Requirements Eng (2017) 22:453–473
DOI 10.1007/s00766-016-0247-5

ORIGINAL ARTICLE

A framework for quality assessment of just-in-time requirements: the case of open source feature requests

Petra Heck[1] · **Andy Zaidman**[2]

Received: 14 April 2015 / Accepted: 29 February 2016 / Published online: 14 March 2016

Abstract Until now, quality assessment of requirements has focused on traditional up-front requirements. Contrasting these traditional requirements are just-in-time (JIT) requirements, which are by definition incomplete, not specific and might be ambiguous when initially specified, indicating a different notion of "correctness." We analyze how the assessment of JIT requirements quality should be performed based on the literature of traditional and JIT requirements. Based on that analysis, we have designed a quality framework for JIT requirements and instantiated it for feature requests in open source projects. We also indicate how the framework can be instantiated for other types of JIT requirements. We have performed an initial evaluation of our framework for feature requests with eight practitioners from the Dutch agile community, receiving overall positive feedback. Subsequently, we have used our framework to assess 550 feature requests originating from three Open Source Software systems (Netbeans, ArgoUML and Mylyn Tasks). In doing so, we obtain a view on the feature request quality for the three open source projects. The value of our framework is threefold: (1) it gives an overview of quality criteria that are applicable to feature requests (at creation time or JIT); (2) it serves as a structured basis for teams that need to assess the quality of their JIT requirements; and (3) it provides a way to get an insight into the quality of JIT requirements in existing projects.

✉ Petra Heck
p.heck@fontys.nl

[1] Fontys Applied University, Eindhoven, The Netherlands

[2] Delft University of Technology, Delft, The Netherlands

Keywords Just-in-time requirement · Quality framework · Quality assessment · Feature request · Open source

1 Introduction

It is increasingly uncommon for software systems to be fully specified before implementation begins. As stated by Ernst et al. [8], "The 'big design up-front' approach is no longer defensible, particularly in a business environment that emphasizes speed and resilience to change." They observe that an increasing number of industry projects treat requirements as tasks, managed with task management tools like Jira or Bugzilla. A similar task-based approach is seen in the agile movement and in open source projects [23, 33]. In an earlier paper, Ernst and Murphy [9] use the term "just-in-time requirements" (JIT requirements) for this. They observed that requirements are "initially sketched out with simple natural language statements," only to be fully elaborated (not necessarily in written form) when being developed. This indicates that the notion of quality for JIT requirements is different from the notion of quality for traditional up-front requirements.

Requirements verification "ensures that requirements specifications and models meet the necessary standard of quality to allow them to be used effectively to guide further work" [20]. Verification activities as in this definition ensure that the requirements are specified in a correct way. In this paper, we focus on the informal verification of JIT requirements, which we will call *quality assessment*. Standards such as IEEE-830 [19] define criteria for "informal correctness": requirements should be complete, unambiguous, specific, time-bounded, consistent, etc. However, this standard focuses on traditional up-front

requirements. These are requirements sets that are completely specified (and used as a contract) before the start of design and development. We have not found a practical implementation of quality assessment for JIT requirements.

There is some evidence that correctly specified requirements contribute to a higher software product quality [10, 21, 22]. The question is: Does the same hold for JIT requirements? After all, incorrect JIT requirements will be spotted early on because of short iterations and can be more easily corrected because of high customer involvement. Our hypothesis is that, at the least, early verification helps to save time and effort in implementing the requirements. After all, even if the work can be redone in a next iteration to correct wrong implementations, it still pays off to do it right the first time. We would like to investigate which quality criteria define "doing it right" for JIT requirements. This leads us to our **main research question**: *Which criteria should be used for the quality assessment of just-in-time requirements?*

We use a quality framework from previous work [13] to present the JIT requirements quality criteria in a structured way. Ernst and Murphy [9] describe two types of JIT requirements: *features* and *user stories*. User stories are the requirements format typically used in agile projects [24]. A feature or feature request is a structured request (issue report with title, description and a number of attributes) "documenting an adaptive maintenance task whose resolving patch(es) implement(s) new functionality" [16]. Most open source projects use this type of structured requests (also called "enhancements") for collecting requirements [14]. For an example, see Fig. 1. For the purpose of this paper, we focus on open source feature requests documented in online issue trackers because of their public availability and their structured (i.e., the fields of the issue tracker) nature. This leads us to the detailed research question: **[RQ1]** *Which quality criteria for informal verification apply to feature requests?*

As the first version of our quality framework for feature requests is based on the literature and our own experience, we deem it necessary to evaluate the resulting criteria with practitioners. This leads to the following research question: **[RQ2]** *How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?*

Once practitioners deem our framework valuable, we apply it to existing open source projects. In that way, we get both a) experiences in applying the framework in practice and b) insight into the quality criteria of feature requests in open source projects. This constitutes our last research question: **[RQ3]** *What is the level of quality for feature requests in existing open source projects as measured by our framework?*

The remainder of this paper is structured as follows. Section 2 explains the quality framework used. Section 3 instantiates the framework for feature requests. Section 4 indicates how to customize the framework for other situations and types of JIT requirements, with an example for user stories. Sections 5 and 6 describe our evaluation of the framework. Section 7 highlights the findings from the application of the framework to the existing projects. Section 8 discusses the research questions, including recommendations for practitioners working with feature requests, while Sect. 9 contains related work. Section 10 concludes this paper.

2 A quality framework

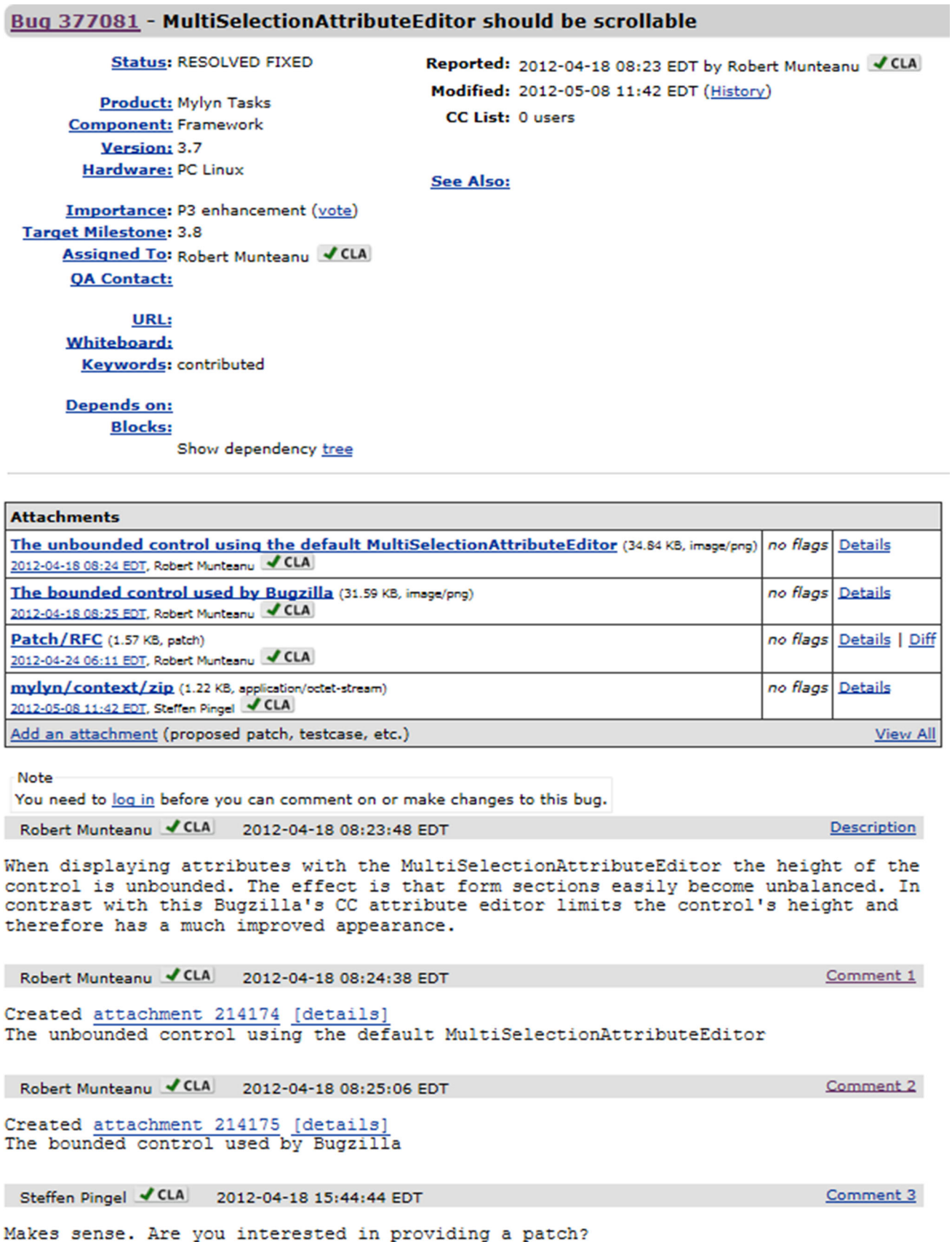
In previous work, we have performed an in-depth study on quality criteria for traditional up-front requirements, which we collected from an extensive list of standards (ISO/IEC/IEEE/ESA) and a literature review (see [13] for more details). The resulting quality criteria are included in the software product certification model (SPCM), a quality framework for software products with traditional up-front requirements.

The SPCM divides a software product (including all design, documentation and tests) into so-called elements. For traditional up-front requirements, the elements according to SPCM are: use cases or functional requirements, behavioral properties (e.g., business rules), objects (in, e.g., an entity-relationship diagram or a glossary) and non-functional requirements.

The SPCM furthermore structures the quality criteria for all parts of a software product in three groups, called certification criteria (CC):

[CC1] Completeness. All required elements are present. Group CC1 contains quality criteria for three different levels of detail (or formality) in those elements: required, semiformal or formal.

[CC2] Uniformity. The style of the elements is standardized. Group CC2 contains quality criteria for three different levels of standardization of those elements: within the project, following company standards and following industry standards.



