Forging High-Quality User Stories: Towards a Discipline for Agile Requirements

Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E.M. van der Werf and Sjaak Brinkkemper
Department of Information and Computing Sciences
Utrecht University
Email: {g.lucassen, f.dalpiaz, j.m.e.m.vanderwerf, s.brinkkemper}@uu.nl

***Abstract*—User stories are a widely used notation for formulating requirements in agile development. Despite their popularity in industry, little to no academic work is available on determining their quality. The few existing approaches are too generic or employ highly qualitative metrics. We propose the Quality User Story Framework, consisting of 14 quality criteria that user stories should strive to conform to. Additionally, we introduce the conceptual model of a user story, which we rely on to subsequently design the AQUSA tool. This conceptual piece of software aids requirements engineers in turning raw user stories into higher quality ones by exposing defects and deviations from good practice in user stories. We evaluate our work by applying the framework and a prototype implementation to multiple case studies.**

I. Introduction

As practitioners transition to agile development, requirements are increasingly expressed [1] as user stories. Invented by Connextra in the United Kingdom and popularized by Mike Cohn [2], user stories only capture the essential elements of a requirement: *who* it is for, *what* it expects from the system, and, optionally, *why* it is important (a key aspect in RE [3]). The most well-known format, popularized by Mike Cohn in 2004 [2] is: "As a ⟨type of user⟩ , I want ⟨goal⟩, [so that ⟨some reason⟩]". For example: "As an Administrator, I want to receive an email when a contact form is submitted, so that I can respond to it".

In a 2014 survey among requirements analysts in an agile environment, user stories were the most used requirements documentation method [1]. Despite this popularity the number of methods to assess and improve user story *quality* is limited. Existing approaches to user story quality employ highly qualitative metrics, such as the heuristics of the INVEST (Independent-Negotiable-Valuable-Estimable-Scalable-Testable) framework [4], and the generic guidelines for ensuring quality in agile RE proposed by Heck and Zaidman [5].

The goal of this paper is to introduce a comprehensive approach to assessing and enhancing user story quality. To achieve this goal, we take advantage of the potential offered by natural language processing (NLP) techniques, while taking into account the reservations of Daniel Berry and colleagues [6]. Existing state-of-the-art NLP tools for RE such as QuARS [7], Dowser [8], Poirot [9] and RAI [10] are unable to transcend from academia into practice. The ambitious objectives of these tools necessitate a deep understanding of the requirements' contents [6]. A currently unachievable necessity which will remain impossible to achieve in the foreseeable future [11].

Instead, tools that want to harness NLP are effective only when they focus on the *clerical* part of RE that a tool can perform with 100% recall and high precision, leaving thinking-required work to human requirements engineers [6]. Additionally, they should conform to what practitioners actually do, instead of what the published methods and processes advise them to do [12]. User stories' popularity among practitioners and simple yet strict structure make them ideal candidates.

Throughout the remainder of this paper we make five concrete contributions that pave the way for the creation of these types of tools:

- Section II formulates a revised notion of user story quality. The Quality User Story Framework separates the algorithmic aspects that NLP can automatically process from the thinking-required concerns which necessitate involving human requirements engineers. We illustrate each quality criteria with a real-world example to demonstrate that the quality defect exists in practice;
- Section III proposes the conceptual model of a user story, detailing the decomposition of a single user story. We use this conceptual model throughout the paper;
- Section IV explores the different semantic relationships within sets of user stories, enabling identification of possible semantic quality improvements;
- Section V presents the design of a prototype tool (an example of a *dumb tool* [6]) that restricts itself to correcting the algorithmically determinable errors of user stories;
- Section VI empirically evaluates the feasibility of our approach by applying the framework and the prototype tool to a case study.

To conclude we discuss our work and explore future research opportunities in Section VIII.

II. What is User Story Quality?

Many different perspectives on requirements quality exist. The IEEE Recommended Practice for Software Requirements Specifications defines eight quality characteristics for a requirement [13]: correct, unambiguous, complete, consistent, ranked for importance/stability, verifiable, modifiable and traceable. However, most requirements specifications are unable to adhere to these in practice [14]. On top of this, these

TABLE I
QUALITY USER STORY FRAMEWORK

Criteria	Description
Syntactic	
- Atomic	A user story expresses a requirement for exactly one feature
- Minimal	A user story contains nothing more than role, means and ends
- Well-formed	A user story includes at least a role and a means
Semantic	
- Conflict-free	A user story should not be inconsistent with any other user story
- Conceptually sound	The means expresses a feature and the ends expresses a rationale, not something else
- Problem-oriented	A user story only specifies the problem, not the solution to it
- Unambiguous	A user story avoids terms or abstractions that may lead to multiple interpretations
Pragmatic	
- Complete	Implementing a set of user stories creates a feature-complete application, no steps are missing
- Explicit dependencies	Link all unavoidable, non-obvious dependencies on user stories
- Full sentence	A user story is a well-formed full sentence
- Independent	The user story is self-contained, avoiding inherent dependencies on other user stories
- Scalable	User stories do not denote too coarse-grained requirements that are difficult to plan and prioritize
- Uniform	All user stories follow roughly the same template
- Unique	Every user story is unique, duplicates are avoided

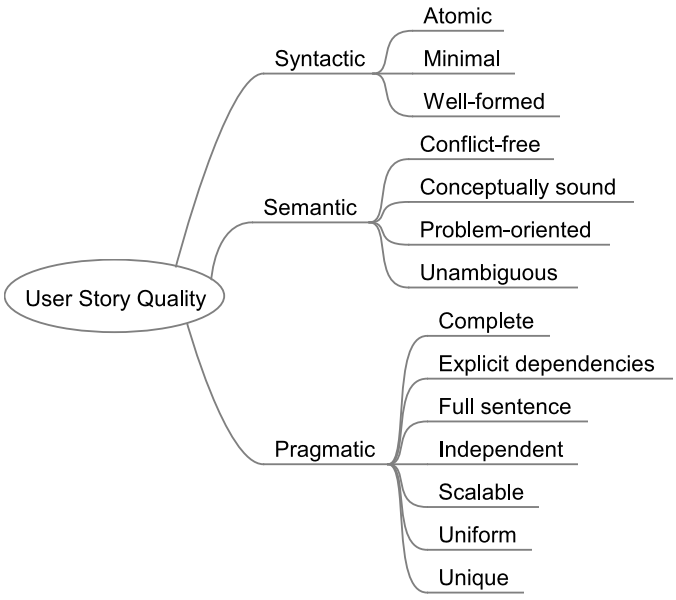

Fig. 1. Quality User Story Framework

quality characteristics were not developed with user stories nor agile development in mind.

