Chapter

Software Design for Success

Laura M. Castro

Abstract

Technical books focus most of the times in technical stuff, as one should expect. However, this creates the illusion that technology is somewhat free of bias, always neutral, thus fitting everyone. Reality, later on, when the product is already there, proofs us otherwise. Inclusion and representation are crucial from the design and modeling stages. Visibility of minorities or underrepresented groups is on the rise, yet so much of the way is left for us technicians to walk. In this chapter, we will analyze, from an architectural point of view, which non-functional requirements are most sensible to this and how to start the conversation about them to maximize the possibilities for success of our software products.

Keywords: inclusion, visibility, representation, software architecture, non-functional requirements

1. Introduction

Software is omnipresent. From personal computers and laptops, it has extended its presence to tablets, smartphones, smart watches, and wristbands. From software packages delivered on CDs, it has moved to apps and services which run uninterruptedly on remote servers. From our professional workplace, it has conquered our personal lives, relationships, and leisure activities.

We could see this as proof of the success of the software industry, the technology revolution. But is it? What does success represent exactly, in societal terms? Are we solving people's problems, or rather are we creating new ones? In order to make this argument more objective, we need to define success. We can argue that success of the software industry is proved by its constant innovation. But innovation is not necessarily equal to progress, which should be the key indicator in terms of societal benefit.

According to Wikipedia, "progress is the movement towards a refined, improved, or otherwise desired state (...), the idea that advancements in technology, science, and social organization can result in an improved human condition" [1]. Arguably, we seem to be constantly producing advancements in technology, but it has also become evident that the technology advancements we are producing are not improving the condition of all humans equally. Mass media is regularly hit by news where "algorithms" are revealed as biased, showing behaviors which are sexist, racist, LGBTI-phobic, etc. Software leaves minorities out, and apps discriminate on bases of age or socioeconomic status [2]. Be it in recruiting [3], evaluating risk for a financial product, assigning probability of crime involvement [4, 5], targeting adds [3], or classifying our pictures [6], the direct consequence of these errors is blatant failure.

Of course, there are many angles to this systemic problem. In this chapter, we will analyze how we can contribute to progress, ensuring the success of our software product, from the perspective of software architecture.

Software architecture is the part of software development which defines the high-level decomposition of a system into a set of functional components and describes its responsibilities and interactions. A software architect is thus responsible for selecting those components, defining said interactions, and describing the constraints that operate over both of them. These decisions are grounded on both functional and non-functional requirements of the software and will serve as basis for the design, implementation, and testing stages (no matter which development cycle is used).

Consequently, one of the most important skills of a software architect is asking the relevant questions which answers can make the difference between product success and failure. The said questions need to provide confidence in that both functional and non-functional requirements are correctly elicited, understood, and quantified, as a mandatory previous step to allow their correct development and validation. In particular, failure to properly define non-functional requirements (also referred to as "system requirements") is the third cause of software project failure [7].

In the remainder of this chapter we will go over the definition of non-functional requirement and provide a taxonomy for practical use. We will identify the non-functional requirements that are more closely related to software success in terms of societal progress. We will discuss them and provide insights on how to extract and test them.

The aim we pursue by doing this is to hand in a handbook of rules, an enhanced checklist, that would be useful for practitioners and future professionals that want to specialize in the area of software architecture. We trust the contents that follow will spark their interest in and concern about really successful software, as well as be a useful guide in building it. However, by keeping our technical level purposely abstract, we aim to make this chapter readable for the general public as well, a general public who, as massive consumer of software products, can also benefit from awareness about what kind of successful products they can and should be demanding from the software industry.

2. Software architecture: it's all about non-functional requirements

Software requirements, as defined by the IEEE [8], are "a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document." As we can see, no explicit mention is made as of the *character* of the said condition or capability. Traditionally, the software industry showed a tendency to focus on functional requirements, which are defined as combinations of behaviors between inputs and outputs [9]. Requirement elicitation and other software development and software life cycle management practices provided support for functional requirements in terms of formalisms like use cases, user stories, etc. The focus was on *what* software had to do, rather on *how* it was supposed to do it.

It is the definition of software architecture as a critical part of the software development process [10] which brings attention to non-functional requirements. Non-functional requirements (sometimes referred to as "quality requirements") are defined as criteria to be used to judge the operation of the system, rather than specific behaviors. It is this system-wide relevance that makes most non-functional requirements *architecturally significant* [11], since they impose constraints on the

DOI: http://dx.doi.org/10.5772/intechopen.86538

design or implementation. **Figure 1** shows a common taxonomy of non-functional requirements.

