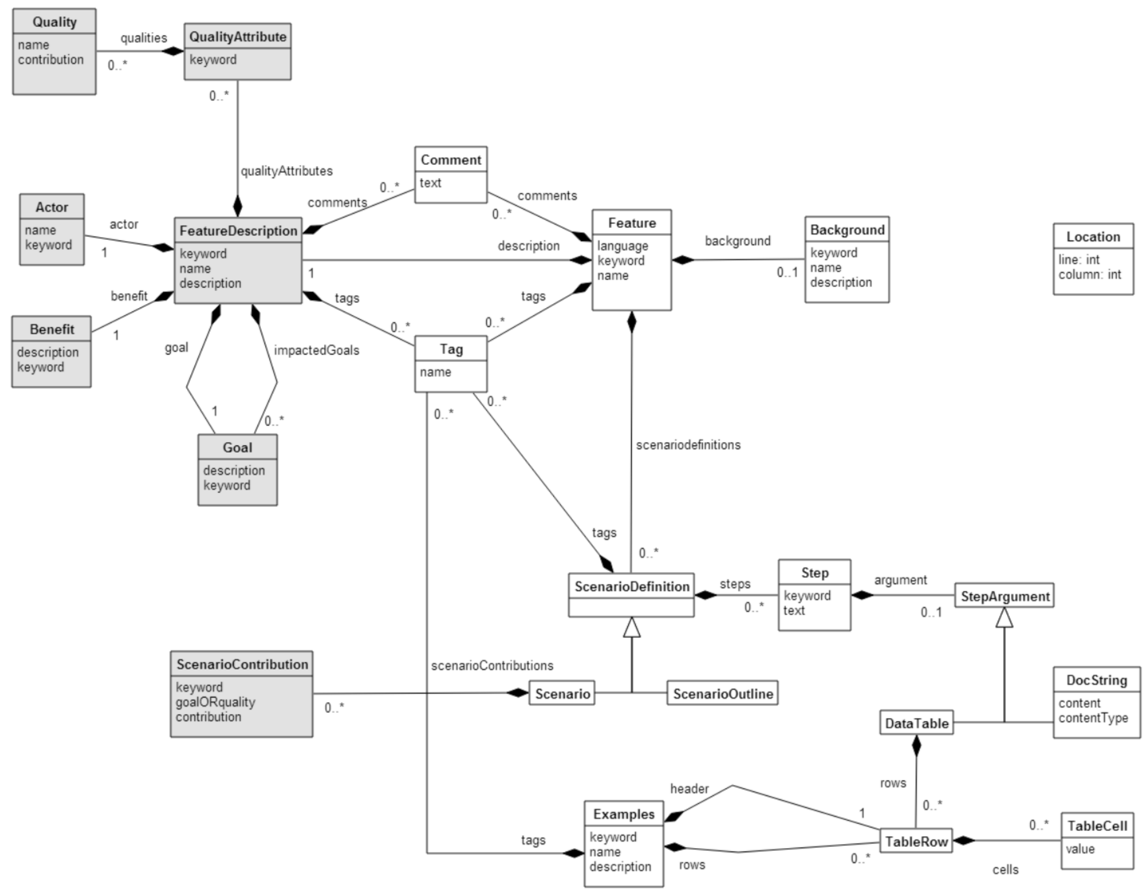


Figure 5.4: Updated Gherkin class model (shaded nodes are new)

The first change replaced the *description* attribute of a *Feature* by a composed object *FeatureDescription*. As any other node in the AST, a *FeatureDescription* contains a *Location* node (that link is not shown to prevent cluttering the diagram).

Recall from chapter 4 that we decided to represent features using the Connextra user story format (reproduced again below)

As a <*Actor* >, **I want** <*Goal* >, **So that** <*Benefit* >

For this reason, *FeatureDescription* contains a reference to an *Actor*, a *Goal* and a *Benefit* node representing the segments within the user story format above. These three new nodes, simply hold the *keyword* matched, which for an actor is ‘As a ’, for a goal is either ‘I want to ’ or ‘I want the ’ and for a benefit is ‘So that ’ (listing 5.1 contains all of these keywords) and the lexeme parsed, which is the remaining content of the line being parsed without keyword and cleared of any spaces in between. With what has been described so far, feature contained in listing 4.1 can be parsed successfully.

The next change introduced a composition between *FeatureDescription* and *Goal* node, to support specifying dependencies between goals. *FeatureDescription* has a collection (possibly empty) of *impacted goals*. This is so that features such as the one in listing 4.2 can be parsed successfully.

Another composition was added between *FeatureDescription* and *QualityAttribute* to support attaching quality constraints to goals. The keywords supported are ‘Without ignoring’ and ‘Constrained by’. We could use the first, whenever we intend to focus attention on attaining those qualities and the second when instead, we prefer to surface the fact that one or more qualities constrain the particular goal being modelled.

The last remaining change, was to add a composition between *Scenario* and *Scenario Contribution* to allow specifying the contribution each *Scenario* can have to the feature’s goal or to a quality.

Listing 5.2 shows an extract of the relevant parts of the alterations to the *Berp* grammar for the Gherkin language. *Berp* uses a syntax similar to BNF [8] and provides support for languages without explicit tokenization rules, such as Gherkin. The symbols $+, ?, *$ have the meaning expected and as used in regular expressions syntax. Tokens are preceded with a # character and some of them can be ignored such as in the cases of comments or spaces.

Code for the parser will be shown later, and is expecting a feature file containing a *Feature*. The production rules can be read as in any context-free grammar. As an example, the first production rule states that a *Feature* can have the following form: a *Feature_Header* composed of an optional *Language* token, optional *Tags*, which are one or more *TagLine* tokens; an optional *Background* (production for background not shown in this listing, as it did not change), a *Feature_Description* and zero or more *Scenario_Definition*. In order to fully understand this production rule we have to do the same for production rules not expanded, that is *Background*, *Feature_Description* and *Scenario_Definition*.

```
...

Feature! := Feature_Header Background? Feature_Description Scenario_Definition*

Feature_Header! := #Language? Tags? #FeatureLine

Feature_Description! := Tags? #AsA_Step #IWant_Step #SoThat_Step WichMayImpact* Quality_Step?

Quality_Step!  := #QualityAttributeLine DataTable

WichMayImpact!  := #WichMayImpact

Scenario_Definition! := Tags? (Scenario | ScenarioOutline)

Scenario! := #ScenarioLine Scenario_Step* ScenarioContribution*

ScenarioContribution! := #ScenarioContribution

Tags! := #TagLine+
...
```

Source Code / Feature 5.2: Changes to gherkin grammar

We now turn attention to the actual Gherkin parser code. Listing 5.3 shows classes involved in the creation of a Parser object. The parser makes use of a *TokenScanner* to read the feature file line by line and create *Tokens* assigned with a *Location* object with details of the current position in the input file.

TokenMatcher has two responsibilities: one is to setup a *GherkinDialect* object – corresponds to the deserialised version of the JSON content listed earlier – that is capable of recognizing the keywords in the default language (English) or in the language specified in the feature file; the other is to match parsed lexemes with tokens returned by *TokenScanner*. Listing 5.3 shows the initial steps in constructing a *Parser* object.

[8]Backus-Naur Form – a widely used notation to specify the syntax of languages

ASTBuilder, as the name implies, is responsible for building the AST based on the token or tokens parsed already. It keeps a stack of parsed nodes and whenever the a production rule has been 'activated' in its entirety a node is popped from the stack and the current AST node replaced with a transformed one. An extract of the transformation function can be seen in listing 5.6

```
\\ somewhere, perhaps on a test file, a parser object is constructed
var parser = new Parser();
var parsingResult = parser.Parse(testFeatureFile);

\\ which in turn, causes this constructor to be invoked
public Parser() : this(new AstBuilder<T>())
{
}

// and this, setting an astBuilder private readonly object
public Parser(IAstBuilder<T> astBuilder)
{
    this.astBuilder = astBuilder;
}

\\ then the Parser's Parse method is invoked
public Feature Parse(string sourceFile)
{
    using (var reader = new StreamReader(sourceFile))
    {
        return Parse(new TokenScanner(reader));
    }
}

// After which, the Parse method in generic base class Parser<Feature> is invoked
public T Parse(ITokenScanner tokenScanner)
{
    return Parse(tokenScanner, new TokenMatcher());
}
```

Source Code / Feature 5.3: Parser construction

Listing 5.4 is the core method where parsing actually occurs. After a set of initialisation instructions, the parser runs a state machine generated by *Berp* when supplied with the gherkin grammar and a template in the language the parser is to be generated in, and in our case that is *.NET/C#*.

