Subconscious Requirements: The Fright of Every Tester

Hans van Loenhoud

Abstract Subconscious requirements are requirements that stakeholders do not explicitly ask for, but still expect to be present in a system. Such requirements are problematic to testers as they will easily be overlooked and not be included in the specifications and therefore may not be tested at all. Over time, less notable (non-) functional requirements tend to end up as subconscious. The development of ever more complex IT solutions adds to the occurrence of subconscious requirements. They do need to be tested, as ignoring them will lead to expensive rework or rejection. Due to the lack of specifications, tests can only rely on experience-based techniques. To acquire the necessary experience, testers can turn to proven techniques from the requirements engineering discipline.

Keywords Software testing · Software quality · Subconscious requirements · Software tester

1 Introduction

As testers, we all have the same experience once in a while: a few days after the go-live of a new release, a user comes up with a serious bug. Usually, it is not about the core functionality, because everything works fine there, but more likely an issue on non-functionals like usability, security, or performance. That may seem less important, but in fact such issues can be quite nasty, difficult and expensive to solve. And then, the project manager comes to *your* desk and asks: “Why didn’t you find this bug in your tests? You spent more than xxx hours on testing and still the #!@! thing has bugs in it!!!”

If you are a beginner in testing, you turn red in the face, try to hide under your desk and consider looking for another job. If you are an experienced tester, you remain calm, look the project manager right into her eyes and say: “Nobody told

H. van Loenhoud
Taraxacum, Kockengen, Utrecht, The Netherlands

me that this was a requirement, so how could I know that it needed testing? Go to the BA and ask why this has not been specified in the first place.” So, off goes the project manager, but still you feel uncomfortable about the situation.

And although you did not know about this requirement, you still consider the absence of defects in the system to be your responsibility, no matter that you learned from the “seven testing principles” that software never will be 100% defect free (see [1]).

What actually happened was that you were struck by a subconscious requirement: a requirement that everybody, at least at the user’s side, knows about, but that is considered to be so self-evident that no one takes care of telling, capturing, or documenting it. In the remainder of this chapter, we will look into a bit of theory about these subconscious requirements and see if we can find ways to deal with them.

2 What Are Subconscious Requirements?

The concept of subconscious requirements stems from the work of Noriaki Kano, professor at the Tokyo University of Science. In the early 1980s Kano laid the foundation for a new approach to modeling customer satisfaction and developed a customer satisfaction model (now known as the Kano model) that distinguishes between essential and differentiating attributes related to concepts of customer quality [2].

Although originating from marketing research, the Kano model has become one of the fundamentals of business analysis and requirements engineering in IT, see for instance [3, 4]. The Kano model (see Fig. 1) discerns three categories of factors that are relevant for customer satisfaction:

- Performance factors

Performance factors relate to features that the customer explicitly asks for. They have a linear relationship with customer satisfaction: the more these factors are present in a product or service, the higher the satisfaction. Kano called this “one-dimensional quality,” and in requirements engineering, they are usually referred to as “satisfiers” or “conscious requirements.”

- Basic factors

These are factors that customers implicitly expect to be present in a product or service. This is what requirement engineering calls “subconscious requirements,” because customers consider these features self-evident or even are unaware of them. They are also called “dissatisfiers”: when such features are present, no one will notice them, so they don’t contribute to the customer’s satisfaction. However, when they are missing, the customer will consider the product or service to be unusable and will be very dissatisfied. Kano used the term “must-have quality.”

- WOW-factors

The third category concerns features, that the customers do not consider to be possible, so they will never ask for them. Therefore, they are called “unconscious requirements.” If they are absent in a product or service, the customer will

Fig. 1 The Kano model

not notice it, so it does not affect satisfaction. On the other hand, when they are present, the customer will be pleasantly surprised: “delighters,” or in the terminology of Kano, “attractive quality.”

One interesting observation by Kano was that requirements tend to change over time. New features start, almost by definition, as unconscious requirements. As customers discover, experience and like a new feature, it becomes a conscious requirement that is explicitly asked for. Gradually, as all similar and competitive systems implement the same feature, customers forget that originally systems did not have such a feature and start to take it for granted, turning it into a subconscious requirement. That is why many systems contain features that users consider as indispensable without knowing why and thus without explicitly asking for them.