The Agile Requirements Verification Framework [5] defines three high-level verification criteria for requirements in an Agile environment: completeness, uniformity, and consistency & correctness. The framework proposes specific criteria to be able to apply the quality framework to both feature requests and user stories. Many of these criteria, however, require supplementary, unstructured information that is not captured in the primary user story text, making them inadequate for dumb RE tools and our perspective.

With this in mind, we take inspiration from the verification framework [5] to define a new Quality User Story (QUS) Framework (Figure 1 and Table I). The QUS Framework only focuses on the information that is derivable from user story texts themselves, disregarding all requirements management concerns such as effort estimation and additional information sources such as descriptions or comments. The QUS Framework comprises 14 criteria that influence the quality of a user story or set of user stories. Because user stories are entirely textual, we classify each quality criteria according to three concepts borrowed from linguistics, similar to Lindland [15]:

Syntactic quality, concerning the textual structure of a user story without considering its meaning;

Semantic quality, concerning the relations and meaning of (parts of) the user story text;

Pragmatic quality, regarding choosing the most effective alternatives for communicating a given set of requirements.

In the next subsections, we introduce each criterion by presenting: (1) a comprehensive explanation of the criterion, (2) an example user story that violates the specific criterion, and (3) why the example violates the specific criterion. The example user stories originate from two real-world user story databases of software companies in the Netherlands. One contains 98 stories that specify the development of a tailor-made web information system. The other consists of 26 user stories from an advanced health care software product for home care professionals. We refrain from disclosing additional application details due to confidentiality constraints.

A. Syntax

a) Atomic: A user story should concern only one feature. It is tempting to combine multiple features in one user story when they are related or similar. Not doing this, however, makes estimation of the expected effort more accurate. In theory the combined effort estimation of two small, clear-cut user stories is more accurate than the estimation of one larger, more opaque user story. The user story in US_1 consists of two separate requests, the act of clicking on a location and the display of associated landmarks. The requirements engineer should split this user story into two autonomous user stories.

TABLE II
USER STORIES THAT BREACH QUALITY CRITERIA FROM TWO REAL-WORLD CASES

ID	Description	Issues
US_1	As a User, I'm able to click a particular location from the map and thereby perform a search of landmarks associated with that latitude longitude combination	Not atomic: two stories in one
US_2	As a care professional I want to see the registered hours of this week (split into products and activities). See: Mockup from Alice NOTE: - First create the overview screen - Then add validations	Not minimal, due to additional note about the mockup
US_3	Add static pages controller to application and define static pages	Missing role
US_4	As a User, I want to open the interactive map, so that I can see the location of landmarks	Conceptual issue: the end is in fact a reference to another story
US_5 US_6	As a User, I'm able to edit any landmark As a User, I'm able to delete a landmark which I added	Conflict: US_5 refers to any landmark, while US_6 only to those that user has added
US_7	As a care professional I want to save a reimbursement. - Add save button on top right (never greyed out)	Hints at the solution
US_8	As a User, I am able to edit the content that I added to a person's profile page	Unclear: what is content here?
US_9	As an Administrator, I am able to view content that needs to be reviewed	The type of content is not specified
US_{10}	Server configuration	In addition to being syntactically incorrect, this is not even a full sentence
US_{11} US_{12}	As an Administrator, I am able to add a new person to the database followed by As a Visitor, I am able to view a person's profile	Viewing relies on first adding a person to the database
US_{13}	As a care professional I want tosee my route list for next/future days, so thatI can prepare myself (for example I can see at what time I should start traveling)	Difficult to estimate because it is unclear what see my route list implies
US_{14}	As an Administrator, I receive an email notification when a new user is registered	Deviates from the template, no "wish" in the means
EP_A US_{15}	As a Visitor, I'm able to see a list of news items, so that I can stay up to date on news As a Visitor, I'm able to see a list of news items, so that I can stay up to date on news	The same equirement is repeated both in epic EP_A, and in a user story US_{14}

b) Minimal: User stories should contain a role, means and, optionally, ends. Nothing more. Any additional information such as comments, descriptions of the expected behavior or testing hints are to be captured as additional notes. Take, for example, US_2; aside from a role and means, this user story includes a reference to an undefined mock-up and a note on how to approach the implementation. The requirements engineer should move both to a comments or description section.

c) Well-formed: Before it can be considered a user story, the core requirements text needs to define a role and what the expected functionality entails: the *means*. For example, US_3 is one of the eight user stories in our sample user story database that do not adhere to this basic syntax, forgoing the role. This is most likely done because the requirement is quite technical, defining static pages and database issues. Nevertheless, the requirements engineer should fix this issue by introducing the relevant role.

B. Semantic

a) Conceptually sound: The means and ends parts of a user story play a specific role. The means should capture a concrete feature, while the end expresses the rationale for that feature. Consider US_4: the end is actually a dependency on another (hidden) functionality. The end (implicitly) references a functionality which is required before the means can be realized, implying the existence of a landmark database which isn't mentioned in other stories. A significant additional feature that is mistakingly represented as an end, but should be a means in a separate user story.

b) Conflict-free: To prevent implementation errors and rework, a user story should not conflict with any of the other user stories in the database. Requirements conflict occur when two or more requirements cause an inconsistency [16]. Although a comprehensive taxonomy of all types of conflicts is beyond the scope of this work, two major families of conflicts concern *activities* or *resources* [17]. Take, for example, US_5 and US_6. The second user story contradicts the earlier requirement that a user can edit any landmark. The new information that users are only allowed to delete content that they added themselves, raises the question whether this constraint also counts for the first requirement. The requirements engineer should modify the first user story and explicitly include whether a user can modify all landmarks.

c) Problem-oriented: A user story should only specify the problem, not the solution. If absolutely necessary, include implementation hints as a comment or description of the user story. Aside from breaking the minimal quality criteria, US_7 includes an implementation specification within the core user story text. The requirements engineer should remove this section and, if essential, post it as a comment.

d) Unambiguous: Ambiguity is inherent to natural language requirements, but the requirements engineer writing user stories should avoid it as much as possible. Not only should a user story refrain from being ambiguous itself, it should strive to be clear in relationship to all other user stories as well. The Taxonomy of Ambiguity Types by Berry and Kamsties is a comprehensive overview of the kinds of ambiguity that can be encountered in a systematic requirements specification [18]. In the context of user stories, we explore some possibilities in Section IV. Examples include registering alternative means with the same purpose or linking superclass terms to its respective elements. For example, in US_8, "content" is a superclass referring to audio, video and textual media

uploaded to the profile page. The requirements engineer should explicitly mention which media are editable.