Bug 377081 - MultiSelectionAttributeEditor should be scrollable

Status: RESOLVED FIXED
Reported: 2012-04-18 08:23 EDT by Robert Munteanu CLA
Modified: 2012-05-08 11:42 EDT (History)
Product: Mylyn Tasks
Component: Framework
Version: 3.7
Hardware: PC Linux
CC List: 0 users
See Also:
Importance: P3 enhancement (vote)
Target Milestone: 3.8
Assigned To: Robert Munteanu CLA
QA Contact:
URL:
Whiteboard:
Keywords: contributed
Depends on:
Blocks:
Show dependency tree

Attachments		
The unbounded control using the default MultiSelectionAttributeEditor (34.84 KB, image/png) 2012-04-18 08:24 EDT, Robert Munteanu CLA	no flags	Details
The bounded control used by Bugzilla (31.59 KB, image/png) 2012-04-18 08:25 EDT, Robert Munteanu CLA	no flags	Details
Patch/RFC (1.57 KB, patch) 2012-04-24 06:11 EDT, Robert Munteanu CLA	no flags	Details | Diff
mylyn/context/zip (1.22 KB, application/octet-stream) 2012-05-08 11:42 EDT, Steffen Pingel CLA	no flags	Details
Add an attachment (proposed patch, testcase, etc.)		View All

Note
You need to log in before you can comment on or make changes to this bug.

Robert Munteanu CLA 2012-04-18 08:23:48 EDT Description

When displaying attributes with the MultiSelectionAttributeEditor the height of the control is unbounded. The effect is that form sections easily become unbalanced. In contrast with this Bugzilla's CC attribute editor limits the control's height and therefore has a much improved appearance.

Robert Munteanu CLA 2012-04-18 08:24:38 EDT Comment 1

Created attachment 214174 [details]
The unbounded control using the default MultiSelectionAttributeEditor

Robert Munteanu CLA 2012-04-18 08:25:06 EDT Comment 2

Created attachment 214175 [details]
The bounded control used by Bugzilla

Steffen Pingel CLA 2012-04-18 15:44:44 EDT Comment 3

Makes sense. Are you interested in providing a patch?

Fig. 1 Feature request in Bugzilla (Mylyn Tasks project)

[CC3] Conformance. All elements conform to the property to be certified. For requirements, this property typically is "Correctness and consistency": each element in the requirements description is described in a correct and consistent way. Furthermore, the relations between the elements in the requirements description are correct and consistent. The quality criteria in group CC3 for correctness and consistency of traditional up-front requirements from SPCM are:

1. No two requirements contradict each other
2. No requirement is ambiguous
3. Functional requirements specify what, not how
4. Each requirement is testable
5. Each requirement is uniquely identified
6. Each use case has a unique name
7. Each requirement is atomic
8. Ambiguity is explained in the glossary

2.1 A quality framework for just-in-time requirements

Through an analysis of JIT requirements, we evaluate which of the quality criteria from the SPCM are also applicable to feature requests (see Sect. 3). Based on the SPCM, we define the same three overall criteria for JIT requirements. We just rename them to quality criteria (QC):

[QC1] Completeness. All required elements should be present. We consider three levels: basic, required and optional. In that way, we differentiate between requirement elements that are mandatory or nice to have.

[QC2] Uniformity. The style and format should be standardized. A standard format leads to less time for understanding and managing the requirements, because all team members know where to look for what information or how to read, e.g., models attached to the requirement.

[QC3] Conformance. The JIT requirements should be consistent and correct.

The overall QCs are detailed into specific criteria [QCx.x] for each type of JIT requirements. Figure 2 shows the instantiation of the framework with the specific quality criteria for feature requests. These specific criteria will be explained in the next section.

There is, however, another dimension to JIT requirements that clearly differentiates them from traditional requirements, namely the observation that JIT requirements are "initially sketched out with simple natural language statements" [9], only to be fully elaborated when being developed. This leads us to introduce the notion of *time* in our quality framework. For each of the quality criteria, we indicate when it should hold:

***C** At creation time. This criterion should hold as soon as the requirement or the requirement part is created.

***J** Just-in-time. This criterion does not necessarily have to hold when the requirement (part) is created. However, it should hold at a later moment, just-in-time for a certain step in the development process. This could be further detailed by specifying which step is the latest moment for the criterion to hold.

In that way, the framework can be used to give a structured overview of requirement qualities that should be there from the beginning and requirement qualities that should be there just-in-time, see also Fig. 2.

3 Specific quality criteria for feature requests

To answer *[RQ1] Which quality criteria for informal verification apply to feature requests?* we evaluate which of the quality criteria from SPCM (see Sect. 2) are applicable to feature requests (in open source projects). Next to that, we try to come up with new specific criteria based on existing literature about just-in-time requirements and based on our own experience with feature requests in open source projects. Section 4 indicates how the framework could be instantiated for other types of JIT requirements.

3.1 Feature requests in open source projects

A feature request typically corresponds to one requirement ("a documented representation of a condition or capability needed by a user to solve a problem or achieve an objective" [18]). In open source projects, they are used as JIT requirements: a feature request can be specified by users or developers at any moment and the development team will decide whether and at what moment it will be implemented. The feature requests to be implemented will be selected based on priorities set by the developers and/or the users. The initial specification of the feature request might be "incomplete" or "incorrect." The "just-in-time" monicker stems from the idea that this is acceptable, as long as the specification is corrected once the feature request is selected for implementation. This "correction" of an open source feature request is done by adding comments to the original request. In this way, a discussion is created that continues until all parties are satisfied with the implementation of the request. This is different from traditional

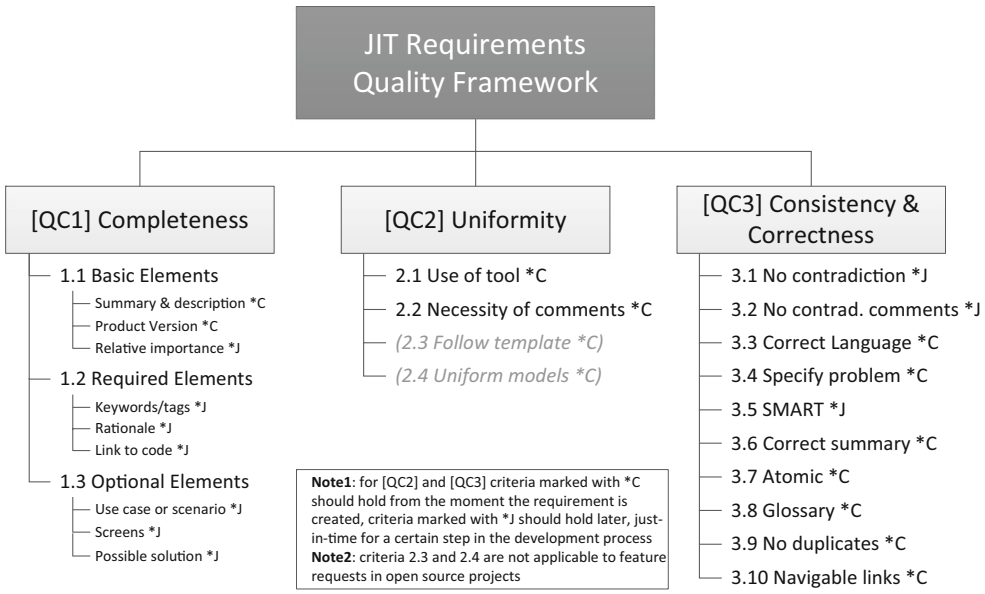


Fig. 2 JIT requirements quality framework, see also Tables 6 and 7

up-front requirements that are specified as a complete, prioritized, correct and consistent set. This set of requirements is usually collected in one big "requirements document" before the implementation starts. Correction of traditional up-front requirements is usually done by producing a new requirements document.

The below sections analyze what the just-in-time specification of feature requests in open source projects means for the quality assessment of those feature requests. The complete list of specific criteria for feature requests is shown in Fig. 2 and is explained in more detail in Tables 6 and 7. In the below analysis, each of the specific criteria is clarified and indicated with "(QC xx *J/*C)." This is the identifier of the specific criterion and an indication if the criterion should hold at creation time (*C) or just-in-time (*J). This indication is also repeated in Tables 6 and 7.

3.2 Completeness for feature requests

Completeness (QC1) in our framework means that all elements of the specification are present. This should not be confused with the completeness of the content of the specification ("did we specify the complete user need?").

The SPCM (see Sect. 2) considers a requirement specification complete if it includes use cases or functional requirements, behavioral properties (e.g., business rules), objects (entity model or glossary) and non-functional requirements.

Alspaugh and Scacchi [2] find that the overwhelming majority of requirements-like artifacts in open source projects may be characterized as what they term *provisionments*. Provisionments "state features in terms of the attributes provided by an existing software version, a competing product, or a prototype produced by a developer advocating the change it embodies. Most provisionments only suggest or hint at the behavior in question; the expectation seems to be that the audience for the provisionment is either already familiar with what is intended, or will play with the cited system and see the behavior in question firsthand."

This form of specification makes it hard to consider the completeness of elements in the same way as we do for traditional up-front requirements. We can, however, look at the attributes of a feature request. Which fields need to be filled for a feature request to be complete? The basic elements (QC1.1) are the ones that define a feature request, such as title (= unique name) and description. The required elements (QC1.2) are the ones that are necessary for management of the feature request: keywords to organize them, a rationale to determine importance and a link to the source code once implemented for traceability. The optional elements (QC1.3) are the ones that add value to the developer when specified by the author, but can also be clarified later on in the process (e.g., by prototyping or asking questions): scenarios, screen mock-ups or hints for a solution.

Based on what we observed while analyzing a large number of open source feature request we determine that at creation time (*C) of the feature request, the author only needs to fill the summary (= title) and description (QC1.1a *C) of what he/she requires and the product (also which version of the product, QC1.1b *C) for which they require it. As can be seen in Table 6, all other fields/attributes can be filled in at a later moment during the development cycle

(*J). Relative importance (QC1.1c *J), rationale (QC1.2b *J), scenarios (QC1.3a *J), screen mock-ups (QC1.3b *J) and hints for a solution (QC1.3c *J) need to be present just before coding starts because they determine when and how things get implemented. Although keywords (QC1.2a *J) should be added by the author of the feature request at creation time, they can be updated during the entire life cycle of the requirement (just-in-time), because new topics can emerge in the discussion of the feature request. For the link to the source code (QC1.2c *J), it is obvious that it can only be added once the feature request is implemented.