The Gherkin parser generator (*Berp*) generates several functions that are used to determine the transition from a state to the next state, based on a matched token. Listing 5.5 shows the matching function when the parser engine is on state 0 and has asked *TokenScanner* for the next *Token*. The function listed correspond to the production rule for a *Feature* (see listing 5.2)

```
// The main loop where parsing occurs
public T Parse(ITokenScanner tokenScanner, ITokenMatcher tokenMatcher)
{
    tokenMatcher.Reset();
    astBuilder.Reset();
    var context = new ParserContext
    {
        TokenScanner = tokenScanner,
        TokenMatcher = tokenMatcher,
        TokenQueue = new Queue<Token>(),
        Errors = new List<ParserException>()
    };

    StartRule(context, RuleType.Feature);
    int state = 0;
    Token token;
    do
    {
        token = ReadToken(context);
        state = MatchToken(state, token, context);
    } while(!token.IsEOF);

    EndRule(context, RuleType.Feature);

    if (context.Errors.Count > 0)
    {
        throw new CompositeParserException(context.Errors.ToArray());
    }

    return GetResult(context);
}
```

Source Code / Feature 5.4: Parser execution

```
// Start
int MatchTokenAt_0(Token token, ParserContext context)
{
    if (Match_Language(context, token))
    {
        StartRule(context, RuleType.Feature_Header);
        Build(context, token);
        return 1;
    }
    if (Match_TagLine(context, token))
    {
        StartRule(context, RuleType.Feature_Header);
        StartRule(context, RuleType.Tags);
        Build(context, token);
        return 2;
    }
    if (Match_FeatureLine(context, token))
    {
        StartRule(context, RuleType.Feature_Header);
        Build(context, token);
        return 3;
    }
    if (Match_Comment(context, token))
    {
        Build(context, token);
        return 0;
    }
    if (Match_Empty(context, token))
    {
        Build(context, token);
        return 0;
    }

    const string stateComment = "State: 0 - Start";
    token.Detach();
    var expectedTokens = new string[] {"#Language", "#TagLine", "#FeatureLine",
                                    "#Comment", "#Empty"};
    var error = token.IsEOF ? (ParserException)new UnexpectedEOFException(token,
                                                                          expectedTokens,
                                                                          stateComment)
        : new UnexpectedTokenException(token, expectedTokens, stateComment);
    if (StopAtFirstError)
        throw error;

    AddError(context, error);
    return 0;
}
```

Source Code / Feature 5.5: Parser execution – Token Matching

```
\\ the end of a production rule has been reached will build a
 ↪ node of the AST
private object GetTransformedNode(AstNode node)
{
 switch (node.RuleType)
 {
    // ... Other case statements ommitted
    case RuleType.WichMayImpact:
    {
        var impactedGoalNode =
          ↪ node.GetToken(TokenType.WichMayImpact);
        return new Goal(GetLocation(impactedGoalNode),
          ↪ impactedGoalNode.MatchedKeyword,
          ↪ impactedGoalNode.MatchedText);
    }
    case RuleType.Feature_Description:
    {
       var tags         = GetTags(node);
       var actorLine    = node.GetToken(TokenType.AsA_Step);
       var actor        = new Actor(GetLocation(actorLine),
         ↪ actorLine.MatchedKeyword, actorLine.MatchedText);
       var goalLine     = node.GetToken(TokenType.IWant_Step);
       var goal         = new Goal(GetLocation(goalLine),
         ↪ goalLine.MatchedKeyword, goalLine.MatchedText);
       var benefitLine  =
         ↪ node.GetToken(TokenType.SoThat_Step);
       var benefit      = new
         ↪ Benefit(GetLocation(benefitLine),
         ↪ benefitLine.MatchedKeyword,
         ↪ benefitLine.MatchedText);

       var impactedGoals =
         ↪ node.GetItems<Goal>(RuleType.WichMayImpact).ToArray();
       var qualities     =
         ↪ node.GetSingle<QualityAttributes>(RuleType.Quality_Step);

       return new FeatureDescription(tags,
         ↪ GetLocation(actorLine), actor, goal, benefit,
         ↪ impactedGoals, qualities);
    }
    case RuleType.Feature:
    {
       var header              =
         ↪ node.GetSingle<AstNode>(RuleType.Feature_Header);
       var tags                = GetTags(header);
       var featureLine         =
         ↪ header.GetToken(TokenType.FeatureLine);
       var featureDescription  =
         ↪ node.GetSingle<FeatureDescription>(RuleType.Feature_Description);
       var background          =
         ↪ node.GetSingle<Background>(RuleType.Background);
       var scenariodefinitions =
         ↪ node.GetItems<ScenarioDefinition>(RuleType.Scenario_Definition).ToArray();
       var description         = GetDescription(header);
       var language            =
         ↪ featureLine.MatchedGherkinDialect.Language;

       return new Feature(tags, GetLocation(featureLine),
         ↪ language, featureLine.MatchedKeyword,
         ↪ featureLine.MatchedText, featureDescription,
         ↪ background, scenariodefinitions,
         ↪ comments.ToArray());
    }
 }
 return node;
}
```

Source Code / Feature 5.6: Parser execution – Building AST nodes

5.2 Gherkin to GRL Translator

The fact that the Gherkin parser returns an AST, allows for other tools to take that intermediate representation and use it to translate Gherkin instances to different formats. Listing 5.7 below contains the relevant section of the body of the translator program from instances of Gherkin to GRL for use in jUCMNav.

```
// Load Default Quality Catalogue
var lastAssignedId = 0;
var defaultQualityCatalogue =
 ↪ QualityCatalogue.BuildDefault(out lastAssignedId);

// Initialise a grlCatalogue
var grlCatalogue =
 ↪ GRLCatalogueFactory.NewGRLCatalog(DEFAULT_CATALOGUE_NAME,
 ↪ DEFAULT_CATALOGUE_DESC, DEFAULT_CATALOGUE_AUTHOR);

var generator = new GRLCatalogueGenerator();

// Avoid parsing features twice by collecting parsing results
var parsingResults = new Ast.Feature[] { };

// Prepend qualities from default catalogue
generator.AppendQualityCatalogue(defaultQualityCatalogue,grlCatalogue);

// Process all feature files passed as arguments
foreach (var featureFiles in args)
{
 var parsingResult =
  ↪ UpdateGRLCatalogueWithFeature(featureFiles, grlCatalogue,
  ↪ generator);
 parsingResults = parsingResults.Concat(new [] { parsingResult
  ↪ } ).ToArray();
}

// Add any dependencies amongst goals. Can only run when all
 ↪ features have been processed
foreach (var parsingResult in parsingResults)
{

     ↪ generator.UpdateGRLCatalogueWithImpactedGoals(grlCatalogue,
     ↪ parsingResult);
}

// Add any global goals. Need to have access to all AST trees
 ↪ to ensure no dependencies between global goals are created
generator.UpdateGRLCatalogueWithGlobalGoals(grlCatalogue,
 ↪ parsingResults);

// Serialise resulting GRL Catalogue
var grlCatalogueAsXmlString =
 ↪ XMLSerializerHelper.SerializeObject(grlCatalogue);
Console.WriteLine(XMLSerializerHelper.RemoveBOM(grlCatalogueAsXmlString));
```

Source Code / Feature 5.7: Translator to GRL – main body

The translator works by populating an object model derived from the XSD schema defined in Appendix C – created using *xsd.net*, a well known *.NET* serialisation tool [9] – in a sequence of logical steps.