C. Pragmatic

a) Complete: Implementing a set of user stories should create a feature-complete application. In some cases, however, the requirements engineer might forget or neglect writing a crucial user story causing a feature-gap that breaks the application's functionality. Unfortunately, it is impossible to include a missing user story in a table, but as an example consider that to be able to delete an item you first need to create it.

b) Explicit dependencies: Whenever a user story has a non-obvious dependency, it should explicitly link to the user story tag of the user story it depends on. For example, US_9 does not explicate what types of text or media 'content' references. To fix this, the requirement engineer should add the user story tags of the user stories that capture creating the relevant types of content.

c) Full sentence: A user story should read like a full sentence, without typos or hindering grammatical errors. US_{10}, for example, is not expressed as a full sentence (in addition to not complying with syntactic quality). By reformulating the feature as a full sentence user story, it will automatically specify what exactly needs to be configured.

d) Independent: User stories should not overlap in concept and should be schedulable and implementable in any order [4]. Note that this quality criterion is more of a ground rule that you try to follow to the best of your possibilities. Much like in programming loosely coupled systems, it is impossible to never breach this quality criterion. For example, US_{12} is dependent on US_{11}, because it is impossible to view a person's profile without first laying the foundation for creating a person. As such, there is a dependency between these two stories, that the requirements engineer is unable to do anything about. Although it is contradictory that a quality criteria is unattainable, requirements independence is sufficiently important to warrant inclusion in the QUS Framework.

e) Scalable: As user stories grow in size, it becomes more difficult to accurately estimate the effort required for the entire set of user stories. Therefore, each user story should not become so large as to avoid their estimation and planning with reasonable certainty [4]. For example, US_{13} requests a route list so that care professionals can prepare themselves. While this might be just an unordered list of places to go to during a workday, it is likely that the feature includes ordering the routes algorithmically to minimize distance traveled and/or showing the route on a map. This many functionalities makes accurate estimation difficult. The requirements engineer should split the user story into multiple, more specific user stories so that effort estimation is more accurate.

f) Uniform: All user stories should follow the same, agreed upon template. Minimal deviations are allowed when this better suits the narrative structure of the user story. For instance, using *I am able* instead of *I want to* is an acceptable deviation. However, US_{14} is a bigger deviation, along with 9 other stories in the first user story set (10.9% of the total). For uniformity, the requirements engineer should redefine these user stories using a wish.

g) Unique: A user story should not be a duplicate of another user story nor epic. For example, both epic EP_A and user story US_{15} (expressed as part of epic EP_A) share the same text. In this case, the epic contains seven other user stories that also concern other artifacts than news items. To resolve this issue, the requirements engineer should formulate an epic that captures the importance of both news items and events.

III. A Conceptual Model of User Stories

There are multiple syntax variations for user stories. Although originally an unstructured written description similar to use cases [19] but restricted in size [2], nowadays user stories follow a strict, compact template that captures *who* it is for, *what* it expects from the system, and (optionally) *why* it is important in a simple manner. The most well-known format, popularized by Mike Cohn in 2004 [2], is as follows: "As a ⟨type of user⟩, I want ⟨goal⟩, [so that ⟨some reason⟩]". The so-that clause between square brackets is optional.

When used in SCRUM, two other artifacts are relevant: epics and themes. An epic is a label for a large user story, which is broken down into smaller, implementable user stories. A theme is a collection of user stories grouped according to a given criterion [2]. For simplicity, and due to their greater popularity, we only include epics in our conceptual model.

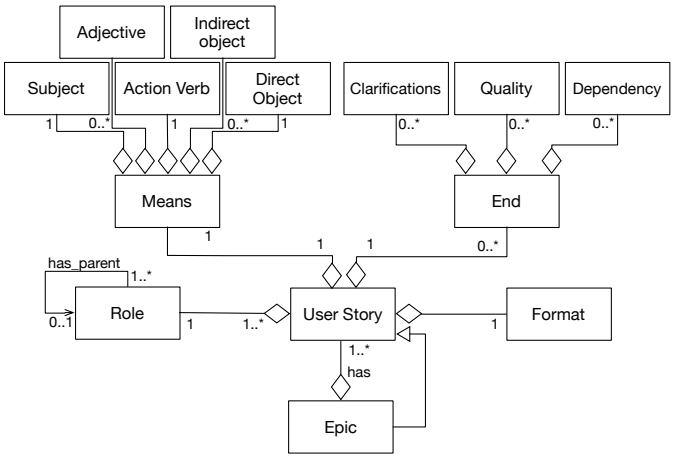

Fig. 2. Conceptual model of user stories

Figure 2 shows a UML class diagram for user stories. A user story itself consists of four parts: one role, one means, optionally one or more ends and a format. In the following subsections we elaborate on how to decompose each of these. Note that we deviate from Cohn's terminology, using the more abstract means-end instead of goal-reason. We do so because alternative user story templates talk about desire instead of goal, or benefit instead of reason and we do not want to choose one over the other.

A. Format

A user story should follow some pre-defined template as agreed upon among the stakeholders, such as the one proposed in [2]. The pre-defined text or skeleton of the template is what we depict as *format* in the conceptual model. This format is the foundation, in between which the role, means and optional end(s) are interspersed to forge a user story.

B. Role

A user story always defines one relevant role. Roles hierarchically relate to each other through the "has_parent" relationship. This can be used, for example, to determine that "Editor" is a type of (has parent) "User".