3.3 Uniformity for feature requests

Uniformity (QC2) means all requirements have the same format. For traditional up-front requirements, the SPCM [13] defines three levels of uniformity: all elements have the same format, all elements follow company standards, and all elements follow industry standards. For feature requests in open source projects, company or industry standards usually do not apply. For example, feature requests are text only, so no modeling language is used that can be compared to industry use. Most format choices for feature requests are determined by the issue tracker being used (QC2.1 *C). Issue trackers have a number of pre-defined fields that must be filled in and that are always shown in the same way. It is recommended to use the issue tracker from creation time of the feature request (*C) such that all information on the complete life cycle of the feature request is logged in one place. Note that although company standards do not apply, open source projects might have specific uniformity criteria on top of the use of an issue tracker. These uniformity criteria (like "All titles start with a code for the module of the software that the request is for") should also be included as specific criterion under QC2, in addition to the general specific criteria as mentioned in this paper.

The other thing to look at is the "uniformity of comments" (QC2.2 *C). A feature request is entered with summary and description by the author. Then other persons (users or developers) can add comments to the feature request. This is done for discussion of the request or for tracking the implementation of the request. The comments in the different feature requests should be uniform, meaning that they should be "necessary to understand the evolution of the feature request." This is a subjective criterion but it definitely rules out comments like "I am sorry for losing the votes." (Netbeans FR #4619) or "Wooo! Party :) thanks!" (Netbeans FR #186731). Uniformity of comments is needed from creation time of the comment (*C), because no unnecessary comments should be created at all.

In general, we can state that all uniformity criteria should hold from creation time (*C) because at any given moment in the development cycle the team profits from things being specified in a uniform manner. It is best thus to make things uniform from the beginning. For example, if a team uses a specific template for specifying JIT requirements, then it makes no sense to create the requirement in any other format than with this template.

3.4 Consistency and correctness for feature requests

Consistency and correctness (QC3) indicate those criteria that state something on the quality of an individual feature request (correctness), or on the quality of the link between two or more feature requests (consistency).

For each of the eight SPCM quality criteria from Sect. 2, we discuss whether, and how, they apply to feature requests. The resulting quality criteria for feature requests are mentioned between round brackets (QCx.y *J/C), see Fig. 2 for an overview and Table 7 for the description of those criteria.

SPCM CC3.1 No two requirements contradict each other.

Does this hold for feature requests? For a complete set of up-front requirements, contradictions can more easily be established than for the ever-growing set of feature requests in an open source project. As feature requests are typically submitted by many different authors, they often do not have a good picture of the feature requests that have been submitted before, resulting among others in many duplicate requests [14]. The identification of related and possibly conflicting feature requests (QC3.1 *J) is important for developers to determine the correct implementation. Another check that can be done is to see that the comments of a single feature request are not contradicting each other (QC3.2 *J). Ideally, the creation of conflicting requests and conflicting comments is avoided all together, but since this is very hard with an open source online community where every user can submit feature requests and comments, we require that at least just before development starts (*J) all conflicts should be resolved.

SPCM CC3.2 No requirement is ambiguous.

Does this hold for feature requests? As stated by Philippo et al. [28], there are many factors that can decrease the effect of ambiguity and most of them are accounted for in JIT environments. For feature requests, it is not such a problem if the description is ambiguous because there is a habit of online discussion before implementation [31]. Another method that is frequently used in open source projects is prototyping [2]. We can, however, require a basic level of clarity from the author of a feature request at creation time (*C): write in full sentences without spelling/grammar mistakes (QC3.3 *C).

SPCM CC3.3 Functional requirements specify what, not how.

Does this hold for feature requests? As we saw above, the author of a feature request may include hints for implementation of the feature request. As mentioned in Noll and Liu [26], the majority of features is asserted by developers. This makes it more natural that some feature requests are stated in terms of the solution domain [2]. They should however at creation time (*C) *also* specify the problem that needs to be solved (QC3.4 *C), for developers to be able to come up with alternative solutions.

SPCM CC3.4 Each requirement is testable.

Does this hold for feature requests? As Alspaugh and Scacchi [2] state, an open source product that is evolving at a sufficiently rapid pace may be obtaining many of the benefits of problem-space requirements processes through solution-space development processes. This means that the fact that some feature requests may not be specified in a testable way can be compensated by follow-up discussions in comments, extensive prototyping and involving the author of the feature request as a tester later in the process (*J). However, we can require from the author to come up with verifiable feature requests and make the statement as precise as possible (QC3.5 *J): e.g., "I cannot read blue text on an orange background" instead of "I need more readable pages."

SPCM CC3.5 Each requirement is uniquely identified.

Does this hold for feature requests? A unique identifier is added automatically for each new feature request that is entered in an issue tracker (IQ1, see Sect. 3.5).

SPCM CC3.6 Each use case has a unique name.

Does this hold for feature requests? Each feature request should have a unique name ("Summary" or "Title," QC1.1a *C). The summary should be in the same wording as the description and give a concise picture of the description (QC3.6 *C) from the moment the feature request is created (*C).

SPCM CC3.7 Each requirement is atomic.

Does this hold for feature requests? For feature requests in an issue tracker, it is very important that they are atomic, i.e., describe one need per feature request (QC3.7 *C). If a feature request is not atomic from creation time (*C), the team runs into problems managing and implementing it (a feature request cannot be marked as "half done"). The risk also exists that only part of the feature request gets implemented because the comments only discuss that specific part and the other part gets forgotten.

SPCM CC3.8 Ambiguity is explained in the glossary.

Does this hold for feature requests? In open source projects, it is often assumed that users and developers involved are familiar with the terminology of the project (like "DnD" means "Drag and Drop"), but the bigger and older the project gets, the more likely that new unfamiliar persons arrive. It is a good practice to maintain a glossary (e.g., wiki pages) for such project-specific terms and abbreviations (QC3.8 *C) and add any unclear terms in the feature request from the moment it is created (*C). The advantage of online tools is that one can easily link terms used to such a glossary.

From our own experience with open source projects [14], we saw many duplicate entries in the issue trackers. This is a risk because discussions on both duplicate feature requests might deviate if the duplication relationship goes unnoticed (QC3.9 *C). Worst case leads to two different implementations of the same feature. Issue trackers offer functionality to mark feature requests as "DUPLICATE" such that users and developers are always referred to the master discussion. From the very first moment that a duplicate is detected (*C), it should be marked as such, to avoid duplicate work being done.

A last item is about the linking of feature requests. Each link to another feature request should be clearly typed and navigable (QC3.10 *C) from the moment it has been created (*C). If the author of a comment wants to refer to another feature request, then he/she should make sure to insert a URL (some tools do this automatically when using a short code) and to give an explanation why he/she is linking the two requests.

3.5 Inherent qualities of feature requests in issue trackers

As said before in this paper, we consider open source projects that use an issue tracker to store the requirements. In that case, a number of quality criteria are automatically fulfilled. We did not include these *inherent qualities* [IQx] as explicit criteria:

[IQ1] Unique ID: as stated above, an electronic tool will automatically assign a unique ID to each added requirement.

[IQ2] History: electronic tools automatically track all changes to a requirement. This can be viewed directly from the tool's GUI or in the database.

[IQ3] Source: electronic tools automatically log the author of a requirement and the author of each comment.

[IQ4] Status: electronic tools have a separate "Status" field where the status of the requirement can easily be seen. Most tools support a workflow in which the status field is updated (manually or automatically) based on the workflow step the requirement is in.

[IQ5] Modifiable (see Davis et al. [4]): electronically stored requirements are by definition modifiable because the tool provides a structure and style such that individual requirements can easily be changed.

[IQ6] Organized (see Davis et al. [4]): electronic tools offer an easy way to add attributes to requirements. With built-in search options, this allows the tool user to locate individual requirements or groups of requirements.

4 Instantiating the framework for other types of JIT requirements

The previous sections describe the specific criteria for feature requests in open source projects as presented in Tables 6 and 7. However, we advocate the use of the same framework for other types of JIT requirements. This means that the three quality criteria (QC) and the time dimension (*C/*J) remain the same and that just the list of specific criteria (QC) should be adjusted to the specific situation for the team.

An example of this is to customize the specific criteria for the tool that the team is using. Teams that use the quality framework for their JIT requirements should check whether their tool also defaults the six quality criteria in Sect. 3.5. If not, it makes sense for them to include the not supported criteria as extra check in [QC1] or [QC3]. An example of customization is demonstrated in the next paragraph where we analyze which specific criteria would apply for another important type of JIT requirements as discussed in Ernst and Murphy [9]: User Stories.

For a team to customize, the specific criteria for their specific JIT environment are simple in theory (the team decides on the specific checks and metrics) but at the same time difficult in practice (on what grounds would the team decide this?). Our advice is to start with the criteria list in Tables 6 and 7 and first decide which criteria are (not) relevant. Then missing criteria can be found by interviewing team members, by re-evaluating old requirements (why do we think this requirement is good/bad?), or by just applying the framework in practice and improving it on-the-fly.

Then the team should decide whether they need to get an absolute scoring for the JIT requirement, or need to obtain a professional opinion. In most cases, it is more important to find violations (e.g., "Do I see not-SMART statements" or "Do I see irrelevant comments?") and improve the requirement based on that than to get absolute scorings for the requirement. As such, exact metrics might not be needed; a simple yes/no answer for each criterion with the goal to answer all criteria positively could be enough.

4.1 Specific quality criteria for user stories

As a feature request can also be described with one or more user stories [24], we investigate whether the same quality criteria apply. A user story is the agile replacement for what has been traditionally expressed as a functional requirement statement (or use case). A user story is a brief statement of intent that describes something the system needs to do for the user. User stories usually take a standard (user voice) form: "As a ⟨role⟩, I can ⟨activity⟩ so that ⟨business value⟩" [24].