These steps are implemented in a *GRLCatalogueGenerator* object that contains the logic required to create goals, softgoals, tasks, links between them and set containment of goals within

[9] see https://msdn.microsoft.com/en-us/library/x6c1kb0s(v=vs.110).aspx for details

actors. The caller, in this case the translator main method, is responsible for orchestrating the calls to the right methods at appropriate times.

The first step consists of populating the GRL catalogue with a default set of quality characteristics and sub-characteristics. In this work we use the ones defined in ISO/IEC 25010:2011, but others could be used as easily.

```
public static string[][] DEFAULT_QUALITY_CATALOGUE_SPEC = {
// Product Quality Charactristics and sub-characteristics
new [] {NextId,"Functional suitability","And",
        NextId,"Functional completeness",
        NextId,"Functional correctness",
        NextId,"Functional appropriateness"},
new [] {NextId,"Performance efficiency","And",
        NextId,"Time behaviour",
        NextId,"Resource utilization",
        NextId,"Capacity"},
new [] {NextId,"Compatibility","And",
        NextId,"Co-existence",
        NextId,"Interoperability"},
...
};
```

Source Code / Feature 5.8: Translator to GRL – default quality set

Listing 5.8 contains an extract of those qualities. We use a multi-dimensional array where each element is another array containing one particular characteristic and zero or more sub-characteristics. The first two elements of that array are the identifier and name; the third element is the type of decomposition to use (we only use *AND* decompositions, but others like *OR* and *XOR* could also be used), and the other elements, processed in groups of two elements at a time, are the sub-characteristics identifier and name. For example, *Compatibility* is a quality with two sub-qualities, namely *Co-Existence* and *Interoperability*. We use a global element identifier to ensure all elements are assigned a unique and incremental id. From this list of qualities we build a *QualityCatalogue* class model as shown in figure 5.5.

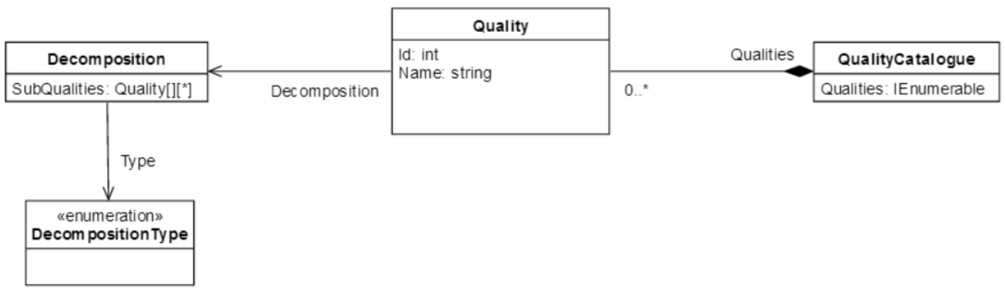


Figure 5.5: Default quality catalogue class model

Note that in the graphs in chapter 4, section 4.2 we remove any parent and child nodes without any outgoing links from this hierarchy to reduce cluttering in the diagrams. That is the reason, only *Interoperability* is shown and not the complete quality hierarchy with parent *Compatibility* and sibling *Co-existence*.

Once default qualities have been added to the catalogue, the next step is to individually process all feature files passed in as arguments to the translator. The translator will invoke the Gherkin parser for each feature file and pass the results to the generator object instructing it to perform a first pass through each of the AST returned by the parser.

The reason we mention first pass, is due to the nature of impacted goals and global constrains or, in other words, the way links are established within the catalogue. The translator has to do a first pass on ASTs so that all intentional elements are created, so that on a second pass, dependencies between elements can be added as they require goals involved in the dependency to be already present in the catalogue.

```
public void UpdateGRLCatalogue(grlcatalog grlCatalogue, Ast.Feature parsingResult)
{
    // Add new actor element or retrieve existing one from container
    var actorElement = AddActorToGRLCatalogue(parsingResult, grlCatalogue);

    // Add new goal (This is the "I want to..." goal of a BDD Feature)
    var goalElement = AddFeatureGoalToGRLCatalogue(parsingResult, grlCatalogue);

    // Attach goal to Actor
    BindElementToActorAndUpdateGRLCatalogue(grlCatalogue, actorElement, goalElement);

    // Are there any non-functionals?
    if (parsingResult.Description != null
        && parsingResult.Description.QualityAttributes != null
        && parsingResult.Description.QualityAttributes.Qualities.Length > 0)
    {
        // Add Qualities
        AddQualitiesToGRLCatalogue(parsingResult, grlCatalogue, actorElement);

        // Add Contribution Links from qualities to goal
        AddQualityGoalContributionsToGRLCatalogue(parsingResult, grlCatalogue, goalElement);
    }

    // Are there any scenarios?
    if (parsingResult.ScenarioDefinitions != null
        && parsingResult.ScenarioDefinitions.Count() > 0)
    {
        // Add contributions from scenarios to goal
        AddScenarioGoalContributionsToGRLCatalogue(parsingResult, grlCatalogue, goalElement);
    }
}
```

Source Code / Feature 5.9: Translator to GRL – update catalogue with AST

Processing each AST includes adding (if not already present) the actor to the catalogue, add the new goal to the catalogue and setting containment of this goal to the actor. If this goal is constrained by any qualities, then those qualities and contributions from each quality to the feature's goal is added to the catalogue. Similar consideration is done if the AST contains nodes for parsed scenarios.

We will not be listing all of these auxiliary methods as they all follow a similar pattern described next (figure 5.10 contains an example of one such methods).

Each of these methods has access to a *GRLElementsContainer*, an implementation of the *Registry* pattern [10] and that creates new and locates existing elements that have been added to the catalogue.

This object is very important, as it allows finding elements already added in the catalogue by name/description which is all that an AST returned by Gherkin has access to. The GRL catalogue instead, requires all elements, and links between them, to have identifiers and use them to establish those links.

Each of the elements to be registered in the catalogue implements an interface, named *IElementWithIdentity*, that ensures they have a valid identifier, name, and description, which is used in registering and locating an element in the registry, but also in code generation to

[10] see http://martinfowler.com/eaaCatalog/registry.html for details

```
private grlcatalogActor AddActorToGRLCatalogue(Ast.Feature
 ↪ parsingResult, grlcatalog grlCatalogue)
{
  var actor = parsingResult.Description.Actor;

  // Create or obtain actor element from container
  var actorExists = false;
  var actorElement =
   ↪ container.RegisterElement<grlcatalogActor>(actor.Name,
   ↪ out actorExists);

  // Only add to catalogue if not already present
  if (!actorExists)
    grlCatalogue.actordef = grlCatalogue.actordef.Union(new[]
     ↪ { actorElement }).ToArray();

  return actorElement;
}
```

Source Code / Feature 5.10: Translator to GRL – add actor to catalogue

produce the GRL catalogue as all elements are required to have those attributes. Listing 5.11 shows the most relevant methods of the registry object.

Once all feature files have gone through the first pass, we invoke the generator's method *UpdateGRLCatalogueWithImpactedGoals* which adds any dependencies amongst goals (if there is any). Figure 5.12 shows the method's implementation.

It is important to note that we avoid adding a dependency if a reverse dependency already exists. For example, if no check was made, then two global goals A and B would create two dependencies $A \rightarrow B$ and $B \rightarrow A$ and that would break the GRL Catalogue consistency.