C. Means

Means can have different structures, for they can be used to represent different types of requirements. However, grammatically speaking, all means have three things in common: (1) they contain a *subject* with an intent such as "want" or "am able", (2) followed by an *action verb*[1] that expresses the action related to the feature being requested, and (3) a *direct object* on which the subject executes the action. For example: *"I want to open the interactive map"*. Aside from this basic requirement, means are essentially free form text which allow for an infinite number of constructions. Two common additions are an adjective or an indirect object, which results in the following example: *"I want to open a larger (adj) view of the interactive map from the person's profile page (io)"*. We included these interesting cases in the conceptual model, but left out all other variations, which will be studied in future research. However, they can be included in a domain-specific conceptual model by integrating a metamodel of, for instance, Web Requirements Engineering [20] or Embedded Real-Time Software [21].

D. End

The end part provides the reason for the means [2]. However, user stories frequently end up including other information. By analyzing the ends available in our learning set of user stories, we define three possible variants of a well-formed end:

1. **Clarification of means.** The end explains the reason of the means. Example: "As a User, I want to edit a record, so that I can correct any mistakes".
2. **Dependency on another (hidden) functionality.** The end (implicitly) references a functionality which is required for the means to be realized. Although dependency is an indicator of a bad quality criteria, having no dependency at all between requirements is impossible [4]. There is no size limit to this dependency on the (hidden) functionality. Small example: "As a Visitor, I want to view the homepage, so that I can learn about the project". The end implies the homepage also has relevant content, which requires extra input. Larger example: "As a User, I want to open the interactive map, so that I can see the location of landmarks". The end implies the existence of a landmark database, a significant additional functionality of the same requirement.
3. **Qualitative requirement.** The end communicates the intended qualitative effect of the means. For example: "As a User, I want to sort the results, so that I can more easily review the results" indicates that the means contributes maximizing easiness.

Note that these three are not mutually exclusive, but can occur simultaneously such as in "As a User, I want to open the landmark, so that I can more easily view the landmark's location". The means only specifies that the user wishes to view a landmark's page. The end, however, contains elements of all three types: (1) a clarification that you want to open the landmark to view its location, (2) additional functionality of the landmark and (3) the qualitative requirement that it should be easier than an alternative.

[1]While other types of verbs are in principle admitted, in this paper we focus on action verbs, which are the most used in user stories requesting features

IV. Identifying Cross-Story Relationships

There are two dimensions of user story quality: individual and cross-story. While several criteria in Table I concern individual user stories, some require taking into account all other user stories for the project at hand. For example, whether a user story is complete, independent, uniform and unique depends on the entire set of user stories.

In this section, we explore the relationships *between* user stories. We characterize each relationship, formalize them via first-order logic predicates that enable the identification of these relationship, and associate the relevant quality criteria. Our aim is to provide an initial set of relevant relationships, which is far from being complete.

Notation. Lowercase identifiers refer to single elements (e.g., one user story), and uppercase identifiers denote sets (e.g., a set of user stories). We employ the following notation:

- U for the set of user stories;
- $r_1, r_2, \ldots$ for role identifiers;
- $m_1, m_2, \ldots$ for means identifiers, where $m = \langle s, av, do, io, adj \rangle$ with s being a subject, av an action verb, do a direct object, io an indirect object, and adj an adjective (do and io may be null, see Fig. 2);
- $e_1, e_2, \ldots$ for end identifiers;
- $E_1, E_2, \ldots$ for identifiers of sets of ends;
- $f_1, f_2, \ldots$ for user story format;
- $\mu_1, \mu_2, \ldots$ for user stories, where $\mu = \langle r, m, E, f \rangle$, or, expanding m, $\mu = \langle r, \langle s, av, do, io, adj \rangle, E, f \rangle$

Furthermore, we assume that the equality, intersection, etc. operators are semantic, i.e., they look at the meaning of an entity (e.g., they account for synonyms, etc.), in addition to the syntax. To denote that an operator is merely syntactic, we add the "syn" subscript; so, for instance, $=_{syn}$. In this section, we employ theoretical semantic operators; Sec. V will discuss the feasibility of some of them. Finally, the function $depends(av, av')$ denotes that executing the action av on a specific object requires first executing av' on that very object (e.g., "delete" depends on "create").

A. Complete

Some user stories imply the necessity of other functionality not yet captured in another user story. A simple example is user stories with action verbs that refer to a non-existent direct object: to read, update or delete an item you first need to create it. In practice, however, completeness is very context dependent. Because of this, we define this relationship on a high abstraction level, but focusing on dependencies concerning the means' direct object. Formally, the predicate $missesDep(\mu)$ holds when a dependency for μ's direct object is missing:

$$missesDep(\mu) \leftrightarrow depends(av, av') \ \wedge \ \nexists \mu' \in U.\ do' = do$$

B. Independent

Many different types of dependency exist, and our intent is not to list them here. For illustration, we explain two cases.

a) Causality: In some cases, it is necessary that one user story μ_1 is completed before the developer can start on another user story μ_2 (US_{11} and US_{12} in Table II). Such a causal dependency foremost impacts the *independent* quality criterion. When this dependency is non-obvious and implicit, the requirements engineer needs to consider adding an *explicit dependency*. Formally, the predicate $hasDep(\mu_1, \mu_2)$ holds when μ_1 causally depends on μ_2:

$$hasDep(\mu_1, \mu_2) \leftrightarrow depends(av_1, av_2) \ \wedge \ do_1 = do_2$$

b) Superclasses: An object of one user story μ_1 can refer to multiple other objects of a set of user stories $\{\mu_2, \mu_3, \ldots\}$, indicating that the object of μ_1 is a parent or *superclass* of the other objects. 'Content' for example can refer to different types of multimedia, as exemplified in US_8. This relationship has an impact on the *unambiguous*, *independent* and *explicit dependencies* quality criteria. One can detect ambiguity or missing dependencies by scanning for Create, Read, Update and Delete (CRUD) actions which have a unique direct object. Formally, predicate $hasIsaDep(\mu, do')$ is true when μ has a direct object superclass dependency based on the sub-class do' of do.

$$hasIsaDep(\mu, do') \leftrightarrow \exists \mu' \in \ U.\ is\text{-}a(do', do)$$

C. Uniformity

Uniformity in the context of user stories means that a user story has a format that is consistent with the format of all other user stories. To test this, the requirements engineer needs to review the syntax of all other user stories to determine the most frequently occurring format, typically the format agreed upon with the team. The format of an individual user story $\mu = \langle r, m, E, f \rangle$ is syntactically compared to the most common format f_{std} to determine whether it adheres with the *uniformity* quality criterion. US_{14} in Table II was en example of a non-uniform user story. Formally, predicate $isNotUniform(\mu, f_{std})$ is true if the format of μ deviates from the standard:

$$isNotUniform(\mu, f_{std}) \leftrightarrow f \neq_{syn} f_{std}$$

D. Unique

A user story is unique when no other user story is (semantically) the same or too similar. There are many different ways in which two user stories can be similar. For example, all user stories should follow a similar user story format. The type of similarity we are interested in, however, is a potential indicator of duplicate or conflicting user stories such as user stories US_5 and US_6 or US_{15} and epic EP_A in Table II. We discuss five similarity relationships that help detecting similarity among user stories.