Our conclusion is that most specific quality criteria we defined for feature requests (see Fig. 2) are also applicable to user stories. In Table 1, we detail the differences.

[QC1.x′] indicates that the criterion has the same title as for feature requests, but with different elements that should be part of the user story. [QC2.3] is added for user stories because the user voice form as mentioned before is specific for user stories. [QC2.4] is added for user stories because we did not see any feature requests in open source projects that have attachments with detailed specification models, but we have spoken to many companies that use this mechanism to provide more detail for their user stories; this form of specifying user stories is also mentioned by Leffingwell [24]. Leffingwell introduces INVEST(see [32]) as the agile translation of SMART, hence [QC3.5′]. The other [QCx.x] are valid for user stories without changes.

A team working with user stories should decide which specific quality criteria apply to their practice. If, e.g., the product owner is in a remote location, then the quality criteria for documented user stories should be applied. If, e.g., user stories are only documented as a "user voice statement" and comments are discussed off-line, then [QC2.2] and [QC3.2] do not apply. The quality criteria could be incorporated into the team's "Definition of Ready" (see [29]) that determines when a user story is ready to be included in the planning for the next development iteration.

5 Empirical evaluation of the framework for feature requests: setup

The result of our analysis on *[RQ1] Which quality criteria for informal verification apply to feature requests?* is the framework as presented in Sect. 3, see also Fig. 2. To answer *[RQ2] How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?*

Table 1 Specific criteria for user stories

[QC1.1′*C]	Basic Elements: role, activity, business value ("Who needs what why?") instead of summary and description
[QC1.2′*J]	Required Elements: acceptance criteria or acceptance tests to verify the story instead of rationale (already as business value in QC1.1′)
[QC1.3′*J]	Optional Elements: the team could agree to more detailed attachments to certain user stories (e.g., UML models) for higher quality
[QC2.3 *C]	Stories Uniform: each user story follows the standard user voice form
[QC2.4 *C]	Attachments Uniform: any modeling language used in the attachments is uniform and standardized
[QC3.5′*J]	INVEST: User stories should be Independent, Negotiable, Valuable, Estimable, Small, Testable [32]

and *[RQ3] What is the level of quality for feature requests in existing open source projects as measured by our framework?* we followed two different approaches. Section 5.1 describes the setup of an initial evaluation of our framework that consisted of interviewing eight practitioners. Section 5.2 describes the setup of a case study in which we applied the framework to 620 feature requests from three open source projects.

5.1 Interview Setup

In order to get an initial feeling of the practicality (usability, completeness and relevance) of our framework for the quality assessment of feature requests, we decided to interview eight practitioners from the Dutch agile community. They were sourced through our personal network. We chose the agile community to get a first idea of whether the framework would also be useful for companies using JIT requirements. The participants are not necessarily experienced in open source projects, but are familiar with both the traditional up-front requirements engineering and the JIT requirements engineering. This makes them well suited to comment on the underlying principles of our framework. The interview consisted of two parts:

1. General questions on JIT requirements quality, including an exercise to evaluate feature requests from the Firefox (www.mozilla.org/firefox) project;
2. An exercise to use our quality framework on feature requests from the Bugzilla project (www.bugzilla.org), followed by questions to rate the quality framework.

The first part of the interview was done with minimal introduction from our side and above all without showing the participants our framework. For the second part, we have turned our quality model into a checklist (in MS Excel) for the participants to fill in. For each check, the answer set was limited. When each check is filled in, the spreadsheet automatically calculates a score for each of the quality criteria and an overall score for the quality of a single feature request (LOW/MEDIUM/HIGH), see Sect. 5.3 for the inner workings.

The feature requests used for the exercise were manually selected by the first author using the following selection criteria: a substantial but not too big amount of comments (between 7 and 10) in the feature request, feature request has been implemented, contents of the feature request are not too technical (understandable for project outsiders). This last criterion is also why we selected the two projects: both Firefox and Bugzilla are well-known (types of) tools such that project outsiders should be able to understand or recognize the features. The feature requests were accessed online.

The data sets (five feature requests from Firefox, ten from Bugzilla), Excel checklist and interview questions can be found online [12].

5.2 Case study design

Our framework also allows us to get an insight into the quality of feature requests of existing open source projects. As such, we asked a group of 93 software engineering students to apply the checklist (as described in the previous section) to a large number of feature requests. As a side effect, we also get additional qualitative feedback on the practicality of the checklist.

The three open source projects that we used in this case study are:

- Eclipse MyLyn Tasks http://projects.eclipse.org/projects/mylyn.tasks: 100 feature requests selected out of around 400 total
- Tigris ArgoUML http://argouml.tigris.org: 210 feature requests selected out of around 1275 total
- Netbeans http://netbeans.org: 310 feature requests selected out of around 4450 total

The projects [15] were selected because: (1) they are mature and still actively developed; (2) they differ in order of magnitude in terms of number of feature requests; (3) they use Java as a programming language (important because some feature requests contain source code fragments); and (4) they use Bugzilla as an online tool to manage feature requests. The feature requests were

selected randomly based on the status (preferably "CLOSED" since then we have the complete feature request history) and the number of comments (preferably between 7 and 12 because this yields a proper text size to analyze manually). For Mylyn Tasks, being a smaller project, we had to extend the criteria to status not equal to "NEW" and between 3 and 12 comments. All feature requests were accessed online in Bugzilla.

The application of the checklist was assigned to a group of 93 final-year computer science students majoring in software engineering at the Fontys Applied University in the Netherlands as part of one of their courses. During their studies, they have gathered a proficiency in Java programming and worked with Eclipse, UML tools and Netbeans. Therefore, we assume that they possess sufficient background knowledge to have a high-level understanding of the feature requests presented to them. Each student was assigned 20 specific feature requests. Furthermore, each feature request was assigned to 3 different students to be able to compare answers from different raters.

For the purpose of easy online data collection, we have transformed our framework for feature requests into a Google Forms questionnaire. Some specific criteria were not enclosed in the questionnaire because the answer would always be the same for all feature requests (e.g., "does the feature request have a title?"), resulting from the fact that all analyzed feature requests are entered in the Bugzilla issue tracker that has mandatory fields. The remaining criteria were transformed into multiple choice questions with a short explanation for each question. We have included an additional comment box ("Opmerkingen" in Dutch) for the students to fill in any free-format remarks. So each student had to fill in 20 questionnaires, one questionnaire for each feature request assigned to them.

We received 1699 filled in questionnaires from the students through Google Forms: 83 students completed the assignment for 20 feature requests, 2 students did only 19 feature requests, and 1 student did only 1 feature request. We cleaned up the data by correcting wrongly entered student numbers and feature request numbers (typos and use of network ID instead of student number). We also corrected small mistakes that four students reported by e-mail (because students were not able to resubmit already sent questionnaires). We transformed this Excel file back into our original criteria list by adding the criteria previously not enclosed (because of standard Bugzilla as explained in previous paragraph), and we added the scoring algorithm explained below. In that way, we obtained 1699 "scorings" of the 620 feature requests. On average, each feature request was scored 2.7 times; 10 feature requests were scored only once.

The list of feature requests used, the Google Form questionnaire and the resulting feature request scorings can be found online [12].

5.3 Scoring setup

In this section, we explain the scoring model that was used.

In Tables 6 and 7, it is indicated for each criterion what the outcome can be (column "Metric"). For each specific criterion, we translate the answers into percentage scorings. For a criterion with two possible answers, the score is either 0 % (low quality) or 100 % (high quality). For a criterion with three possible answers, there is also a 50 % score (medium quality). As indicated in Table 6, QC2.2 directly results in a percentage (of relevant comments).

We calculate some higher-level scorings as indicated in Table 2. In particular, the overall score for [QC3] is calculated by taking the simple average of all percentage scores, because in our opinion no single criterion is more important for correctness than the other criteria. For the final score, we first look at the [QC1] score. If QC1.1 or QC1.2 score below 100 % (meaning that basic or required elements are missing), the final score is always 0 %. Otherwise, the final score is a weighted average of [QC1.3], [QC2.2] and [QC3]. QC3 has a weight of 3 in this average as we feel that the "Correctness" is the most contributing factor to the overall quality of the feature request. [QC1.3] are "optional elements," comments (QC2.2) are just a small factor for uniformity, and [QC3] really looks at if everything that *has* been written is written in a correct way. The overall quality score is considered "HIGH" when equal to or above 75 %, "LOW" when below 55 % and "MEDIUM" otherwise.

This scoring algorithm is based on our professional opinion on what is appropriate for feature requests in open source projects. For other situations, different rules or a (different) weighted average might be more appropriate.

Table 2 Quality score calculation

Individual scores	
Yes, Very much, N/A	100 %
A little bit	50 %
No, Not at all	0 %
Overall scores	
QC1.1, QC2.1	Always 100 % for open source feature requests
QC1.2, QC1.3	Average([QC1.x a] till [QC1.x c])
QC2.2	percentage of relevant comments
QC3	Average(QC3.1 till QC3.10)
Total	IF ((QC1.1 < 100 %) OR (QC1.2 < 100 %)) THEN 0 % ELSE $\frac{[QC1.3]+[QC2.2]+3*[QC3]}{5}$

6 Interview results

As mentioned in Sect. 5.1, we first wanted to get an initial feeling of the practicality (usability, completeness and relevance) of our framework for the quality assessment of feature requests. We interviewed eight practitioners from the Dutch agile community.

6.1 Part one: background and JIT requirements quality

All participants are experienced IT specialists with good knowledge of JIT requirements engineering. They work for five different Dutch companies in the area of software development and quality consulting; their roles in agile projects vary from coach, to trainer or consultant. Most of them also have experience as analyst or tester in agile projects. All participants mention user stories as a format for JIT requirements, but also use cases, features and wireframes (i.e., screen blueprints). Some participants mentioned that they also consider informal communication as being part of "the requirement." We made clear that for the purpose of our framework, we only consider the *written* part.