Last remaining step is to serialize the catalogue object by using simple *.NET* object serialisation techniques.

```
class GRLElementsContainer
{
 private Dictionary<int, IElementWithIdentity> registry;
 private int lastAssignedId = 0;

 public int Add<T>(T value) where T : class, IElementWithIdentity
 {
     if (value == null)
         throw new ApplicationException("Cannot register null object in GRLContainer");
     if (registry.Any(keyval => keyval.Value.name == value.name))
         throw new ApplicationException("Cannot register value as it already exists");
     registry.Add(++lastAssignedId, value);
     return lastAssignedId;
 }

 public T GetElementByName<T>(string name) where T : class, IElementWithIdentity
 {
   if (registry.Any(keyval => keyval.Value.name == name))
     return registry.First(keyval => keyval.Value.name == name).Value as T;
   return null;
 }

 public T RegisterElement<T>(string name, out bool exists) where T : class,
                                                              IElementWithIdentity,
                                                              new()
 {
   T intElement = new T();
   exists = false;
   var existingElement = GetElementByName<T>(name);
   if (existingElement != null)
   {
     exists = true;
     return existingElement;
   }

   var id = Add<IElementWithIdentity>(intElement);
   intElement.id          = id.ToString();
   intElement.name        = name;
   intElement.description = "";
   return intElement;
 }
```

Source Code / Feature 5.11: Translator to GRL – a Registry object for GRL elements

```
foreach (var parsingResult in parsingResults)
{
    var goalDescription = parsingResult.Description.Goal.Description;
    var goalElement = container.GetElementByName<grlcatalogIntentionalelement>(goalDescription);
    if (parsingResult.Description != null
        && HasGlobalTag(parsingResult.Description))
    {
        foreach (var intentionalElement in grlCatalogue.elementdef)
        {
            var otherGoalElement = container.GetElementByName<grlcatalogIntentionalelement>(intenti
            var isGlobalGoal = parsingResults.Any(f => String.Compare(f.Description.Goal.Descriptio

            if (goalElement.id != otherGoalElement.id
            && intentionalElement.type == "Goal"
            && !isGlobalGoal)
            {
                var dependencyElement = BuildDependencyElement(goalElement, otherGoalElement);
                grlCatalogue.linkdef.dependency = grlCatalogue.linkdef.dependency.Union(new[] { dep
            }
        }
    }
}
```

Source Code / Feature 5.12: Translator to GRL – Add global goals

6 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).

Out work helps to address two of the most common requirements risks, which are ambiguous requirements and overlooked stakeholders (Wiegers and Beatty, 2013, p. 20). By viewing requirements as goals of stakeholders that when satisfied produce some identified benefit, and by ensuring all stakeholders interests are reflected on identified features, we reduce the potential for overlooked stakeholders. By producing examples of both functional and non-functional requirements, in the spirit of *Specification by Example*, and representing the latter as constrains to the former, we reduce the ambiguity in requirements elicitation and analysis methods.

Our work is also actual and contributes to major requirements trends in recent years, including the increased use of agile development methods and the evolution of techniques for handling requirements on agile projects; the maturing of tools for assisting with requirements development activities and 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. 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).

Our approach helps to determine requirements and builds on the foundational principles of agile development and BDD by extending Gherkin with constructs to represent actors, goals and their dependencies, constrains and global quality attributes. Using our extension to BDD, we are able to address three types of requirements : functional, non-functional and design constrains in a coherent manner (Leffingwell, 2011).

Documented and clear requirements greatly facilitate system testing. All of these increase chances of delivering high-quality products that satisfy all stakeholders (Wiegers and Beatty, 2013). Specifications in Gherkin use a ubiquitous language – a common, rigorous language used between developers and users and based on the domain they belong to – that fosters communication and clarifies requirements. We benefit from these characteristics and extend its use to the systematic handling of non-functional requirements.

As the software industry became more mature, 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 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).

6.1 Related work

Rosa et al. (2004) present a process-oriented approach defined around basic abstractions and in which functional and non-functional requirements are refined together in a semi-formal way. It defines non-functional properties using a set of abstractions, namely *NF-Attributes*, *NF-Statements* and *NF-Actions* and integrates them in architectures of software systems by associating them with architectural elements that are then refined into concrete implementations.

Araujo and Ribeiro (2005) defines an aspect-oriented agile requirements approach where initial cross-cutting requirements are modelled as coarse-grained scenarios. Scenarios are used as descriptions of desired or existing system behaviour and composition rules are defined to weave coarse-grained aspectual and non-aspectual scenarios to completely describe functionalities.

Also exploring aspect-orientation and looking at reuse from a non-functional (quality) perspective, Leite et al. (2005) combine ideas from reuse, from goal-oriented requirements, aspect-oriented programming and quality management, to obtain a goal-driven process to enable a quality-based reusability of software artefacts. The authors detail propose a goal-oriented representation language to support quality reusability introducing a representation for functions, topics, quality types, pre-conditions, pointcuts (relations among functions, topics and quality types), contribution structures and quality operationalizations.

Franch, 1998 presented a proposal for putting non-functional information of software systems into software architectures. Non-functionality is described by means of a notation called *NoFun*, which allows to introduce non-functional attributes of software, to give values to these attributes in component and connectors, and to formulate non-functional requirements in terms of these attributes.

There are three research streams that have influenced our work. The first one is closely related to ours, but comes from a different angle and is limited to some types of non-functional requirements (Barmi and Ebrahimi, 2011). The authors propose *ProBDD*, an extension to BDD, in which probabilistic non-functional requirements– such as safety, security, and performance – are specified in BDD scripts using specification patterns for probabilistic properties with a structured English grammar. The proposed extended BDD scripts can be parsed and executed with randomly generated input data and test results can verify the specified non-functional requirements fulfilment.

Their work differs from ours as it is restricted to non-functional requirements that can be written as probabilistic statements, that is, those that can be quantified and can be expressed in a probabilistic form. Also, their focus is on automated testing and validation of non-functional requirements while our approach is applicable to all non-functional requirements types, is independent of taxonomy chosen and is intended to foster analysis, detection and management of dependencies and conflicts among non-functional requirements.

The second work, describes the building blocks of a lightweight agile methodology (called *NORMAP*) for identifying, linking, and modelling functional and non-functional requirements aimed at improving software quality and support agility (Farid and Mitropoulos, 2012). The authors proposed *Agile Use Cases (AUCs)* – hybrid of use cases and agile user stories – to model functional requirements, *Agile Loose Cases (ALCs)* – loosely-defined agile NFRs – and *Agile Choose Cases (ACCs)* to model potential solutions (operationalizations) to non-functional requirements. The three artefacts are combined in a visual framework to promote agile modelling of non-functional requirements (primarily) and how they are linked to functional requirements. The authors also enhanced the agile user story card to a new *W8* Story Card Model that captures functional or non-functional agile requirements that has served as inspiration to our proposed additional constructs in Gherkin.

Finally, the foundational work laid by the *NFR Framework* Chung et al., 1999, that first introduced the concept of ‘softgoal’, which represents a goal that has no clear-cut definition and/or criteria as to whether it is satisfied or not. The *NFR Framework* allowed systematic handling of non-functional requirements and later influenced other goal-oriented approaches, such as GRL Amyot, 2003 that we integrate in our research.

6.2 Further research

Our work has introduced changes in the Gherkin parser but has largely ignored its impact on the compiler. Further research is needed to adapt the compiler for these new constructs. This also implies the need to adapt existing tools, such as Cucumber [1] or SpecFlow [2] for the new capabilities in order for those tools to assist in producing specifications that include non-functional constrains alongside functional requirements.

These tools could also play a role in interfacing between Gherkin and GRL by, for example, causing the successful execution of test cases to mark corresponding *Tasks* in GRL as satisfied, or supporting typical editor features (e.g. intellisense and code completion, syntactic analysis, etc.)

Further research is also required in applying the methodology to larger case studies, preferably originated in complex industrial settings where the suitability (or not) of the practices and techniques described in this work, can be assessed.

At the moment, we only support forward generation of GRL models from Gherkin specifications. In an ideal scenario, it would also be possible to do the reverse, that is, start with a GRL model of goals and softgoals, goal decompositions and dependencies among them, a set of actors, tasks and contributions to goals, and generate one or more Gherkin specifications.