To detect these types of relationships, each user story part needs to be compared with the parts of other user stories, using a combination of similarity measures that are either syntactic (e.g., Levenshtein's distance) and semantic (e.g., employing an ontology to determine synonyms). When a similarity exceed a certain threshold, a human analyst is required to examine the user stories for potential conflict and/or duplication.

a) Full duplicate: A user story μ_1 is an exact duplicate of another user story μ_2 when the stories are identical. This impacts the *unique* quality criterion. Formally,

$$isFullDuplicate(\mu_1, \mu_2) \leftrightarrow \mu_1 =_{syn} \mu_2$$

b) Semantic duplicate: A user story μ_1 that duplicates the request of μ_2, while using a different text; this has an impact on the *unique* quality criterion. Formally,

$$isSemDuplicate(\mu_1, \mu_2) \leftrightarrow \mu_1 = \mu_2 \wedge \mu_1 \neq_{syn} \mu_2$$

c) Different means, same end: Two or more user stories that have the same end, but achieve this using different means. This relationship potentially impacts two quality criteria, as it may indicate: (i) a feature variation that should be explicitly noted in the user story to maintain an *unambiguous* set of user stories, or (ii) a conflict in how to achieve this end, meaning one of the user stories should be dropped to ensure *conflict-free* user stories. Formally, for user stories μ_1 and μ_2:

$$diffMeansSameEnd(\mu_1, \mu_2) \leftrightarrow m_1 = m_2 \wedge E_1 \cap E_2 \neq \emptyset$$

d) Same means, different end: Two or more user stories that use the same means to reach different ends. This relationship affects the qualities of user stories to be *unique* or *independent* of each other. If the ends are not conflicting, they could be combined into a single larger user story; otherwise, they are multiple viewpoints that should be resolved. Formally,

$$sameMeansDiffEnd(\mu_1, \mu_2) \leftrightarrow m_1 \neq m_2 \ \wedge \\ (E_1 \setminus E_2 \neq \emptyset \vee E_2 \setminus E_1 \neq \emptyset)$$

e) Different role, same means and/or same end: Two or more user stories with different roles, but same means and/or ends indicates a strong relationship. Although this relationship has an impact on the *unique* and *independent* quality criteria, it is considered good practice to have separate user stories for the same functionality for different roles. As such, requirements engineers could choose to ignore this impact. Formally,

$$diffRoleSameStory(\mu_1, \mu_2) \leftrightarrow r_1 \neq r_2 \ \wedge \\ (m_1 = m_2 \ \vee \ E_1 \cap E_2 \neq \emptyset)$$

f) Purpose = means: The end of one user story μ_1 is identical to the means of another user story μ_2. Indeed, the same piece of text can be used to express both a wish, and a reason for another wish. When there is this strong a semantic relationship between two user stories, it is important to add *explicit dependencies* to the user stories, which is a trade-off with the *independent* quality criterion. Formally, $purposeMeans(\mu_1, \mu_2, x)$ is true if x is an end in μ_1 and a means in μ_2

$$purposeMeans(\mu_1, \mu_2, x) \leftrightarrow \exists e \in E_1 \ s.t. \ e = m_2$$

V. A Dumb Tool for Improving User Story Quality

The Quality User Story (QUS) Framework provides structured guidelines for improving the quality of (a set of) user stories. To support the framework, we propose the Automatic Quality User Story Artisan (AQUSA) tool, which exposes defects and deviations from good practice in user stories.

Unlike most NLP tools for RE, and in line with Berry's notion of a *dumb tool* [6], we require our tool to detect defects with close to 100% recall. This is necessary to avoid that a human requirements engineer has to double check the entire requirements document for missed defects [22]. On the other hand, *precision*, the number of false positives in proportion to the detected defects, should not be so high that the user perceives AQUSA to report useless errors. Thus, AQUSA is designed as a tool that focuses on easily describable, algorithmically determinable defects. This allows the requirements engineer to focus on thinking-required defects for which 100% recall with high precision is impossible [22]. Consequently, AQUSA can support only certain QUS criteria in an effective manner:

- Syntactical criteria are detectable with 100% recall; AQUSA can report need-to-improve defects with high precision.
- Semantic criteria are impossible to detect with 100% recall and are thus out of scope for AQUSA.
- Some aspects of the pragmatic criteria are detectable with 100% recall, while others are not. For AQUSA, we select a number of algorithmically determinable subparts.

Next, we present the selected quality criteria, discuss their theoretical implementation and provide accompanying example input and output user stories. This is followed by how we envision AQUSA's architecture and implementation.

A. Syntax

a) Well-formed: One of the essential aspects of verifying whether a string of text is a requirement, is splitting it into role, means and end(s). This process consists of two steps: (1) *chunking* on commas and common indicator texts such as *As a*, *I want to*, *I am able to* and *so that*; (2) verifying that each chunk contains their relevant part. A grammatical tagger assigns a word category to each of the words in the chunk. For each chunk, AQUSA tests the following rules:

Role: Is the last word a noun? Do the words before the noun match a known role format?

Means: Is the first word I? Can we identify a known means format? Does the part include two verbs and a noun?

End: Is the end present? Does it start with a known end format?