All participants agree that JIT requirements should fulfill certain quality criteria. This helps the understanding within the team and is important for traceability or accountability toward the rest of the organization. When asked for a list of quality criteria, the participants do not only mention quality criteria like the ones in our framework (SMART, not ambiguous, sufficiently small, atomic, following company standards/template), but also include process-oriented criteria like "approved by the product owner," "estimated in hours."

When asked to score 2 feature requests from the Firefox project (175232 and 407117) as HIGH/MEDIUM/LOW quality (without prior knowledge of our framework, just based on professional opinion), the participants do not always agree on the exact score, but they consistently score 175232 lower than 407117.

6.2 Part two: our JIT quality framework

Each participant filled in the checklist for at least two different feature requests from the Bugzilla project to get some hands-on experience with the checklist. The goal of this exercise was not to collect quantitative data, but to get qualitative feedback from the participants on the checklist.

Four participants mention "# of relevant comments" (QC2.2), and 2 participants mention "SMART" (QC3.5) as checks that are unclear or difficult to fill in. For [QC2.2], they find it difficult to determine whether a comment is "relevant" or not, and for [QC3.5], they have difficulties determining the overall score on 5 criteria (Specific, Measurable, Acceptable, Realistic, Time-bound) in one check. We agree that these two checks are quite subjective, but we chose not to objectivize them in further detail. As one participant remarks: "I am in favor of checklists but quantifying in too much detail triggers discussions on scores and weighing. The discussion should be on the content of the requirement." This is what we also conclude in our Sect. 4 about customization of the framework.

When asked to rate the score calculated by the Excel sheet for each feature request (LOW/MEDIUM/HIGH), the opinions vary. On a scale from 1 (no match at all with my personal opinion) to 5 (great match), all ratings have been given, although 6 out of 8 participants rate 3 or higher. This shows that most participants consider the final score of the model to be relevant. Yet, we also accept that our initial weighting scheme for the checklist requires fine-tuning for future use.

For example, in the checklist used in this interview a feature request always scores LOW if one of the basic (QC1.1) or required (QC1.2) elements is missing and not all participants agree with this choice. They, for example, argue that a feature request with a missing "Rationale" (QC1.2b) can still be a correct feature request if it is self-explanatory enough. We agree. We added a scoring algorithm to help the participants in judging feature request quality, but the scoring algorithm should not be taken as an absolute judgement (one participant: "A practical checklist like this always helps, but I am not sure how useful it is to calculate a final score from the individual checks."). As stated before, the checklist is very useful as a reminder of what to check when looking for good feature request quality. It is the reviewer or author of the requirement that can still decide how serious a violation is in the given situation, e.g., by marking it as "N/A."

Some participants answered that they would like to add topics such as non-functional impact (e.g., usability, performance), business value and domain models. We see this as valuable suggestions for practitioners customizing the checklist for their JIT projects. We feel that these topics are not applicable for feature requests in open source projects. As one participant mentions "It is refreshing that this checklist is tailored for this specific situation. The ultimate result would be to know how to construct such a tailored checklist." Section 4 shows how this customization could be done.

But why would teams do the effort of including such a checklist in their development process? All participants rated the checklist as helpful when judging the quality of a feature request (compared to using "gut-feeling"), see Fig. 3. They valued the help of the checklist to not forget

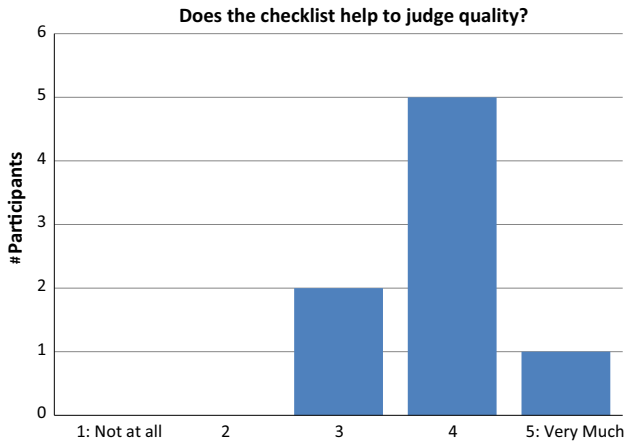


Fig. 3 Participants rate the checklist as helpful

criteria, to base their opinion on facts, to use it as an education for new team members and to standardize the review process. One participant (that rated the checklist as "Very Much" helpful) nuances this by stating "It is not always the case that high-quality requirements lead to high-quality products. The checklist is helpful but just a small part of all factors that influence final product quality." This is a valid point. Our study shows that also in JIT environments requirements quality is considered important, but that there is no one-size-fits-all solution. All participants confirmed that our framework is a good starting point to get to a tailored process for quality assessment of JIT requirements.

7 Case study results: findings on quality of feature requests in open source projects

To collect experiences from the use of our framework in practice, we applied it to 620 feature requests from the Netbeans, ArgoUML and Mylyn Tasks open source projects. For details on this setup, see Sect. 5.2. In this section, we highlight the aggregated findings. First, we explain how we cleaned the data to identify any potential participant that did not take the questionnaire seriously. Then we present some aggregated findings on feature request quality. Lastly, we describe what we learned about our framework for feature requests.

7.1 Data cleaning

When collecting data through surveys, data quality can be a concern. In particular under conditions of obligatory participation, so-called careless responses can be a worry [25]. To this end, we performed two checks on the response to identify suspect participants. These suspect participants could either not have taken their participation in the survey seriously, or they might lack a basic understanding. The two checks are:

1. We used criterion [QC3.9] (No duplicate requests) as a "control question." The answer to this question is not subjective (each feature request in Bugzilla is clearly marked as "CLOSED DUPLICATE"). We remove all 17 participants that have answered this question wrongly in one or more cases.
2. For each participant, we calculated the absolute distance of his questionnaires to the average questionnaires for the same feature requests. This resulted in absolute distances between 73 and 308 (where maximum distance is 340 if a participant always answers completely the opposite of the other participants that have scored the same feature request) with an average distance of 136. We remove the 5 outlier participants (that have a distance of more than twice the standard deviation from the average, i.e., more than 214) and note that 3 out of these 5 participants are also included in the 17 removed participants from the previous check.

In the aggregated analysis in the next paragraph, we present the results after removing all of the suspect participants (19 in total). This might be more than needed for the purpose of our analysis, but makes us more confident that we are working with valid data. This leaves 67 participants filling in 1319 questionnaires for 570 feature requests (200 ArgoUML, 80 Mylyn Tasks, 290 Netbeans). Out of these, 90 feature requests have been scored by only 1 participant. On average, each feature request is scored by 2.3 participants.

7.2 Aggregated results

The aggregated answers and resulting scorings (see Sect. 5.3) are given in Table 3. A lot of interesting observations can be made from this table. We highlight a few observations in this section:

Overall Score The overall score (= the average score of all feature requests in the project) for the three projects is quite low (5 % for Netbeans and ArgoUML, 15 % for Mylyn Tasks). This is mainly due to the fact that a lot of feature requests score 0 % because of missing keywords, missing rationale or missing link to source code. Note that the QC3 "correctness" score is quite high for each of the three projects. This led us to conclude that the way we calculate the overall score (scoring 0 % if basic or required elements are missing) does not provide particular insight into overall feature request quality.

Completeness As indicated in the previous observation, not all required elements are present: on average 54 % (QC1.2). For optional elements on average, 36 % is

Table 3 Scorings from open source projects

Specific Criterion	Answer	MT	AU	NB	Total
QC1.1 Basic elements	Average score	100 %	100 %	100 %	100 %
QC1.2a Keywords	Not at all	53 %	87 %	88 %	82 %
	A little bit	44 %	11 %	11 %	18 %
	Very much	3 %	1 %	0 %	1 %
QC1.2b Rationale	No	32 %	33 %	30 %	31 %
	Yes	68 %	67 %	70 %	69 %
QC1.2c Link to source code	N/A	19 %	17 %	19 %	18 %
	No	41 %	13 %	13 %	18 %
	Yes	39 %	70 %	68 %	64 %
QC1.2 Required elements	Average score	51 %	54 %	54 %	54 %
QC1.3a Scenario	No	59 %	61 %	47 %	54 %
	Yes	41 %	39 %	53 %	46 %
QC1.3b Mock-up	No	77 %	90 %	94 %	90 %
	Yes	23 %	10 %	6 %	10 %
QC1.3c Solution	No	46 %	48 %	47 %	47 %
	Yes	54 %	52 %	53 %	53 %
QC1.3 Optional elements	Average score	39 %	34 %	37 %	36 %
QC2.1 Use of tool	Average score	100 %	100 %	100 %	100 %
QC2.2 Relevant comments	% relevant	66 %	61 %	60 %	61 %
QC3.1 Contradicting requirements	(Not scored)	N/A	N/A	N/A	N/A
QC3.2 No Contradicting comments	Not at all	0 %	2 %	4 %	3 %
	A little bit	17 %	22 %	26 %	23 %
	Very much	82 %	76 %	70 %	74 %
QC3.3 Correct language	Not at all	5 %	6 %	6 %	6 %
	A little bit	27 %	24 %	29 %	27 %
	Very much	68 %	70 %	65 %	67 %
QC3.4 Problem stated	No	24 %	23 %	18 %	21 %
	Yes	76 %	77 %	82 %	79 %
QC3.5 SMART	Not at all	33 %	55 %	45 %	46 %
	A little bit	52 %	37 %	46 %	44 %
	Very much	15 %	8 %	9 %	9 %
QC3.6 Title correct	No	16 %	30 %	16 %	21 %
	Yes	84 %	70 %	84 %	79 %
QC3.7 Atomic	No	14 %	10 %	8 %	10 %
	Yes	86 %	90 %	92 %	90 %
QC3.8 Clear terms	Not at all	3 %	6 %	5 %	5 %
	A little bit	31 %	30 %	30 %	30 %
	Very much	66 %	64 %	65 %	65 %
QC3.9 No duplicate	No	12 %	28 %	20 %	22 %
	Yes	88 %	72 %	80 %	78 %
QC3.10 Links clear	No	13 %	11 %	9 %	10 %
	Yes	87 %	89 %	91 %	90 %
QC3 Consistency and correctness	Average score	80 %	75 %	78 %	77 %
Total	Average score	15 %	5 %	5 %	7 %

NB Netbeans, *AU* ArgoUML, *MT* Mylyn Tasks

present (QC1.3). For the individual elements, we see that, for example, the use of mock-ups and keywords is not so common in the three projects. Mylyn Tasks does clearly better than the other two projects here. This explains the higher overall quality score of 15 % for Mylyn Tasks.