With those two generation models in place, it would be easier to support full traceability and navigation between Gherkin and GRL models, offering the capability of doing updates in place where only modified parts of one model are updated in the other.

Another area of future research consists in exploring interaction between our work and more formal and semi-formal languages that explore modelling approaches for capturing non-functional requirements. In one of such approaches, Li et al. (2014) propose a modelling language for non-functional requirements that views them as requirements over qualities, mapping a software-related domain to a quality space. Their methodology refines informal non-functional requirements elicited from stakeholders, resulting in unambiguous, de-idealized, and measurable requirements. This would address a particular limitation with our approach, in which nothing intrinsic to improves the capacity of requirements being measured or not. If such research is carried out, it is important to investigate the extent to which actors and goals satisfaction algorithms can be adapted to support requirements defined in such languages.

Another possible route for research would be to investigate the extent to which our work could be integrated with UCM – a scenarios-based notation for gathering functional requirements such as operational or architectural requirements, supporting dynamic refinement at the behavioural and structural level (Roy, 2007) – by exploring how scenarios in BDD can be mapped to their equivalent entities in UCM and determine how activities, tasks or functions within a system can be traced back to original BDD scenarios and be part of goal reasoning in GRL.

[1] see https://cucumber.io/ for details

[2] see http://www.specflow.org/ for details

Our proposal does not solve completely all enunciated issues with the treatment of non-functional requirements in software development projects. However, our proposal is an effective step towards the explicit treatment of this kind of properties in agile software development, and BDD in particular, and establishes a defined agenda for research in this area.

Appendices

A Case Study

The table below lists all original functional requirements contained in the *PROMISE* dataset. Note that for easy of referencing the tables below retain original requirement identifiers.

Table A.1: Meeting scheduler original functional requirements (*PROMISE*, 2015)

Requirement ID	Functional Requirement Description
R 48	The product shall record meeting entries
R 49	The product will notify employees of meeting invitations
R 50	The product shall have the ability to send meeting reminders to employees
R 51	The product shall assign the organizers contact information to each meeting they create
R 52	The product will record meeting acknowledgements
R 53	The product shall store new conference rooms
R 54	The product shall update existing conference rooms
R 55	The product will be able to delete conference rooms
R 56	The product shall be able to store new room equipment
R 57	The product will update existing room equipment
R 58	The product shall be able to delete room equipment
R 59	The product shall allow an organizer to invite other employees to meetings
R 60	Each time a conference room is reserved the conference room schedule shall be updated to reflect the time and date of the reservation
R 61	The product shall record the transportation status of equipment reserved
R 62	The product shall display a map of the company building showing conference room locations
R 63	The product shall record updated meeting agendas
R 64	The product shall send a meeting confirmation to the meeting organizer
R 65	The product shall display room equipment according to search parameters
R 66	The product shall display conference rooms according to search parameters
R 67	The product shall record different meeting types

Continued on next page

Table A.1 – *Continued from previous page*

Requirement ID	Functional Requirement Description
R 68	The product shall record all the equipment that has been reserved
R 69	The product shall notify building personnel of equipment transport requests
R 70	The product will allow privileged users to view meeting schedules in multiple reporting views
R 71	The product shall be able to send meeting notifications via different kinds of end-user specified methods
R 72	The product shall have a customizable Look and Feel
R 73	The product shall have an intuitive user interface
R 74	The product will display an available status for unreserved conference rooms

The table below lists all the original non-functional requirements for the *meeting scheduler* project included in the *PROMISE* dataset, including the classification proposed by Li et al. in Li et al., 2014.

Table A.2: Meeting scheduler original non-functional requirements (*PROMISE*, 2015)

Requirement ID	Classification	Non-functional Requirement Description
R 138	FR+QR	The product must work with most database management systems (DBMS) on the market whether the DBMS is colocated with the product on the same machine or is located on a different machine on the computer network.
R 139	FR+QR	The product will function alongside server software on any operating system where the Java runtime can be installed.
R 140	FR	The product will require collaboration with a database management system (DBMS). The DBMS may be located on the same machine as the product or on a separate machine residing on the same computer network.
R 141	QR	The product must make use of web/application server technology. Open source examples include Apache web server Tomcat and the JBoss application server.
R 142	FR	A database management system such as Oracle DB2 MySql or HSQL will need to be integrated with the product
R 143	FR+QR	The product's Look and Feel shall be able to incorporate aspects of the customer's organization such as logo branding and identity

Continued on next page

Table A.2 – *Continued from previous page*

Requirement ID	Classification	Non-functional Requirement Description
R 144	QR	The product shall have a conservative and professional appearance
R 145	QR	The product shall make the users want to use it. 80\% of the users surveyed report they are regularly using the product after the first 2 weeks postlaunch
R 146	QR	The product shall be easy to use. 90\% of users will be able to successfully reserve a conference room within 5 minutes of product use
R 147	FR+QR	The product shall give users feedback when necessary. 80\% of the users surveyed report that the product accurately confirms their actions
R 148	FR+QR	The product shall allow the user to select a chosen language from one of the target market countries
R 149	FR+QR	The product shall allow for customization of start page and views preferences
R 150	QR	An employee will be able to successfully use the product within a few minutes. After informally navigating the product for less than 15 minutes users shall be able to successfully setup meetings and reserve conference rooms
R 151	QR	The product shall use a standard navigation menu familiar to most web users
R 152	FR+QR	The product shall allow for intuitive searching of available conference rooms
R 153	QR	The product shall conform to the Americans with Disabilities Act
R 154	QR	The response shall be fast enough to avoid interrupting the user's flow of thought. 90\% of tasks shall complete within 5 seconds. 98\% of tasks shall complete within 8 seconds
R 155	QR	Aside from server failure the software product shall achieve 99.99\% up time
R 156	FR+QR	The product shall create an exception log of problems encountered within the product for transmission to our company for analysis and resolution

Continued on next page

Table A.2 – *Continued from previous page*

Requirement ID	**Classification**	**Non-functional Requirement Description**
R 157	QR	The product shall be capable of handling up to 1 000 concurrent requests. This number will increase to 2 000 by Release 2. The concurrency capacity must be able to handle peak scheduling times such as early morning and late afternoon hours
R 158	QR	The product shall be able to process 10 000 transactions per hour within two years of its launch. This number will increase to 20 000 by Release 2
R 159	QR	The product shall be expected to operate for at least 5 years for each customer installation
R 160	DA	The product shall be used in office environments
R 161	FR+QR	The product must be able to interface with any HTML browser. The product shall transmit data between the user and the product without problems. The product shall display HTML properly in 80\% of all HTML browsers tested (minimum is to test 8 browsers)
R 162	FR+QR	The product must be able to interface with various database management systems. The product shall communicate successfully with the database management system on 100\% of all transactions.
R 163	FR+QR	The product must be able to interface with various email servers. The product shall be able to send email
R 164	FR	The product shall be available for distribution via the Internet as a binary or ZIP file
R 165	FR	The product shall be available for distribution as a packaged CD
R 166	FR	The product will be available for licensing as a one-server two-five servers or five-or-more servers license
R 167	QR	A new user must be able to be added to the system within 10 minutes.90\% of new users are able to log into the system within 10 minutes
R 168	QR	The product must be designed using Design Patterns and coding best practices. 90\% of maintenance software developers are able to integrate new functionality into the product with 2 working days