When AQUSA encounters the user story *"Add static pages controller to application and define static pages"*, it includes the user story in its report, highlighting that it is not well-formed because it does not explicitly contain a role nor means. The well-formed example input user story *"As a Visitor, I want to register at the site, so that I can contribute"*, however, would be verified and separated into the following chunks for later use:

Role: As a Visitor
Means: I want to register at the site
End: so that I can contribute

b) Atomic: To audit that the means of the user story concerns only one feature, AQUSA parses the means for occurrences of *"and, &, +"* to include any double feature requests in its report. Additionally, AQUSA suggests the reader to split the user story into multiple user stories. US_1 from Table II, for example would generate a suggestion to be split into two user stories: (1) *"As a User, I want to click a particular location from the map"* and (2) *"As a User, I want to perform a search of landmarks associated with the latitude longitude combination of a location."*.

c) Minimal: To test this quality criterion, AQUSA relies on the results of chunking and verification of the *role and means* quality criterion. When this process has been successfully completed, AQUSA reports any user story that contains additional text after a dot, hyphen, semicolon or other separating punctuation marks. For example the *"See: Mockup from Alice"* text from US_2 is sufficient for AQUSA to report that the user story is not minimal.

B. Pragmatic

a) Explicit Dependencies: Whenever a user story includes an explicit dependency on another user story, it should include a navigable link to the dependency. Because the popular issue trackers Jira and Pivotal Tracker use numbers for dependencies, AQUSA checks for numbers in user stories and checks whether the number is contained within a link. The example *"As a care professional, I want to edit the planned task I selected, so that I can create a realization for this line - see 98059."* would prompt the user to change the isolated number to *"See PAW-98059"*. On the end of the issue tracker, this should automatically change to *"see PAW-98059 (http://companyname.issuetracker.org/browse/PAW-98059)"*

b) Uniform: Aside from chunking, AQUSA extracts the user story format parts out of each chunk and counts their occurrences throughout the set of user stories. The most commonly occurring format is used as the standard user story format. All other user stories are marked as non-compliant to the standard and included in the error report. For example, AQUSA reports that "As a User, *I am able to* delete a landmark" deviates from the standard *'I want to'*.

c) Unique: AQUSA implements each of the similarity measures as outlined in Section IV using the WordNet lexical database [23] to detect semantic similarity. For each verb and object in a means or end, AQUSA runs a WordNet::Similarity calculation with the verbs or objects of all other means or ends. Combining the calculations results in one similarity degree for two user stories. When this metric is bigger than 90% AQUSA reports the user stories as potential duplicates.

C. Architecture and Implementation

AQUSA is designed as a simple, stand-alone, deployable as-a-service application that analyzes a set of user stories regardless of its source of origin. Being a service, AQUSA can be invoked by some popular requirements management tools that requirements engineers use to create user stories, but also MS Excel spreadsheets describing user stories can be imported. By retaining its independence from other tools, AQUSA is capable of easily adapting to future technology changes. Aside from importing user stories, AQUSA consists of four main architectural components (Figure 3):

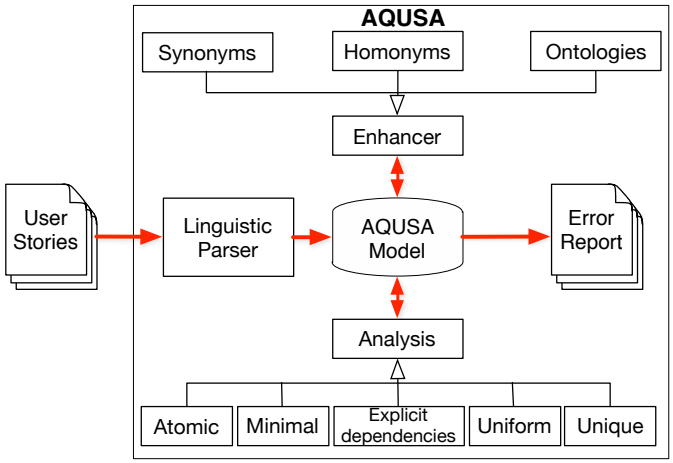

Fig. 3. Architecture of AQUSA

Linguistic parser: the first step for every user story is validating that it is well-formed. This takes place in the linguistic parser, which chunks the text according to common indicator texts and assigns word categories based on the NLTK grammatical tagger.

AQUSA model: a parsed user story is captured as an object—aligned with the conceptual model in Fig. 2—in the AQUSA model database, ready to be processed further.

Enhancer: AQUSA enhances user stories by adding possible synonyms, homonyms and possibly semantic information extracted from an ontology to the relevant words in each chunk.

Analysis: AQUSA analyzes user stories by running methods that verify the selected syntactic and pragmatic quality criteria: atomic, minimal, explicit dependencies, uniform and unique.

We developed an AQUSA proof of concept with a limited scope that is still under active development[2]. Although currently only the syntactical quality criteria are implemented that has only been tested with a single user story set, these preliminary results are promising. Out of 96 user stories, this proof of concepts finds 7 that are not well-formed and 8 that are not atomic. A 100% recall in comparison to the manual analysis in Section VI.

VI. Empirical Evaluation

We evaluate the QUS Framework in two ways. In Subsection VI-A, we manually apply all the quality criteria to a third set of mature user stories to validate their occurrence besides the two user story sets we used while constructing our approach. Additionally, in Subection VI-B, we apply the algorithmically determinable quality criteria of AQUSA to all three user story sets to show which errors exist in real-world user story requirements, and to evaluate the effectiveness of our identification mechanisms.

A. Applying the Quality User Stories Framework

We validate the QUS Framework by applying its criteria to a third set of *intuitively high-quality* user stories, with the aim to assess if its perceived quality is confirmed by our quality framework. These user stories belong to a point of sales software product with customers world-wide. A requirements engineering team in Belgium creates the user stories and outsources their development to a near-shore team in Romania. Over the span of a year, both sides made significant investments to be able to collaborate in an effective manner. User stories are central to this collaboration. In the experience of the Belgian team, user story quality directly influences the quality of the software product. Despite their high quality, however, these user stories still contain flaws. In the following paragraphs, we discuss how and why some criteria are breached, emphasizing three interesting patterns that arose.

a) The Pragmatic Requirements Engineer: The data set contains a number of quality errors that are easily avoidable. They request more than one feature, do not comply to the user story format or specify a solution instead of a problem. Aside from a few exceptions, these errors appear to exist because of the attempt of representing technical requirements using the user story format.

b) Keep it Minimal: It is particularly difficult to adhere to one quality criteria: minimal. More than 10% of the user stories include additional information, distracting the reader from the user story itself.

c) Context is King: The examined user stories concern a mature software product for a specific business domain with its own particular jargon. As a consequence, it is difficult for a non-informed outsider to accurately test some quality criteria. Not knowing the technical implications of user stories makes it difficult to estimate their scalability, an effect that jargon amplifies by causing lexical ambiguity. Moreover, without

[2]Code available at http://github.com/gglucass/aqusa

intimate knowledge of the application, it is nearly impossible to detect dependencies or conflicts among user stories of which one cannot be sure that they are conceptually sound. This clearly calls for reliance on domain-specific ontologies.