Uniformity The percentage of relevant comments (QC2.2) is similar for all three projects: around 60 %. This means that out of every three comments (in total more than 4500 comments were read by the participants) one is irrelevant. We assume this has implications for the understandability of the feature request.

Creation time The feature requests that have been selected are all in status "Closed." This means that the QC3 overall scorings in Table 3 are based on both the creation time (*C) and the just-in-time (*J) criteria. When we only average the seven creation time criteria for QC3, we see a clear difference in scoring:

	Mylyn Tasks (%)	ArgoUML (%)	Netbeans (%)	Total (%)
QC3 *C	83	80	84	82
QC3 *C + *J	80	75	78	77

This shows that the quality of the feature requests slightly deteriorates for all three projects from the moment it is created until it is closed. In the projects, we analyzed this is mainly due to the fact that according to our framework a feature request does not need to be SMART (QC3.5 *J) when initially created. However, a lot of feature requests (on average 46 %) have been scored "not SMART at all," leading to a lower just-in-time quality score.

Our overall conclusion from the data gathered is that although sometimes important elements are missing (QC1) and comments are not always relevant (QC2) in fact the overall correctness/consistency of the feature request is quite high (based on [QC3] scorings).

7.3 Feedback on the use of the framework for feature requests

The participants provided us additional insights as well: they submitted remarks by e-mail or in the comment field of the questionnaire. The most important things we learned about our framework from these remarks are:

- Five participants are confused about the term "Title" we used in the questionnaire for [QC3.6]. This is not a problem of our framework itself, but we should have better explained in the questionnaire that by "Title" we meant the statement in bold just after the identifier of the feature request (see Fig. 1) or for the ArgoUML project the field called "Summary."
- Five participants comment that they consider the feature request they evaluated to be a bug report. This misclassification is a well-known phenomenon. According to Herzig et al. [16], only 3% of feature requests is misclassified. This means that our results are not greatly influenced by this.
- Three participants were confused about the question about keywords (QC1.2a) because the ArgoUML project does not have this field in the feature request. We could have mentioned this to the participants beforehand. This is again not a problem of our framework.
- We learned that some questions are more subjective than others, see Table 4. For example on the question about mock-ups (QC1.3b), all students agree in 90 % of the cases. On the question whether the feature request is SMART, all students agree in only 49 % of the cases. As explained in Sect. 4, this subjectiveness is not a problem, because we are usually not interested in absolute scorings but in finding shortcomings in the JIT requirements. Even if only one person thinks there is a shortcoming, it might be worth to look into the details of it.

Overall the remarks from the participants did not make us change our framework or checklist. The remarks could all be avoided by a more detailed explanation of the structure of the different feature requests in Bugzilla and by a more detailed explanation of the questions in the questionnaire. We did organize a general meeting for this explanation, but participants that were not present during that meeting had to rely on the explanation in the questionnaire itself.

8 Discussion

In this section, we highlight some recommendations for practitioners, based on our findings. Subsequently, we revisit the research questions and discuss limitations of our research approach.

8.1 Practical consequences for practitioners

Looking at the data we collected from the three open source projects, we come to a number of recommendations for practitioners authoring feature requests. These are our top 5 to focus on for improving feature request quality:

1. Enter keywords/tags for each feature request, making them more easily retrievable for future reference. This will likely at the same time help to reduce the number of duplicate feature requests.
2. Indicate the problem that needs to be solved and/or include a rationale for the feature request (why is this feature needed?). This will help developers to better understand the feature request and thus will increase

Table 4 Subjectivity scores per question

Question	Unanimity	
	Two students agree (%)	All students agree (%)
[QC1.2a] Keywords	94	84
[QC1.2b] Rationale	85	67
[QC1.2c] Codelink	81	62
[QC1.3a] Scenario	83	59
[QC1.3b] Mock-up	95	90
[QC1.3c] Solution	82	58
[QC2.2] Relev. comm.	20	20
[QC3.2] Contr. comm.	79	58
[QC3.3] Corr. lang.	72	52
[QC3.4] Problem	88	72
[QC3.5] SMART	73	49
[QC3.6] Summary	85	68
[QC3.7] Atomic	93	85
[QC3.8] Terms	73	50
[QC3.9] Duplicate	97	95
[QC3.10] Corr. Links	90	81
Total	81	65

the chance of the solution matching the actual need of the feature request author.

3. Further increase the understandability of the feature request by adding one or more of additional items: screen mock-ups, descriptions of use-case scenarios or possible solutions.
4. Be as precise as possible in what you write. There is no need to fully specify all aspects at creation time of the feature request, but what is written should not be unnecessarily vague or ambiguous (e.g., avoid the use of abbreviations or terms that are not quantified, try to write SMART). Being precise from the start avoids wasting time on discussions to clarify statements during development.
5. Avoid irrelevant comments. This clutters the discussion on the feature request and thus hinders its correct implementation. Even better would be if the issue tracker used has a way to "categorize" or "hide" comments, for easy retrieval of the relevant ones for the task at hand. We did not see such an option in the projects we considered.

8.2 Research questions

In this section, we revisit the research questions one by one.

[RQ1] We started by asking: *Which quality criteria for informal verification apply to feature requests?* We have developed a framework for quality criteria for JIT requirements based on earlier work on traditional upfront requirements, our experience with feature requests in open source projects and analysis of literature on just-in-time requirements. We have instantiated this framework for feature requests in open source projects.

Table 5 Mapping between Davis et al. [4] and our framework

Davis et al. [4]	JIT requirements quality framework
Unambiguous	[QC2.4], [QC3.3], [QC3.8]
Complete	[QC1]
Correct (contributes to satisfaction of some need)	[QC3.4], [QC1.2]—Rationale
Understandable	[QC1.3], [QC3.3]
Verifiable	[QC3.5]
Internally consistent	[QC3.1], [QC3.2], [QC3.6]
Externally consistent	N/A
Achievable	[QC3.5]
Concise	[QC2.2]
Design independent	[QC3.4] (Solution might be included)
Traceable (facilitates referencing of individual req.)	[QC3.7], [IQ1]
Modifiable (table of contents and index)	[IQ5]
Electronically stored	[QC2.1]
Executable (dynamic behavioral model can be made)	N/A
Annotated by relative importance	[QC1.1]—Relative importance
Annotated by relative stability	N/A
Annotated by version	[QC1.1]—Version
Not redundant	[QC3.9]
At right level of detail	[QC1]
Precise	[QC3.5]
Reusable	N/A
Traced (clear origin)	[QC1.2]—Link to code, [IQ2], [IQ3], [IQ4]
Organized	[QC1.2]—Keywords, [QC3.10], [IQ6]
Cross-referenced	[QC3.1], [QC3.9], [QC3.10]

How does our framework compare to quality criteria for traditional requirements? To highlight the differences with traditional requirements, we compare it to the list of Davis et al. [4]. Davis et al. performed a thorough analysis of qualities of a software requirements specification (SRS, an up-front document). Table 5 shows that four quality criteria apply to traditional requirements, but not to just-in-time requirements (we have marked them as "N/A" in Table 5):

1. *Externally consistent = no requirement conflicts with already baselined project documentation.* The

Table 6 JIT quality framework for feature requests—[QC1] and [QC2]

ID	Criterion	Description	Metric	*C/*J
[QC1] Completeness				
QC1.1 Basic Elements				
a	Summary and Description	The "Description" contains the provisionment and the "Summary" (or "Title") gives a clear short version of the provisionment	Yes/no	*C
b	Product Version	Indicates for which software product and version the provisionment holds	Yes/no	*C
c	Relative importance	The relative importance of the feature request should be clear. Examples of this are a "Priority" or "Severity" field or a voting mechanism (feature requests with more votes are more important)	Yes/no	*J
QC1.2 Required Elements				
a	Keywords/tags	For organization purposes (easily finding related requests) a feature request should be tagged with keywords	Yes/no	*J
b	Rationale	Each feature request has a justification. The author should specify why this feature request is important for him/her. This helps the implementation team in deciding on the priority of the feature request	Yes/no	*J
c	Link to source code for fixed requirement	For solved feature requests indicates in which version of the source code it has been solved. This can be done through a manual comment, or through an automated one generated from the source code management system. Ideally, the feature request has a separate field to track this.	Yes/no	*J
QC1.3 Optional Elements				
a	Use case or scenario	The author specifies the exact steps that he/she is missing in the current version of the software. What is the trigger for the missing functionality and what are scenario's in which the functionality would be useful?	Yes/no	*J
b	Screens	The author could clarify the screens in the existing system that he/she wants to be changed by adding screen shots of the current situation and/or screen mock-ups of the desired situation.	Yes/no	*J
c	Possible solution	The author could add a complete solution as "Attachment" (patch), but could also specify hints for a possible solution in comments	Yes/no	*J
[QC2] Uniformity				
QC 2.1	Issue tracker or other tool should be used	An issue tracker ensures that feature requests are stored in an uniform way, at least with respect to the attributes that are present for the feature request. Of course it remains up to the author to correctly fill those fields	Yes/no	*C
2.2	All comments are necessary	All comments are necessary to understand the evolution of the feature request. The addition of irrelevant comments (e.g., thank you notes or instructions how to behave in the project) makes it more difficult for people to understand the totality of it	The percentage of relevant comments	*C

provisionments (see Sect. 3.2) are not specified with respect to external documents (Davis et al. define this as "already baselined project documentation"), but with respect to existing systems. If in specific situations external documents are relevant, the team should add one or more criteria to [QC3] to check the consistency.