Continued on next page

Table A.2 – *Continued from previous page*

Requirement ID	**Classification**	**Non-functional Requirement Description**
R 169	QR	The product must be highly configurable for use with various database management systems for the end users. 80\% of end users are able to integrate new database management systems with the product without changing the product's software code
R 170	QR	Maintenance releases will be offered to customers once a year. The releases shall take no longer than 5 minutes for 80\% of the customer base to install
R 171	FR+QR	Every registered user will have access to the product's support site via the Internet. 70\% of registered users shall find a solution to their problem within 5 minutes of using the support site
R 172	QR	The customer shall be able to easily integrate new building maps with the product throughout the product's lifecycle. Integration of new maps with the product shall be possible with little to no support from product support staff
R 173	FR+QR	The product shall be translated into foreign languages other than the target market countries languages in future releases. For each emerging target market it shall take no more than 5 days to configure the product for that market's language
R 174	FR+QR	The product is expected to integrate with multiple database management systems. The product will operate with Oracle SQL Server DB2 MySQL HSQL and MS Access
R 175	QR	The product shall be able to be installed in any operating environment within 2 days
R 176	CF+QR	The product shall ensure that only company employees or external users with company-approved user IDs may have product access. 100\% of all logons shall be by either company employees or external users with company-approved user IDs
R 177	CF+QR	Only managers are able to perform search queries for reservations by user. 100\% of all search queries for reservations by user shall be from logons of only management logons

Continued on next page

Table A.2 – *Continued from previous page*

Requirement ID	**Classification**	**Non-functional Requirement Description**
R 178	FR+QR	The product shall ensure that the database's data corresponds to the data entered by the user. Each conference room reservation in the system will contain the same information as was entered by the user
R 179	FR+QR	The product shall prevent the input of malicious data. The product and/or adjacent hardware/software systems data shall remain 100\% uncorrupted each time malicious data is input into the product
R 180	QR	The product and/or dependent databases/filesystems shall remain operational as a result of the input of malicious data. The product and/or dependent databases/filesystems shall not crash 100\% of the time that malicious data is input into the product
R 181	FR+QR	The product shall have the ability to receive automatic software updates as new threats emerge. 100\% of customers will be able to receive automatic software updates transmitted to the installed product
R 182	QR	The language of the product shall accommodate all of the target market countries. 90\% of survey respondents from each target market country will find that the language of the product is acceptable
R 183	FR+QR	The product shall be able to display calendar dates and times according to the user's locale. 90\% of survey respondents from each target market country will find that the date and time formatting of the product is correct
R 184	QR	The product must be developed with the J2SE/J2EE programming language libraries

Table A.3: Meeting scheduler mapping of functional to non-functional requirements (*PROMISE*, 2015)

	Non-Functional Requirements									
Req ID	R 138	R 140 (*)	R 142 (*)	R 146	R 149	R 152	R 162	R 163	R 168	R 178
R 48	I	I	I				I			
R 49										
R 50										
R 51										
R 52										
R 53	I	I	I				I			
R 54	I	I	I				I			
R 55	I	I	I				I			
R 56	I	I	I				I			
R 57	I	I	I				I			
R 58	I	I	I				I			
R 59										
R 60				O,P						O
R 61	I	I	I				I			
R 62						L			MA,MO	
R 63	I	I	I				I			
R 64								I		
R 65										
R 66										
R 67	I	I	I				I			
R 68	I	I	I				I			
R 69										
R 70	I	I	I		AS		I			
R 71								I		
R 72					AS					
R 73					L,AS					
R 74				O,P		L				

Legend: In the above, initials represent qualities

O	Operability
P	Pleasure
I	Interoperability
L	Learnability
MA	Maintainability
MO	Modifiability
AS	user interface aesthetic

(*) Li et al. (2014) marked these as Functional requirements and we do not consider them in our work

A.1 Case study requirements in BDD format

In this section, we list the remaining feature files corresponding to the functional requirements in the *meeting scheduler* case study and not addressed previously in chapter 4, section 4.2. Note that in some listings the lines have been wrapped and a \— used to show where the line breaks, to improve visibility but these line breaks should be removed if the text is to be submitted through Gherkin parser.

```
Feature:    R53 - The product shall store new conference rooms

            As a Facilities Manager
            I want to record new conference rooms
            So that I can keep my facilities inventory updated
            Which may impact update existing conference rooms
            Which may impact delete conference rooms

Constrained by:
            | Interoperability | Help |

# This scenario does not include a contribution.
# Therefore, a default contribution from scenario to goal will be added
Scenario:   Record conference room with minimum details
            Given the application has been started
            And I choose to add new conference room
            When I specify a conference room name, building identifier and floor
            And I proceed to save the conference room
            Then the conference room should be recorded
@NFR
Scenario:   Conference rooms can be recorded in most DBMS
            Contributing to help Interoperability
@NFR
Scenario:   100% of all transactions recording conference rooms in DBMS are successful
            Contributing to help Interoperability
```

Source Code / Feature A.1: R53 – The product shall store new conference rooms

```
Feature:    R54 - The product shall record meeting entries

            As a Facilities Manager
            I want to update existing conference rooms
            So that I can keep my facilities inventory updated

Constrained by:
            | Interoperability | Help |

Scenario:   Update conference room fields other than name

Scenario:   Product does not allow updating conference room field name

@NFR
Scenario:   Conference rooms can be updated in most DBMS
            Contributing to help Interoperability

@NFR
Scenario:   100% of all transactions updating conference rooms in DBMS are successful
            Contributing to help Interoperability
```

Source Code / Feature A.2: R54 – The product shall record meeting entries

```
Feature:    R55 – The product will be able to delete conference rooms

            As a Facilities Manager
            I want to delete conference rooms
            So that I can keep my facilities inventory updated

Constrained by:
            | Interoperability | Help |

Scenario:   Delete conference room not associated with a meeting succeeds

Scenario:   Not possible to delete conference room with meetings associated

@NFR
Scenario:   Conference rooms can be deleted from most DBMS
            Contributing to help  Interoperability

@NFR
Scenario:   100% of all transactions deleting conference rooms in DBMS are successful
            Contributing to help  Interoperability
```

Source Code / Feature A.3: R55 – The product will be able to delete conference rooms

```
Feature:    R56 – The product shall be able to store new room equipment

            As a Facilities Manager
            I want to store new room equipment
            So that I can keep my conference rooms equipment inventory updated
            Which may impact update existing room equipment
            Which may impact delete room equipment

Constrained by:
            | Interoperability | Help |

Scenario:   Record room equipment with minimum details

Scenario:   Record room equipment with complete details

@NFR
Scenario:   Room equipment can be stored in most DBMS
            Contributing to help  Interoperability

@NFR
Scenario:   100% of all transactions storing room equipment in DBMS are successful
            Contributing to help  Interoperability
```

Source Code / Feature A.4: R56 – The product shall be able to store new room equipment

```
Feature:    R57 – The product will update existing room equipment

            As a Facilities Manager
            I want to update existing room equipment
            So that I can keep my conference rooms equipment inventory updated

Constrained by:
            | Interoperability | Help |

@NFR
Scenario:   Room equipment can be updated in most DBMS
            Contributing to help  Interoperability

@NFR
Scenario:   100% of all transactions updating room equipment in DBMS are successful
            Contributing to help  Interoperability
```

Source Code / Feature A.5: R57 – The product will update existing room equipment

```
Feature:    R58 – The product shall be able to delete room equipment

            As a Facilities Manager
            I want to delete room equipment
            So that I can keep my conference rooms equipment inventory updated

Constrained by:
            | Interoperability | Help |

Scenario:   Delete room equipment not associated with a meeting succeeds

# This scenario shows that by producing examples we can uncover other nfrs
Scenario:   Delete room equipment with a meeting associated notifies meeting organiser

@NFR
Scenario:   Notifications of deleted room equipment can be sent through \
            various email servers
            Which helps Interoperability

@NFR
Scenario:   Room equipment can be deleted from most DBMS
            Contributing to help  Interoperability

@NFR
Scenario:   100% of all transactions deleting room equipment from DBMS are successful
            Contributing to help  Interoperability
```

Source Code / Feature A.6: R58 – The product shall be able to delete room equipment

```
Feature:    R60 – Each time a conference room is reserved the conference room schedule \
            shall be updated to reflect the time and date of the reservation

            As a Meeting Organiser
            I want to reserve a conference room and update the schedule with the \
            meeting’s date and time
            So that I prevent double bookings

Constrained by:
            | Operability | Help |
            | Pleasure    | Help |
@NFR
Scenario:   Updated conference room schedule data in the database corresponds to the \
            data entered by the user
            Which helps Operability

@NFR
Scenario:   It is easy to reserve a conference room and update the schedule with the \
            meeting’s date and time
            Which helps Operability
            Contributing to help Pleasure

@NFR
Scenario:   90% of users are able to reserve a conference room within 5 minutes of use
            Which helps Operability
```