The overall quality is high: no dependencies or conflicts were immediately obvious. Nevertheless, using structured approaches one can detect some defects, e.g. by collecting all user stories that contain a frequently occurring phrase we did find some dependencies. For instance: by grouping nine user stories concerning 'price', a causal dependency quickly became apparent that was hidden among the other user stories before.

B. Applying the Automatic QUS Artisan

To test whether a full AQUSA implementation is effective to determine user story quality, we manually apply the rules introduced in Section V to our three user story sets. The resulting quality criteria breaches in Table III show promising results that indicate high potential for successful further development. For each set, at least 25% of the processed user stories violate one or more quality criteria that the AQUSA algorithms can detect. Moreover, AQUSA detect the total amount of user stories with with errors with 71% precision. Furthermore, the quality criteria breaches significantly vary per user story set. User story set #1 is very minimal, but breaks uniformity in 15% of all user stories. For user story sets #2 and #3 the inverse is true; although they are very uniform, they are not minimal.

Using these results, we compare the main issues between the user story sets. By looking at AQUSA's grouping of all non-uniform user stories of user story set #1, we immediately recognize its primary issue: 10 user stories omit 'want to', directly expressing the functionality. Moreover, 5 out of 7 reports of atomic errors are false positives, while the not well-formed user stories are the consequence of technical requirements that are difficult to capture in the user story format.

This is in stark contrast with user story set #2, which contains errors in all AQUSA's criteria but one (unique). It is immediately apparent that the majority of these user stories are not minimal. More than 2/3rds of these user stories contain notes, feedback, testing hints, todo's and/or solution specifications. Although user story set #3 has 17 potential atomic defects according to AQUSA, 12 are false positives. The real primary issue is minimality, albeit less frequent and less severe than for #2. In 7 cases the requirements engineer has added a reference to a specific document, the remaining 9 breaches contain no relevant patterns.

TABLE III
NUMBER OF IDENTIFIED VIOLATIONS (V) AND FALSE POSITIVES (FP) PER QUALITY CRITERIA IN THE THREE USER STORY SETS

	Set 1 (n=96)		**Set 2 (n=24)**		**Set 3 (n=124)**	
	V	*FP*	*V*	*FP*	*V*	*FP*
Atomic	7	5	10	3	17	12
Minimal	0	-	17	-	16	-
Well-formed	8	-	2	-	6	-
Explicit dependencies	0	-	1	-	0	-
Uniform	14	4	2	-	1	-
Unique	2	-	0	-	0	-
Total US with errors	27	9	19	3	37	12

VII. RELATED LITERATURE

Despite their popularity among practitioners [1], academic research on user stories is few and far between. The little work that is available, concerns a diverse list of topics. The connection of user stories to code was studied to retrieve reusable test steps [24]. A conceptual method for identifying dependencies between User Stories [25] was proposed by Gomez and colleagues, using an approach similar to ours, i.e., relying on the data entities that stories refer to. Along the same lines, a basic tool was developed for writing consistent user stories [26]. Plank, Sauer and Schaefer reported on the potential of applying NLP to user stories [27]. They proposed that by analyzing source code, comments, bug reports one can establish links between user stories and their implementation progress. Unfortunately, in private communication, the first author indicated that they chose not to pursue this research line any further. In future work, however, we intend to pursue a similar goal by building on the conceptual model presented in this paper.

Applying natural language processing to RE has historically been heralded as the final frontier of requirements engineering. Nowadays, this ambitious objective is understood to be unattainable in the foreseeable future—at least not without a significant, fundamental breakthrough [11]. Nevertheless, a wide variety of contemporary research in RE applies NLP for specific uses: automatically identifying security requirements hidden in other requirements [28], detecting uncertainty in NL requirements [29] or improving NL requirements quality by semi-automatically detecting a range of bad practices [7]. Tools like these are interesting research artifacts, but still far from becoming mainstream in practice.

Arguing that these tools deliver the opposite effect of what they intend, Berry [6] calls for NLP supported tools that support 100% recall. Our AQUSA tool is an attempt to satisfy this constraint in the context of requirements quality. However, we will investigate the techniques other tools rely upon to determine if some of them have the potential for improving user story quality.

Multiple frameworks exist for characterizing requirements quality, a very vague concept in general. The IEEE Recommended Practice for Software Requirements Specifications is the standard body of work on this subject, defining eight quality characteristics [13]. Unfortunately, most requirements specifications are unable to adhere to them in practice [14], although evidence shows a correlation between high-quality requirements and project success [30].

VIII. CONCLUSION AND FUTURE RESEARCH

In this paper, we have argued for user stories as an ideal candidate for improving requirements quality using Natural

Language Processing (NLP) techniques. They conform to what practitioners in agile development actually do, and detection of errors is possible with 100% recall and high precision. This paper laid down the theoretical foundations for such a tool, and made three contributions.

1) A revised notion of user story quality in the Quality User Story (QUS) Framework which provides requirements engineers with the information necessary to forge higher quality user stories.
2) A conceptual model of a user story that is used by the tool in order to identify points for improvement.
3) An initial set of relationships, with preliminary techniques to identify them, that pinpoint where user stories lack of quality.

Based on these theoretical contributions, we design the Automatic Quality User Story Artisan (AQUSA), a prototype tool which exposes defects and deviations from good practice in user stories. The promising results of our application of both the QUS Framework and AQUSA to 1 and 3 user story sets demonstrates the feasibility and relevance of this work.

This paper paves the way for future work. By studying how requirements engineers apply and experience the QUS Framework in practice, we can validate it further. Moreover, implementing a robust version of AQUSA enables quantitative analysis of user story databases and demonstrating the increase of user story quality when practitioners use the tool for longer periods of time. One of the key challenges will be to reduce the number of false positives, while retaining the ability to achieve a 100% recall. Exploring the potential of both domain and foundational ontologies is another direction to pursue, that has the potential to significantly improve the tooling.