2. *Executable = there exists a software tool capable of inputting the SRS and providing a dynamic behavioral model.* In open source (JIT) projects, this is not done by up-front extensive specification, but by prototyping or frequent releases.
3. *Annotated by relative stability = a reader can easily determine which requirements are most likely to change.* Open source (JIT) projects have embodied change as a known fact. They solve this with short iterations and reprioritization of requirements for each iteration. That is why in open source projects we do not need a special attribute to specify change-proneness up-front.
4. *Reusable = sentence, paragraphs and sections can easily be adopted or adapted for use in a subsequent SRS.* Since open source feature requests are necessarily incomplete ("provisionments"), it makes no sense to reuse them.

All criteria from our resulting framework, see Tables 6 and 7, are in one way or another present in the work of Davis et al. [4], see Table 5. However, we have adjusted the description of each criterion to JIT feature requests; e.g., for the criterion "Design independent," Davis et al. explain that a maximum number of designs should exist to

Table 7 JIT quality framework for feature requests—[QC3]

ID	Criterion	Description	Metric	*C/*J
[QC3] Consistency and Correctness				
QC 3.1	No contradicting feature requests	It is difficult to completely avoid conflicting requests, but they should not go unnoticed. A link can be made through comments and one of the feature requests should be "Closed" to avoid implementation of the wrong request	Yes/no	*J
3.2	No contradicting comments	Each contradicting comment is clarified in later comments. It should be clear what the correct interpretation of contradicting comments is	Very much, a little bit, not at all	*J
3.3	Correct language	Feature requests should be written in full sentences without hindering spelling or typing errors	Very much, a little bit, not at all	*C
3.4	Specify problem	Feature requests may include (hints for) a solution, but should always describe the problem that needs to be solved. This helps developers to think about alternative solutions	Yes/no	*C
3.5	SMART	A feature request can be more quickly and easily resolved if it is Specific, Measurable, Acceptable, Realistic and Time-bounded (SMART, see [6])	Very much, a little bit, not at all	*J
3.6	Correct summary	The summary should be a brief statement of the needed feature. It should be clear from the summary what the feature request is about. The description should give added value to the summary.	Yes/no	*C
3.7	Atomic	Each feature request should contain only one requirement	Yes/no	*C
3.8	Glossary	Each unclear term or abbreviation should be explained in the feature request or in a separate "glossary"	Very much, a little bit, not at all	*C
3.9	No duplicate requests	Due to the nature of open source projects there will always be some duplicates (also users not so familiar with the project get the rights to enter requests). At least the duplicates should be identified, properly linked, and the master should be indicated.	Yes/no	*C
3.10	Navigable links	Links to other feature requests should be navigable (by clicking on the link) and it should be explained what the type of the link is	Yes/no	*C

satisfy user needs. This means that according to the definition of Davis et al. the requirement should just describe the problem and not already present a design solution because that would decrease the number of possible designs. We have included "[QC3.4]—Specify Problem," but we specifically allow the user to *also* specify design solutions (QC1.3c), as this is common practice in open source projects where users are also developers.

Did we introduce any new criteria for just-in-time requirements? If we do the comparison with the list of Davis et al. [4] the other way around, we also see a few differences:

1. *Complete* Davis et al. refer to the completeness of the set of requirements (before development starts). Since this is not a goal for just-in-time requirements engineering, we use completeness in the sense of "are all basic/required/optional elements of one single requirement present" (QC1.x).
2. *Concise* For open source feature requests, comments are added to the original requirements. We have translated the conciseness of a requirement as described by Davis et al. to the demand that the comments that are added are concise (QC2.2).
3. *Internally Consistent* In a similar manner, we have added a demand for internal consistency of the comments that are added to one single requirement (QC3.2). And we also added a demand for consistency between the feature request and its summary title (QC3.6).
4. *Unambiguous* Instead of this one single criterion from Davis et al., we have three related criteria: use of correct language (QC3.3), use of a glossary (QC3.8) and use of uniform attachments (QC2.4).
5. *SMART/INVEST (QC3.5)* Some of the separate components of SMART and INVEST have been mentioned by Davis et al. (Verifiable = Testable, Achievable = Realistic, Precise = Specific), but most of them are new in our model. INVEST is specific for user stories, but in our opinion it can also be used for other types of just-in-time requirements.
6. *Atomic (QC3.7)* The demand that a single requirement should be atomic is not mentioned explicitly by Davis et al., but of course it helps to make requirements traceable.
7. *Cross-referenced* We have translated the cross-referenced criterion of Davis et al. into two separate criteria about "linked duplicates" (QC3.9) and "navigable links" (QC3.10)

For our user stories customization (see Sect. 4), we added the aforementioned "[QC2.4]—uniform attachments" and one more criterion:

8. "[QC2.3]—Follow template" ("As a ⟨role⟩, I can ⟨activity⟩ so that ⟨business value⟩" [24]). This is unique for user stories since traditional requirements and feature request do not follow standard templates (methods that advocate this are not widely used).

Overall, our analysis of the literature in just-in-time requirements and our experience with just-in-time requirements led to 8 instances of "additional criteria" or "new" interpretations of existing criteria. And of course we have added the time dimension to each of the criteria by specifying creation time (*C) or just-in-time (*J).

[RQ2] Then we wondered *How do practitioners value our list of quality criteria with respect to usability, completeness and relevance for the quality assessment of feature requests?* The overall evaluation of the framework for open source feature requests was positive. The interviews with practitioners have made it clear to us that specific situations need some fine-tuning of the specific criteria and the scoring. Our framework caters for that kind of specific tailoring and we have given some hints om how to approach this. The questionnaires filled in by the practitioners only resulted in some minor remarks. We concluded that all of them could be solved by better explaining the questions. The feedback from practitioners did not make us change our basic framework.

Although we have used open source feature requests to evaluate the framework, we see the value of our framework not so much for open source teams to apply it, because in open source projects it is hard to make the whole community comply to quality standards. The interviews with the practitioners indicated to us that a translation of the quality criteria to specific JIT industry settings is both feasible and useful. The practitioners valued our framework as a structured approach for doing this.

[RQ3] Finally, we answered *What is the level of quality for feature requests in existing open source projects as measured by our framework?*. We presented a table with average scorings for each of the quality criteria. These scorings were collected from over 550 feature requests that were rated using our framework. The overall impression is that the score for [QC3] is quite high (77 %), but there is more room for improvement in the scorings for [QC1.2] (54 %), [QC1.3] (36 %) and [QC2.2] (61 %). We also concluded that the quality at creation time (*C, 82 %) was slightly better than at the time of scoring (*J, 77 %). From our results, we have distilled a set of recommendations that makes this research actionable for practitioners.

8.3 Limitations

In this section, we discuss the threats that can affect the validity of our study and show how we mitigated them.

Internal validity regards threats inherent to our study. We assume that the results gathered in our case study with the 86 final-year software students are reliable because any negative effects from participants not understanding or not cooperating would average out over such a large number of entries (randomly selected and randomly divided over the software engineers). In order to minimize the risk of data pollution, we cleaned the data (see Sect. 7.1) and we have investigated how unanimous the answers from the students are (see Table 4).

With regard to the interview setup that we have detailed in Sect. 6, it might be that the participants were influenced by how we approached them or how we explained our framework to them, the so-called *observer-expectancy effect*. We tried to be as neutral as possible toward the interviewees and clearly explained them that we were expecting them to provide honest feedback, which would be most beneficial for our investigation.

Also note that for the scorings of the feature requests in open source projects we assume that all communication around the feature request is logged in the issue tracker. We might have missed some data related to the feature requests that was documented, e.g., on mailing lists or discussion forums. However, if we missed the data, every reader of the feature requests would have missed it, since there is no link to it in the feature request. So from a quality assessment perspective, it makes sense to only look at the data in the issue tracker and judge the feature request quality solely based on that.

Construct validity concerns errors caused by the way we collect data. A possible issue is that the criteria list that we deduced for open source feature requests does not give a good representation of the quality of those feature requests. This would influence the observed quality of open source feature requests in Table 3. While we acknowledge that we still need to further investigate our criteria list, we also want to stress that the industry participants acknowledge that the criteria are relevant.

External validity threats concern the generalizability of our results. In the interview that we describe in Sect. 6, we only have 8 participants from industry. As such, a possible threat to validity is that other participants might have a different opinion on the usefulness of our framework. However, the participants are from sufficiently different companies and backgrounds to get a first overall impression of community feedback.

Similarly, for the cases study that we describe in Sect. 7, we have 86 final-year software engineering students participating. A possible threat here is that all students have a similar background (they all study at Fontys University of Applied Sciences) and that another group of students with another background might react differently. However, we plan for replication studies in future work.

We can also not be sure that we would have had the same results if we would have involved practitioners from industry or open source. To mitigate, this we used last-year students, who are close to becoming practitioners themselves. We also found several papers indicating that students can be good proxies for practitioners, see, e.g., [30].

We have investigated three open source projects in the case study. We cannot be sure that the results presented in Sect. 7 can be generalized to more open source projects. In order to try to mitigate this threat, we have selected the three projects to be sufficiently different in size and domain.

In this paper, we focus on *open source* feature requests because of their public availability. This means we cannot be sure that feature requests that are created as part of a closed source project adhere to the same standards. However, according to Alspaugh and Scacchi [2] "closed source software bug reports and feature requests and the process for managing them look much like those for Open Source Software," so we expect our results to hold in both cases.

9 Related work

For JIT (mostly from the area of agile development processes) and open source projects, there is a body of work both on requirements engineering [11, 26, 27] and quality assurance [1, 17]. In this section, we discuss the few papers that specifically mention quality criteria for JIT or open source requirements.

Duncan [7] analyzes the quality attributes for requirements in extreme programming (XP, one of the agile methods). He does this by comparing *stories* (the main requirements artifact in XP) to the quality attributes presented by Davis et al. [4]. This is the same approach that we followed in our Discussion, see Sect. 8.2.

Dietze [5] describes the agile requirements definition processes performed in Open Source Software (OSS) development. Quality aspects are not part of his analysis. However, he does mention meta-data of a change request corresponding to our [QC1].