Source Code / Feature A.7: R60 – Each time a conference room is reserved the schedule shall be updated

```
Feature:    R61 - The product shall record the transportation status of equipment reserved

            As a Meeting Organiser
            I want to record the transportation status of equipment reserved
            So that I can verify when the equipment is available

Constrained by:
            | Interoperability | Help |

@NFR
Scenario:   Transportation status of equipment reserved can be recorded in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording transportation status of equipment \
            reserved in DBMS are successful
            Which helps Interoperability
```

Source Code / Feature A.8: R61 – The product shall record the transportation status of equipment reserved

```
Feature:    R62 - The product shall display a map of the company building showing \
            conference room locations

            As a Meeting Organiser
            I want to view a map of the company building
            So that I can see conference room locations and choose the best location \
            for the meeting

Constrained by:
            | Learnability    | Help |
            | Maintainability | Help |
            | Modifiability   | Help |

@NFR
Scenario:   Product allows intuitive searching of available conference rooms
            Which helps Learnability

@NFR
Scenario:   Product is designed using Design Patterns and coding best practices
            Which makes Maintainability

@NFR
Scenario:   90% of maintenance software developers are able to integrate new \
            functionality into the product with 2 working days
            Which makes Modifiability
```

Source Code / Feature A.9: R62 – The product shall display a map of the company building

```
Feature:    R64 - The product shall send a meeting confirmation to the meeting organizer

            As a Meeting Organiser
            I want to receive a meeting confirmation
            So that I can be sure the meeting has been recorded

Constrained by:
            | Interoperability | Help |

@NFR
Scenario:   Meeting confirmations can be sent through various email servers
            Which helps Interoperability
```

Source Code / Feature A.10: R64 – The product shall send a meeting confirmation to the meeting organizer

```
Feature:    R68 - The product shall record all the equipment that has been reserved

            As a Meeting Organiser
            I want to record all the reserved equipment
            So that I can be sure all equipment will be available for the meeting

Constrained by:
            | Interoperability | Help |

Scenario:   Reserve projector for a meeting is recorded

Scenario:   Reserve projector and board for a meeting is recorded

@NFR
Scenario:   All the reserved equipment can be recorded in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording all the reserved equipment \
            in DBMS are successful
            Which helps Interoperability
```

Source Code / Feature A.11: R68 – The product shall record all the equipment that has been reserved

```
Feature:    R70 - The product will allow privileged users to view meeting schedules in \
            multiple reporting views

            As a Privileged User
            I want to view meeting schedules in multiple reporting views
            So that I can control schedules

Constrained by:
            | Interoperability          | Help         |
            | user interface aesthetics | SomePositive |

Scenario:   Display monthly view of scheduled meetings

Scenario:   Display view of scheduled meetings for a given conference room

Scenario:   Display view of scheduled meetings organised by a specific individual

@NFR
Scenario:   All meeting schedules can be stored in most DBMS
            Which helps Interoperability

@NFR
Scenario:   100% of all transactions recording meeting schedules in DBMS are successful
            Which helps Interoperability

@NFR
Scenario:   Multiple reporting views can be customised
            Which helps user interface aesthetics
```

Source Code / Feature A.12: R70 – The product will allow privileged users to view meeting schedules

```
Feature:    R71 - The product shall be able to send meeting notifications via different \
            kinds of end-user specified methods

            As a Meeting Organiser
            I want to send meeting notifications via different kinds of end-user methods
            So that I can be sure I can contact all meeting attendees

Constrained by:
            | Interoperability | Help |

Scenario:   Send email notification

Scenario:   Send messaging notification

Scenario:   Meeting confirmations can be sent via different kinds of end-user specified \
            methods and through various email servers
            Which helps Interoperability
```

Source Code / Feature A.13: R71 – The product shall be able to send meeting notifications...

```
Feature:    R72 - The product shall have a customizable look and feel

            @GLOBAL
            As a Procurement Manager
            I want to customise the product's look and feel
            So that I can tailor the product to different departments and teams

Constrained by:
            | user interface aesthetics | SomePositive |

@NFR
Scenario:   Product's look and feel can be customised
            Which helps user interface aesthetics
```

Source Code / Feature A.14: R72 – The product shall have a customizable look and feel

```
Feature:    R74 - The product will display availability for unreserved conference rooms

            As a Meeting Organiser
            I want to see status for all unreserved conference rooms
            So that I am able to chose the best room for a meeting

Constrained by:
            | Learnability | Help |
            | Operability  | Help |
            | Pleasure     | Help |

@NFR
Scenario:   Product allows intuitive searching of all unreserved conference rooms
            Which helps Learnability

@NFR
Scenario:   90% of users are able to see status for all unreserved conference rooms \
            within 5 minutes of product use
            Which helps Operability

Scenario:   It is easy to see status for all unreserved conference rooms
            Which helps Operability
            Contributing to help Pleasure
```

Source Code / Feature A.15: R74 – The product will display availability for unreserved conference rooms

B Gherkin Grammar

This appendix contains listings of the original [1] and updated Gherkin grammars in their entirety.

[1] original extracted from https://github.com/cucumber/gherkin3/blob/master/gherkin.berp

```
[
        Tokens -> #Empty, #Comment, #TagLine, #FeatureLine,
          ↪ #BackgroundLine, #ScenarioLine,
          ↪ #ScenarioOutlineLine, #ExamplesLine, #StepLine,
          ↪ #DocStringSeparator, #TableRow, #Language
        IgnoredTokens -> #Comment,#Empty
        ClassName -> Parser
        Namespace -> Gherkin
]


Feature! := Feature_Header Background? Scenario_Definition*
Feature_Header! := #Language? Tags? #FeatureLine
  ↪ Feature_Description

Background! := #BackgroundLine Background_Description
  ↪ Scenario_Step*

// we could avoid defining Scenario_Definition, but that would
  ↪ require regular look-aheads, so worse performance
Scenario_Definition! := Tags? (Scenario | ScenarioOutline)

Scenario! := #ScenarioLine Scenario_Description Scenario_Step*

ScenarioOutline! := #ScenarioOutlineLine
  ↪ ScenarioOutline_Description ScenarioOutline_Step*
  ↪ Examples_Definition+
// after the first "Examples" block, interpreting a tag line
  ↪ is ambiguous (tagline of next examples or of next
  ↪ scenario)
// because of this, we need a lookahead hint, that connects
  ↪ the tag line to the next examples, if there is an examples
  ↪ block ahead
Examples_Definition!
  ↪ [#Empty|#Comment|#TagLine->#ExamplesLine]:= Tags? Examples
Examples! := #ExamplesLine Examples_Description #TableRow
  ↪ #TableRow+

Scenario_Step := Step
ScenarioOutline_Step := Step

Step! := #StepLine Step_Arg?
Step_Arg := (DataTable | DocString)

DataTable! := #TableRow+
DocString! := #DocStringSeparator #Other* #DocStringSeparator

Tags! := #TagLine+

Feature_Description := Description_Helper
Background_Description := Description_Helper
Scenario_Description := Description_Helper
ScenarioOutline_Description := Description_Helper
Examples_Description := Description_Helper

// we need to explicitly mention comment, to avoid merging it
  ↪ into the description line's #Other token
// we also eat the leading empty lines, the tailing lines are
  ↪ not removed by the parser to avoid lookahead, this has to
  ↪ be done by the AST builder
Description_Helper := #Empty* Description? #Comment*
Description! := #Other+
```