ACKNOWLEDGEMENTS

The authors would like to thank Floris Vlasveld, Erik Jagroep, Jozua Velle and Frieda Naaijer for providing real-world user story data. Additionally, we would like to thank Leo Pruijt for his comments on an earlier draft of this paper.

REFERENCES

[1] X. Wang, L. Zhao, Y. Wang, and J. Sun, "The Role of Requirements Engineering Practices in Agile Development: An Empirical Study," in *Requirements Engineering*. Springer, 2014, vol. 432, pp. 195–209.

[2] M. Cohn, *User Stories Applied: for Agile Software Development*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[3] E. S. K. Yu and J. Mylopoulos, "Understanding "Why" in Software Process Modelling, Analysis, and Design," in *Proc. of the International Conference on Software Engineering*. IEEE, 1994, pp. 159–168.

[4] B. Wake, "INVEST in Good Stories, and SMART Tasks," http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/, 2003, accessed: 2015-02-18.

[5] P. Heck and A. Zaidman, "A Quality Framework for Agile Requirements: A Practitioner's Perspective," *CoRR*, vol. abs/1406.4692, 2014. [Online]. Available: http://arxiv.org/abs/1406.4692

[6] D. Berry, R. Gacitua, P. Sawyer, and S. Tjong, "The Case for Dumb Requirements Engineering Tools," in *Proc. of Requirements Engineering: Foundation for Software Quality*. Springer, 2012, vol. 7195, pp. 211–217.

[7] A. Bucchiarone, S. Gnesi, and P. Pierini, "Quality Analysis of NL Requirements: An Industrial Case Study," in *Proc. of the IEEE International Conference on Requirements Engineering*, 2005, pp. 390–394.

[8] D. Popescu, S. Rugaber, N. Medvidovic, and D. M. Berry, "Reducing Ambiguities in Requirements Specifications Via Automatically Created Object-Oriented Models," in *Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs*. Springer, 2008, vol. 5320, pp. 103–124.

[9] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best Practices for Automated Traceability," *Computer*, vol. 40, no. 6, pp. 27–35, 2007.

[10] R. Gacitua, P. Sawyer, and V. Gervasi, "On the effectiveness of abstraction identification in requirements engineering," in *Proc. of the IEEE International Requirements Engineering Conference*, 2010, pp. 5–14.

[11] K. Ryan, "The Role of Natural Language in Requirements Engineering," in *Proc. of the IEEE International Symposium on Requirements Engineering*, 1993, pp. 240–242.

[12] N. Maiden, "Exactly How Are Requirements Written?" *IEEE Software*, vol. 29, no. 1, pp. 26–27, 2012.

[13] IEEE Computer Society, "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1993*, 1994.

[14] M. Glinz, "Improving the Quality of Requirements with Scenarios," in *Proc. of the World Congress on Software Qualtiy*, 2000, pp. 55–60.

[15] O. I. Lindland, G. Sindre, and A. Sølvberg, "Understanding Quality in Conceptual Modeling," *IEEE Software*, vol. 11, no. 2, pp. 42–49, 1994.

[16] W. N. Robinson, "Integrating Multiple Specifications Using Domain Goals," *SIGSOFT Software Engineering Notes*, vol. 14, no. 3, pp. 219–226, 1989.

[17] M. Kim, S. Park, V. Sugumaran, and H. Yang, "Managing Requirements Conflicts in Software Product Lines: A Goal and Scenario Based Approach," *Data & Knowledge Engineering*, vol. 61, no. 3, pp. 417–432, Jun. 2007.

[18] D. M. Berry and E. Kamsties, "Ambiguity in Requirements Specification," in *Perspectives on Software Requirements*. Springer, 2004, vol. 753, pp. 7–44.

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

[20] M. Escalona and N. Koch, "Metamodeling the Requirements of Web Systems," in *Web Information Systems and Technologies*. Springer, 2007, vol. 1, pp. 267–280.

[21] P.-A. Hsiung, S.-W. Lin, C.-H. Tseng, T.-Y. Lee, J.-M. Fu, and W.-B. See, "VERTAF: An Application Framework for the Design and Verification of Embedded Real-Time Software," *IEEE Transactions on Software Engineering*, vol. 30, no. 10, pp. 656–674, 2004.

[22] S. F. Tjong and D. M. Berry, "The Design of SREE: A Prototype Potential Ambiguity Finder for Requirements Specifications and Lessons Learned," in *Proc. of Requirements Engineering: Foundation for Software Quality*. Springer, 2013, vol. 7830, pp. 80–95.

[23] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[24] M. Landhäusser and A. Genaid, "Connecting User Stories and Code for Test Development," in *Proc. of the International Workshop on Recommendation Systems for Software Engineering*. Piscataway, NJ, USA: IEEE, 2012, pp. 33–37.

[25] A. Gomez, G. Rueda, and P. Alarcn, "A Systematic and Lightweight Method to Identify Dependencies between User Stories," in *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2010, pp. 190–195.

[26] M. Śmiałek, J. Bojarski, W. Nowakowski, and T. Straszak, "Writing Coherent User Stories with Tool Support," in *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2005, pp. 247–250.

[27] B. Plank, T. Sauer, and I. Schaefer, "Supporting Agile Software Development by Natural Language Processing," in *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*. Springer, 2013, pp. 91–102.

[28] M. Riaz, J. King, J. Slankas, and L. Williams, "Hidden in Plain Sight: Automatically Identifying Security Requirements from Natural Language Artifacts," in *Proc. of the IEEE International Requirements Engineering Conference*, 2014, pp. 183–192.

[29] H. Yang, A. De Roeck, V. Gervasi, A. Willis, and B. Nuseibeh, "Speculative Requirements: Automatic Detection of Uncertainty in Natural Language Requirements," in *Proc. of the IEEE International Requirements Engineering Conference*, 2012, pp. 11–20.

[30] M. I. Kamata and T. Tamai, "How Does Requirements Quality Relate to Project Success or Failure?" in *Proc. of the IEEE International Requirements Engineering Conference*. IEEE, 2007, pp. 69–78.