However, we are still failing to systematically build a software that is a success, at least in the terms we were discussing in the previous section. Both what things our systems do and how they do them reveal those omissions, biases, and plain discrimination we would very much like to eradicate.

From the software architecture perspective, this is because **Figure 1** shows a very narrow view of non-functional requirements. Compare this with **Figure 2**, which presents a more exhaustive relation of parameters of interest for any software application.

In the following subsections, we will traverse the taxonomy in **Figure 2** that extends that of **Figure 1** beyond the shadowed area, to provide insights on what might be missing from our products if we overlook them.

2.1 Revisiting product requirements

The fact that product-focused non-functional requirements have received more attention than the rest of the extended taxonomy of non-functional requirements shown in **Figure 2** does not mean they cannot and should not be revisited under the "societal success" mindset. We now see how.

Usability. In software engineering, usability is meant to quantify the quality of user experience and as such is typically described in terms of effectiveness (i.e., number of goals that can be achieved using the software) and efficiency (i.e., time required to complete said goals), alongside with other less objective parameters such as

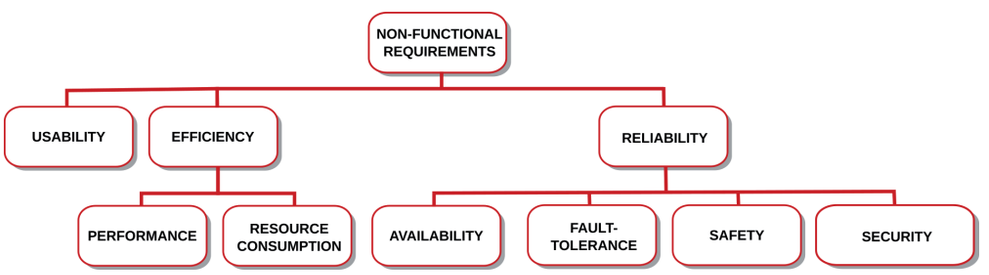

Figure 1.
Traditional taxonomy of non-functional requirements.

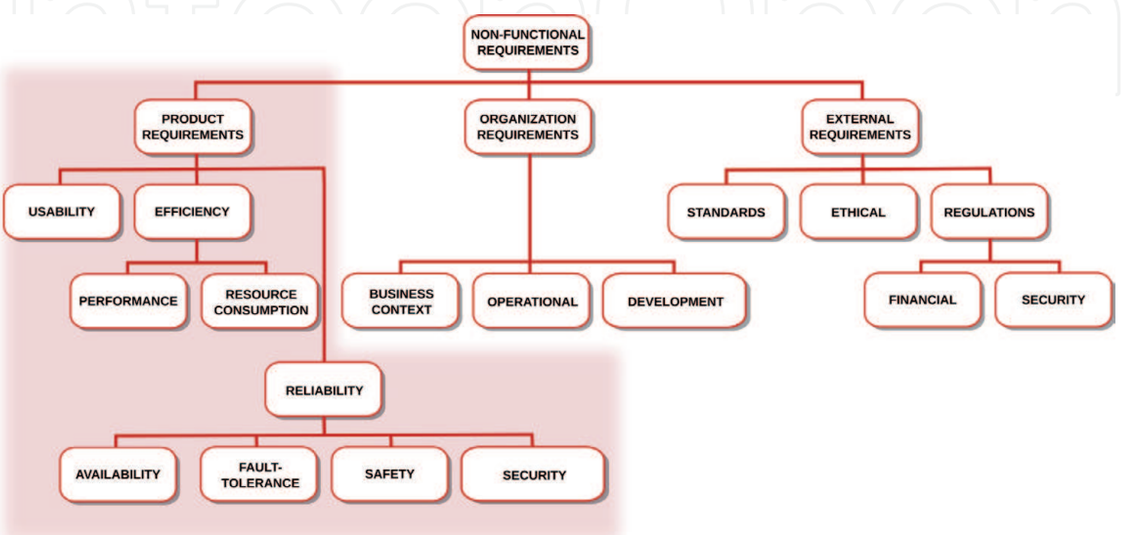

Figure 2.
Extended taxonomy of non-functional requirements.

satisfaction or perceived elegance. Usability should, however, be measured for different types of users, instead of a "regular user" or "normal user," which is often the case. The recommendation here is to look beyond our idea of typical user and consider the widest user base possible, featuring users with different levels of acquaintance with technology, from different age ranges, and health conditions (including transitory states such as pregnancy) and identifying with different gender options, with different socioeconomic backgrounds, and different sensor, motor, and mental skills.