Source Code / Feature B.1: Original gherkin grammar

```
[
        Tokens -> #Empty, #Comment, #TagLine, #FeatureLine,
          ↪ #BackgroundLine, #ScenarioLine,
          ↪ #ScenarioOutlineLine, #ExamplesLine, #StepLine,
          ↪ #DocStringSeparator, #TableRow, #Language,
          ↪ #WichMayImpact, #ScenarioContribution,
          ↪ #QualityAttributeLine, #AsA_Step, #IWant_Step,
          ↪ #SoThat_Step
        IgnoredTokens -> #Comment,#Empty
        ClassName -> Parser
        Namespace -> Gherkin
]

Feature! := Feature_Header Background? Feature_Description
  ↪ Scenario_Definition*
Feature_Header! := #Language? Tags? #FeatureLine

Background! := #BackgroundLine Background_Description
  ↪ Scenario_Step*

Feature_Description! := Tags? #AsA_Step #IWant_Step
  ↪ #SoThat_Step WichMayImpact* Quality_Step?
Quality_Step!        := #QualityAttributeLine DataTable

WichMayImpact!                := #WichMayImpact

// we could avoid defining Scenario_Definition, but that would
  ↪ require regular look-aheads, so worse performance
Scenario_Definition! := Tags? (Scenario | ScenarioOutline)

Scenario! := #ScenarioLine Scenario_Step*
  ↪ ScenarioContribution*

ScenarioContribution! := #ScenarioContribution

ScenarioOutline! := #ScenarioOutlineLine
  ↪ ScenarioOutline_Description ScenarioOutline_Step*
  ↪ Examples_Definition+
// after the first "Examples" block, interpreting a tag line
  ↪ is ambiguous (tagline of next examples or of next
  ↪ scenario)
// because of this, we need a lookahead hint, that connects
  ↪ the tag line to the next examples, if there is an examples
  ↪ block ahead
Examples_Definition!
  ↪ [#Empty|#Comment|#TagLine->#ExamplesLine]:= Tags? Examples
Examples! := #ExamplesLine Examples_Description #TableRow
  ↪ #TableRow+

Scenario_Step := Step
ScenarioOutline_Step := Step

Step! := #StepLine Step_Arg?
Step_Arg := (DataTable | DocString)

DataTable! := #TableRow+
DocString! := #DocStringSeparator #Other* #DocStringSeparator

Tags! := #TagLine+

Background_Description := Description_Helper
Scenario_Description := Description_Helper
ScenarioOutline_Description := Description_Helper
Examples_Description := Description_Helper

// we need to explicitly mention comment, to avoid merging it
  ↪ into the description line's #Other token
// we also eat the leading empty lines, the tailing lines are
  ↪ not removed by the parser to avoid lookahead, this has to
  ↪ be done by the AST builder
Description_Helper := #Empty* Description? #Comment*
Description! := #Other+
```

Source Code / Feature B.2: Updated gherkin grammar

C GRL Catalogue XSD Schema

The listing below, contains the XSD schema that is used to generate a class model used by the Gherkin to GRL translator described in chapter 5, section 5.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="grl-catalog">
  <xs:complexType>
   <xs:sequence>
    <!-- Intentional Elements Definitions -->
        <xs:element name="element-def">
         <xs:complexType>
          <xs:sequence>
           <xs:element maxOccurs="unbounded" minOccurs="0" name="intentional-element">
            <xs:complexType>
                 <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="type" type="xs:string" use="required"/>
                 <xs:attribute name="decompositiontype" type="xs:string" use="required"/>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="description" type="xs:string" use="required"/>
                </xs:complexType>
           </xs:element>
          </xs:sequence>
     </xs:complexType>
    </xs:element>
    <!-- End of Intentional Elements Definitions -->
    <!-- GRL Links Definitions -->
    <xs:element name="link-def">
         <xs:complexType>
          <xs:sequence>
          <!-- Dependency -->
           <xs:element maxOccurs="unbounded" minOccurs="0" name="dependency">
            <xs:complexType>
                 <xs:attribute name="dependeeid" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="dependerid" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="description" type="xs:string" use="required"/>
                </xs:complexType>
           </xs:element>
           <!-- Decomposition -->
           <xs:element maxOccurs="unbounded" minOccurs="0" name="decomposition">
            <xs:complexType>
             <xs:attribute name="srcid" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="destid" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="description" type="xs:string" use="required"/>
                </xs:complexType>
           </xs:element>

           <! -- continues next page -->
```

Source Code / Feature C.1: GRL Catalogue XSD Schema – Part one

```
        <! -- continued from previous page -->

           <!--Contribution -->
           <xs:element maxOccurs="unbounded" minOccurs="0" name="contribution">
                <xs:complexType>
                 <xs:attribute name="srcid" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="destid" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="contributiontype" type="xs:string" use="required"/>
                 <!-- New. Optional for backward compatibility. -->
                 <xs:attribute name="quantitativeContribution" type="xs:integer" use="optional"/>
                 <xs:attribute name="correlation" type="xs:boolean" use="required"/>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="description" type="xs:string" use="required"/>
                </xs:complexType>
           </xs:element>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
        <!-- End of GRL Link Definitions -->
    <!-- Actor Definitions -->
        <xs:element name="actor-def">
         <xs:complexType>
          <xs:sequence>
           <xs:element maxOccurs="unbounded" minOccurs="0" name="actor">
            <xs:complexType>
                 <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="description" type="xs:string" use="required"/>
                </xs:complexType>
           </xs:element>
          </xs:sequence>
     </xs:complexType>
    </xs:element>
    <!-- End of Actor Definitions -->
        <!-- Actor-IE Containment Links -->
        <xs:element name="actor-IE-link-def">
         <xs:complexType>
          <xs:sequence>
           <xs:element maxOccurs="unbounded" minOccurs="0" name="actorContIE">
            <xs:complexType>
                 <xs:attribute name="actor" type="xs:positiveInteger" use="required"/>
                 <xs:attribute name="ie" type="xs:positiveInteger" use="required"/>
                </xs:complexType>
           </xs:element>
          </xs:sequence>
     </xs:complexType>
    </xs:element>
    <!-- End of Actor-IE Containment Links -->
   </xs:sequence>
   <!-- GRL catalogue attributes -->
   <xs:attribute name="catalog-name" type="xs:string" use="required"/>
   <xs:attribute name="description" type="xs:string" use="required"/>
   <xs:attribute name="author" type="xs:string" use="required"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Source Code / Feature C.2: GRL Catalogue XSD Schema – Part two

Bibliography

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

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

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

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

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.

Araujo, J. and J. C. Ribeiro (2005). ‘Towards an Aspect-Oriented Agile Requirements Approach’. In: *Proceedings of the Eighth International Workshop on Principles of Software Evolution*. IWPSE ’05. Washington, DC, USA: IEEE Computer Society, pp. 140–143.

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.

Cohn, M. ((2015). *Non-functional Requirements as User Stories*. Ed. by Mountaingoatsoftware.com. URL: `https://www.mountaingoatsoftware.com/blog/non-functional-requirements-as-user-stories`.

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.

Davies, R. (2010). 'Agile Non-Functional Requirements: Do User Stories Really Help'. In: *Methods & Tools* 18.4.

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

Defense (DoD), D. of (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.

Farid, W. and F. Mitropoulos (2012). 'Novel lightweight engineering artifacts for modeling non-functional requirements in agile processes'. In: *Southeastcon, 2012 Proceedings of IEEE*, pp. 1–7.

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.

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.

Leite, J. C. S. d. P. et al. (2005). ‘Quality-based Software Reuse’. In: *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*. CAiSE’05. Porto, Portugal: Springer-Verlag, pp. 535–550.

Li, F.-L. et al. (2014). ‘Non-functional requirements as qualities, with a spice of ontology’. In: *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pp. 293–302.

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.

Rosa, N., P. Freire Cunha, and G. Ribeiro Justo (2004). ‘An approach for reasoning and refining non-functional requirements’. English. In: *Journal of the Brazilian Computer Society* 10.1, pp. 62–84.

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 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 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.