A good example is the camera function on mobile phones, where this process took less than 20 years. The first time a camera was introduced as part of a phone, most customers were puzzled: no one had asked for this unconscious feature. But they liked it as a delighter and all brands started to implement it in their phones, turning it into a conscious requirement. Nowadays, when buying a new

phone, everybody takes for granted that it will have a camera, so it has become a subconscious requirement.

Conscious requirements are the easiest category to deal with for both the analyst and the tester. They are explicitly mentioned by the stakeholders, so they will clearly be present in the specifications of the system under test (assuming that the business analysts and designers did a good job).

Unconscious requirements are also relatively simple to test. Stakeholders did not ask for them, so the designers have incorporated them deliberately in the system and will have added them prominently to the specifications. The only problem for the tester is that these requirements are based on assumptions of the designers about the system's attractiveness for the customer. These assumptions may be wrong, so they should be tested too. Unfortunately, designers often forget to specify their assumptions, so as a tester, you should ask for them. However, even if an unconscious requirement remains untested and subsequently fails in production, the users will not complain because they did not ask for the feature in the first place (but be aware that such failure may involve extra risk that actually does need testing).

The challenge for the tester lies in the subconscious requirements. The stakeholders do not ask for them, so they are quite likely to be missing in the specifications, provided by analysts and designers as input for the tester. But they still need to be tested, because failure of a system to meet the subconscious requirements will almost certainly lead to rejection. Therefore, the tester faces the task of testing features that are not present in the specifications.

In practice, this usually does not concern the main functionality of a system, being described in detail in the specifications and thus belonging to the conscious requirements. Often, subconscious requirements relate to some "nitty gritty" functionality details and exceptions, to user-related quality criteria like usability, security, or performance, or to implicit technical, infrastructural, organizational, and legal constraints. Of course, depending on the domain knowledge of the analysts and designers, a certain part of all subconscious requirements will still be present in the specifications of a system. But on average, one can be quite sure that a significant portion is missing. If a certain requirement is missing from the specifications, the chance that it is implemented in the system is minimal.

3 How to Deal with Subconscious Requirements?

The problem for the tester lies in the (mostly subconscious) requirements that are missing from the specifications of the system under test. In such a case, common black-box testing techniques cannot be used, as they depend on the analysis of the specifications in the test basis.

White-box techniques, depending on the documented structure of the system are equally inapplicable, as the structure of a system normally is derived from its specifications.

The only thing that is left is to apply experience-based techniques: error guessing, exploratory testing and checklist-based testing (see [1]). In experience-based testing, tests are derived from a test basis consisting of the knowledge and experience of the testers themselves. Therefore, the big question is: how can a tester acquire the necessary, relevant knowledge and experience?

The only answer to this question is: as a tester, apply requirements engineering techniques that are suitable to uncover subconscious requirements. Such techniques can be found in the IREB body of knowledge (see [3, 5]):

Observation Techniques

- Field observation

This technique is about observing the users during their work in their usual environment without interfering. The tester may notice certain unexpected or undescribed behavior, strange sequences in activities, or manual sidesteps. These are strong indicators for subconscious requirements.

- Apprenticing

Apprenticing goes a step beyond field observation. Here, the tester conducts a short hands-on training in the environment in which the system is to be deployed. Key users teach him their work processes to better understand the domain. When actually participating in the work to be done, subconscious requirements will easily come to the surface.

- Contextual inquiry

Contextual inquiry is an iterative, field data-gathering technique, studying a few carefully selected users in depth to arrive at a fuller understanding of the work practice across the entire user base.

Artefact-Based Techniques

- System archaeology

System archaeology is a technique to gather information regarding a new system from the documentation, user interface or code of a legacy or competitor system. Of course, most of the requirements found by this technique will be present in the specifications of the system under test. The remainder will either not be relevant in this particular case or turn out to be subconscious.

- Perspective-based reading

In this technique, the tester uses a specific perspective—in this case looking for unfamiliar requirements—in order to retrieve information from documents that are relevant for the domain in which the system will be used, e.g., process descriptions, company regulations, and applicable legislation.