Scacchi [31] argues that requirements artifacts in Open Source Software development might be assessed in terms of virtues like (1) encouragement of community building; (2) freedom of expression; (3) readability and ease of navigation; (4) and implicit versus explicit structures for organizing, storing and sharing. Virtue (3) and (4) above are covered in our framework, whereas virtue (1) and (2) should be achieved by a correct setup and management of the open source project.

Bettenburg et al. [3] conducted a survey in open source projects on what makes a good bug report, revealing a mismatch between what developers need and what users supply. Most developers consider steps to reproduce, stack traces and test cases as helpful, which are at the same time difficult to provide for users. These three items are somewhat similar to the scenario's and screens we have included in [QC1.3].

Génova et al. [10] describe a framework to measure the quality of textual requirements. They have defined metrics and implemented those metrics in a tool to automatically verify the quality. A requirement is scored as bad, medium or good. The tool is commercially available and users report benefiting from using it. They use formal requirements documents as input data, making some of their quality criteria (such as completeness—covering all user needs—and traceability—explicit relationship with, e.g., design documents) less relevant for JIT requirements. See also our analysis of the earlier work of Davis et al. [4] in Sect. 8.2. We do, however, value the idea of automating certain quality checks as was also requested by one of the participants. This is something we plan to do in the future.

10 Conclusion

Summary In this paper, we have developed a framework for the quality assessment of JIT requirements that caters for tailoring to different situations. We have instantiated this framework for feature requests (in open source projects) and indicated how to do this for other situations. The framework is based on the literature on the quality of traditional requirements as well as the literature on JIT requirements and open source projects. The framework structures quality criteria and includes a time dimension: some quality criteria should hold at creation time (*C) and others should hold just-in-time (*J) for implementation of the JIT requirement.

We have performed an evaluation with practitioners. The framework was positively evaluated by all of them. They also confirm our assumption that quality assessment of JIT requirements is important, the same as for traditional up-front requirements. We have also quantitatively evaluated the framework with 86 software engineering students. This resulted in even more confidence in the usability, completeness and relevance of the framework.

Next to that, we were able to present quantitative results for the feature request quality in three open source projects:

- A lot of feature requests score "LOW" on overall quality because of missing keywords, missing rationale or missing link to source code.
- The percentage of relevant comments is surprisingly low: around 60 %.
- The average "correctness" score is quite high: 77 %

- The correctness of the feature requests slightly deteriorates for all three projects from the moment it is created until it is closed (from 82 to 77 % correctness).

Our analysis of these quantitative results led to a top 5 of recommendations for practitioners to improve feature request quality.

Key Contributions This work makes the following key contributions:

1. a framework for quality criteria for JIT requirements;
2. instructions for practitioners how to customize the quality criteria to their specific JIT environment;
3. an instantiation of the framework for feature requests in open source projects;
4. findings on feature request quality from three open source projects;
5. recommendations for practitioners on improving feature request quality.

In this paper, we have demonstrated the value of our framework in the following three ways: (1) we have used it to give an overview of quality criteria that are applicable to feature requests (at creation time or just-in-time); (2) we have indicated how it can serve as a basis for teams that need to assess the quality of their JIT requirements; and (3) we have used it to get an insight into the quality of feature requests in open source projects.

Future Work In future work, we will extend our analysis of open source feature requests to confirm the findings from the case study with the student participants and to get a more detailed view on feature request quality. Another useful addition for future work is the automation of some of the checks to increase the usability of the checklist. We also plan to validate our framework in case studies in closed source and/or agile JIT environments.

In this paper, we have conducted interviews and a case study. To answer future research questions, we would like to gather more data suitable for statistical analysis by using controlled experiments. Our idea is to investigate related topics that require more quantitative evidence such as the relationship between feature request quality and software product quality (design, tests, code, defects) or team productivity.

Acknowledgments This work has been sponsored by the RAAK-PRO program under grants of the EQuA-project. We thank all participants in the interview sessions: R. Wouterse (SYSQA), S. Jansen, H. Nieboer, P. Devick (Info Support), A. Grund (AGrund.informatiespecialist), S. van der Zee, A. Uittenbogaard (inspearit) and R. van Solingen (TU Delft, Prowareness). We also thank all software engineers of the Fontys Applied University that participated in the case study.

References

1. Aberdour M (2007) Achieving quality in open-source software. Softw IEEE 24(1):58–64. doi:10.1109/MS.2007.2
2. Alspaugh T, Scacchi W (2013) Ongoing software development without classical requirements. In: International requirements engineering conference (RE), pp 165–174. doi:10.1109/RE.2013.6636716
3. Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: International symposium on foundations of software engineering (FSE). ACM, pp 308–318
4. Davis A, Overmyer S, Jordan K, Caruso J, Dandashi F, Dinh A, Kincaid G, Ledeboer G, Reynolds P, Sitaram P, Ta A, Theofanos M (1993) Identifying and measuring quality in a software requirements specification. In: International software metrics symposium, pp 141–152. doi:10.1109/METRIC.1993.263792
5. Dietze S (2005) Agile requirements definition for software improvement and maintenance in open source software development. In: Proceedings of international workshop on situational requirements engineering processes, pp 176–187
6. Doran GT (1981) There's a smart way to write managements goals and objectives. Manag Rev 70(11):35–36
7. Duncan R (2001) The quality of requirements in extreme programming. CrossTalk 19:22–31
8. Ernst NA, Borgida A, Jureta IJ, Mylopoulos J (2014) An overview of requirements evolution. In: Mens T, Serebrenik A, Cleve A (eds) Evolving software systems. Springer, Berlin, pp 3–32
9. Ernst NA, Murphy G (2012) Case studies in just-in-time requirements analysis. In: International workshop on empirical requirements engineering. IEEE, pp 25–32. doi:10.1109/EmpiRE.2012.6347678
10. Génova G, Fuentes J, Llorens J, Hurtado O, Moreno V (2013) A framework to measure and improve the quality of textual requirements. Requir Eng 18(1):25–41. doi:10.1007/s00766-011-0134-z
11. Grau R, Lauenroth K, Bereza B, van Veenendaal E, van der Zee S (2014) Requirements engineering and agile development-collaborative, just enough, just in time, sustainable. IREB
12. Heck P (2014) Jit requirements quality framework (Figshare). doi:10.6084/m9.figshare.938214
13. Heck P, Klabbers M, van Eekelen MCJD (2010) A software product certification model. Softw Qual J 18(1):37–55
14. Heck P, Zaidman A (2013) An analysis of requirements evolution in open source projects: Recommendations for issue trackers. In: International workshop on principles of software evolution (IWPSE). ACM, pp 43–52
15. Heck P, Zaidman A (2014) Horizontal traceability for just-in-time requirements: the case for open source feature requests. J Softw Evol Process 26(12):1280–1296. doi:10.1002/smr.1678
16. Herzig K, Just S, Zeller A (2012) It's not a bug, it's a feature: how misclassification impacts bug prediction. Technical report, Universitaet des Saarlandes, Saarbruecken, Germany
17. Huo M, Verner J, Zhu L, Babar M (2004) Software quality and agile methods. In: International computer software and applications conference, vol 1, pp 520–525. doi:10.1109/CMPSAC.2004.1342889
18. IEEE (1990) IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990 pp 1–84. doi:10.1109/IEEESTD.1990.101064
19. IEEE (1998) IEEE recommended practice for software requirements specifications. IEEE Std 830-1998
20. IIBA (2009) A guide to the business analysis body of knowledge (BABOK Guide). International Institute of Business Analysis (IIBA), Whitby

21. Kamata M, Tamai T (2007) How does requirements quality relate to project success or failure? In: International requirements engineering conference (RE), pp 69–78. doi:10.1109/RE.2007.31
22. Knauss E, El Boustani C (2008) Assessing the quality of software requirements specifications. In: International requirements engineering conference (RE), pp 341–342. doi:10.1109/RE.2008.29
23. Koch S (2004) Agile principles and open source software development: a theoretical and empirical discussion. In: Eckstein J, Baumeister H (eds) Extreme programming and agile processes in software engineering. Springer, Berlin, pp 85–93
24. Leffingwell D (2011) Agile software requirements: lean requirements practices for teams, programs, and the enterprise, 1st edn. Addison-Wesley Professional, Boston
25. Meade AW, Craig SB (2012) Identifying careless responses in survey data. Psychol Methods 17(3):437–455
26. Noll J, Liu WM (2010) Requirements elicitation in open source software development: a case study. In: Proceedings of international workshop on emerging trends in FLOSS software research and development. ACM, pp 35–40
27. Paetsch F, Eberlein A, Maurer F (2003) Requirements engineering and agile software development. In: International workshop on enabling technologies: infrastructure for collaborative enterprises. IEEE, pp 308–308
28. Philippo EJ, Heijstek W, Kruiswijk B, Chaudron MR, Berry DM (2013) Requirement ambiguity not as important as expected—results of an empirical evaluation. In: Doerr J, Opdahl AL (eds) Requirements engineering: foundation for software quality. Springer, Berlin, pp 65–79
29. Power K (2014) Definition of ready: an experience report from teams at cisco. In: Cantone G, Marchesi M (eds) Agile processes in software engineering and extreme programming. Lecture Notes in Business information processing, vol 179. Springer, Berlin, pp 312–319. doi:10.1007/978-3-319-06862-6_25
30. Salman I, Misirli AT, Juristo N (2015) Are students representatives of professionals in software engineering experiments? In: Proceedings of the 37th international conference on software engineering-volume 1, ICSE '15. IEEE Press, Piscataway, pp 666–676. http://dl.acm.org/citation.cfm?id=2818754.2818836
31. Scacchi W (2009) Understanding requirements for open source software. In: Lyytinen K, Loucopoulos P, Mylopoulos J, Robinson B (eds) Design requirements engineering: a ten-year perspective. Lecture notes in business information processing, vol 14. Springer, Berlin, pp 467–494. doi:10.1007/978-3-540-92966-6_27
32. Wake B (2003) INVEST in good stories, and SMART tasks. http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/. Accessed Nov 2013
33. Warsta J, Abrahamsson P (2003) Is open source software development essentially an agile method? In: Workshop on open source software engineering, pp 143–147