These techniques have one major drawback: they are usually quite time consuming. However, this time investment is inevitable to acquire the thorough domain knowledge necessary to recognize subconscious requirements and to design tests to mitigate related risks.

A quite different approach to find and test unconscious requirements is Specification by Example (see [6]). This works best when applied by the whole development team throughout the development process, as it promotes shared understanding and trust within the team. But even for testing itself, it can be very useful.

In this approach, test cases are not derived from specifications, but from an iteratively growing set of real-life examples of inputs and outputs from the work processes that the system is intended to support. Specification by Example will yield tests for all requirements, conscious, unconscious, and subconscious alike, and may, in the long run, be more time-efficient than the techniques mentioned earlier. An additional benefit of Specification by Example and similar approaches like Behavior-Driven Development (see [7]) is that the resulting set of examples/test cases ultimately forms a complete and up-to-date summary of the system and its behavior, the so-called “living documentation.”

Yet another approach that deserves mentioning here is Design Thinking (e.g., see [8]). Several variants of Design Thinking exist, but they all focus on understanding the true needs of the users by introducing human-centered techniques like Persona’s and Empathy Mapping. Especially the exploration of the “pains” and “gains” of different user groups can help the tester identify previously undetected subconscious requirements.

A common technique from most Design Thinking variants is prototyping. Prototyping offers future users the opportunity to gain hands-on experience with early versions of a new system and to provide feedback on its behavior, thus easily uncovering subconscious requirements. Especially the so-called low-fidelity prototypes like UI-sketches, storyboards, and customer journeys are very useful in this respect. Like Specification by Example, Design Thinking is in fact a whole-team effort, involving analysts, designers, developers, and testers.

4 What About the Future?

One might hope that by the growing professionalism and maturity of IT analysis and design, the tester’s problems with subconscious requirements will gradually disappear, as nowadays most product owners, business analysts, and requirements engineers learn about techniques to elicit and communicate them.

However, trends in IT work in the opposite way. To illustrate these trends, the Cynefin Framework (see Fig. 2) is very useful.

The Cynefin framework [9] is a conceptual framework used to aid decision-making as a “sense-making device.” It was created in 1999 by Dave Snowden working for IBM. Cynefin draws on research into systems theory, complexity theory, network theory, and learning theories and offers five decision-making contexts or “domains”:

- Obvious
- Complicated
- Complex
- Chaotic
- Disordered

Fig. 2 The Cynefin framework

Although initially intended for decision-making, it can also be used to categorize IT projects with respect to the level of uncertainty that they must deal with (ignoring “disordered,” a temporary state that changes into another after clarification):

- Obvious projects develop IT systems that offer support for clear, straightforward business processes, like order processing or accounting. Everything relevant to this development is known in advance (“known knowns”), so even subconscious requirements will be documented in the specifications. In these projects, developers and testers can easily rely on best practices.
- Complicated projects are, as the name implies, a bit more complicated. They often relate to the integration of different business processes, for instance, the development of ERP systems. There, developers and testers may be confronted with unexpected and subconscious requirements that are not present in the specifications. However, the type of requirements itself is known (“known unknowns”), so with a sufficient amount of domain knowledge testers will be able to deal with them by choosing from existing good practices.
- Complex projects are projects that develop IT systems for innovation, with well-known examples like Spotify and Uber. Here, developers and testers face “unknown unknowns.” Cause and effect can only be deduced in retrospect, and there are no right answers upfront. Development teams will have to develop their own emergent practices to deal with the inherent uncertainties. Design Thinking offers clues to find the inevitably missing subconscious requirements.

- The most “problematic” category is that of the chaotic projects: not chaotic in the sense of bad project management, but chaotic in relation to the topic that has to be developed. This is about the “unknowables.” The team explores completely new fields of IT, like the Internet of Things, Artificial Intelligence or machine learning. In these kinds of projects, not only the exact features, but even the future users are uncertain. But anyhow, these systems will interact with users, so the chance of meeting subconscious requirements after implementation is 100%. The only thing that can help you there may be a quote from Steve Jobs [10]: “Have the courage to follow your heart and intuition.” As a tester, you will have to detect your own subconscious requirements and make them conscious in your test cases.

The Cynefin framework can also be read as a timeline. In the early decades of IT, most projects were in the obvious quadrant, automating “islands” of single, strictly demarcated business processes. Somewhere from the 1990s onward, we started trying to integrate these islands, and projects became complicated. With the breakthrough of internet and mobile apps, we saw the arrival of complex projects that created and triggered completely new business processes.

As IT extends its scope beyond the original domain of computers, invading products like cars, refrigerators, light bulbs, or weapons, the part of chaotic projects is growing rapidly. So as a tester, be prepared to deal with the unknowables; you will find more and more subconscious requirements on your path. Consider, for instance, the requirements for a “friend-or-foe” system in an autonomous combat robot, let alone the challenge of testing such a system . . .

5 Conclusion

Subconscious requirements are requirements for a feature that users in a certain domain take for granted or only become aware of when it is not implemented. Such requirements are easily overlooked, even by experienced analysts and designers, so there is a good chance that some of them will be missing from the specifications for a system.

Testers still have the responsibility to test the relevant subconscious requirements, but fear overlooking some of them, because they cannot rely on the specifications. The absence or failure of such a feature will often seriously affect the system, leading to expensive repairs or even rejection. To test subconscious requirements, testers cannot rely on their commonly used specification- and structure-based testing techniques, so they switch to experience-based techniques.

This raises the question of how testers should acquire the necessary domain knowledge and experience. In the first place, testers should (be able to) apply observation and artefact-based techniques from the requirements engineering discipline. Such techniques are especially suitable for finding subconscious requirements. Learning and applying these techniques is very relevant but will significantly add to the workload of the tester.

Agile approaches like Specification by Example, Behavior-Driven Development, and Design Thinking also significantly reduce the risk of overlooking subconscious requirements, as they heavily rely on real-life examples, customer empathy, and early feedback. These approaches work best when applied by the whole team throughout the entire development life cycle, so they are not suitable for testing in isolation.

Subconscious requirements are here to stay. The Kano model shows us that many requirements in the long run will end up as subconscious ones. From the Cynefin framework we learn that in the future, even more development will take place in the complex and chaotic domains, where the “unknown unknowns” and “unknowables” predict many subconscious requirements.

The time has gone for testers to derive their tests from a solid and extensive test basis. The future of testing will be predominantly experience based!

References

1. Olson, K., et al.: Certified tester foundation level syllabus version 2018. International Software Testing Qualifications Board. <https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html> (2018)
2. Kano, N., et al.: Attractive quality and must-be quality. *J. Jpn. Soc. Qual. Control* (in Japanese). **14**(2), 39–48 (1984)
3. Stapleton, P.: Agile Extension to the BABOK® Guide V2. International Institute of Business Analysis and Agile Alliance, Toronto, ON (2017)
4. Frühauf, K., et al.: IREB Certified Professional for Requirements Engineering - Foundation Level – Syllabus Version 2.2.2. https://www.ireb.org/content/downloads/2-syllabus-foundation-level/ireb_cpre_syllabus_fl_en_v222.pdf (2017)
5. Häusser, D., et al.: IREB Certified Professional for Requirements Engineering – Advanced Level Elicitation – Syllabus Version 2.0.1. https://www.ireb.org/content/downloads/7-syllabus-advanced-level-requirements-elicitation/cpre_elicitation_al_syllabus_en_2.0.1.pdf (2019)
6. Adzic, G.: Specification by example – How successful teams deliver the right software. Manning Publications, Shelter Island, NY (2011)
7. North, D.: Introducing BDD. <https://dannorth.net/introducing-bdd/> (2006)
8. Alliance for Qualification: Design thinking foundation syllabus. https://isqi.org/nl/en/index.php?controller=attachment&id_attachment=27 (2018)
9. Snowden, D.J., Boone, M.E.: A leader’s framework for decision making. *Harv. Bus. Rev.* **85**, 69–76 (2007)
10. Jobs, S.: You’ve got to find what you love. <https://news.stanford.edu/2005/06/14/jobs-061505/> (2005)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.