Performance. Although different meanings can be associated with the term, the most common one is that of software responsiveness and is usually quantified in terms of requests or actions per unit of time. There are different and well-known approaches to controlling performance: from resource demand management (i.e., adjusting the ratios at which a component or subsystem generates events or polls or the context for information, explicitly limiting execution time, etc.) and resource arbitration (i.e., fixed or dynamic priorities) to its effective management (i.e., exploiting concurrency, duplicating data, or processes). Performance should, however, also take into account the consequences of these well-known strategies beyond the improvement or preservation of certain response times. When such response times have a human user at the other end, all the same we ought to take into account their expectations and perceptions of responsiveness. In this case, these can vary widely, providing further chances for improvement by alleviating the performance demands in some cases, which can be advantageously used to tend to other user profiles. Additionally, the consequences of performance degradation should also be contemplated under the light of the people that would suffer them, since depending on the kind of service we are providing and to whom, it might be more or less critical to comply.

Resource consumption. Similar to performance, resource consumption (be it computing power, volatile or nonvolatile storage, network access, bandwidth, etc.) has two sides to it. Whenever our system or software product needs to preserve resources and we aim to optimize them, unexpected consequences on the widest possible use base should be scrutinized. It is more likely that the problem is the reverse, since it is when we increase the demand for resources (or simply do not limit them) that we are more likely to implicitly exclude sectors of the population which might not have access to them. But in the context of excess of energy consumption that surrounds us, to which the prevalence of technology is no foreign actor (rather the contrary), every system and software should be as energy-aware as possible. Treating resources as unlimited is never a good idea, and it is not socially responsible either.

Availability. The most commonly understood definition of availability in the context of software engineering is the proportion of time a system or application is in a functioning condition, that is to say, capable of providing to its users the answer or services it is meant to, within acceptable conditions (i.e., usability, performance). The definition of this non-functional requirement makes sense considering that error-free software is virtually nonexistent. When we assume there will be errors, we need to define to which extent the presence of this errors can or will affect the normal operation of the system. For some systems, it is okay to be down for a few minutes, hours, or even days (if it is, e.g., software used in a factory which is down during the weekend). How flexible we can be about availability with regard to our software product depends on both the expectations of the users about it and the consequences of violating those expectations. In some cases, not being available might mean the users will turn to use our competitors' product instead (with the risk of not coming back); in others, it might affect their lives, threaten their security, or putting them on harm's way. Once more, we need to consider the broad population when leveraging said expectations and not only our "normal user."

Fault tolerance. Closely related to availability, fault tolerance is the capability of operating properly in the event of (internal) failure(s). The term usually helps us to stress that availability is not a black vs. white kind of situation, since the operating quality (i.e., performance, resource consumption) of a system might decrease proportionally to the severity of the failure(s). When we design for success with fault tolerance in mind, we aim to avoid that no undiscovered error in the software should be able to cause a total breakdown. With regard to societal success, the same considerations as with general availability apply.

Safety. Of course we never mean for our software to pose a risk to its users nor actively nor as a consequence of malfunctioning or unavailability, but actively considering this possibility during the whole development process involves considering safety as one of its requirements. Formally, however, safety includes not only not harming (no matter how severely) people but also goods and/or the environment [12, 13]. In a way, this links back to resource consumption in the energy-awareness aspect that we mentioned before. We could even argue whether introducing new technology where it does not bring societal progress, just for the sake of it, is not *safe*, since the environmental impact of the volume of technology we consume is already too high [14, 15]. Better approaches would always involve reusing or repurposing already existing technology, which is also less likely to exclude less privileged groups of population.

Security. Admittedly one of the major challenges in software engineering nowadays, we can informally define security as the resistance of a system or application to unauthorized uses, while operation is still granted for legitimate ones. There are several aspects of this claim that might be jeopardized if the diversity of the population is not properly accounted for, the most important of which would be to wrongly classify a legitimate request for an unauthorized one [16, 17].

2.2 Refocusing organization requirements

In the previous subsection we have gone over the "classical" non-functional requirements that we have more specifically labeled as product non-functional requirements. There are two more categories of non-functional requirements to consider, one of them being those non-functional requirements derived from our organization. We discuss them next.

Business context. Whether our organization is a startup, fast-growing spin-off, an enterprise with sustained trajectory, or a business in trouble has a big influence on the goals, value, and time-to-market elements that define success in traditional terms. From this lens, a critical look is also needed, in order to detect room for improvement within. In this case, we think in terms of the people who form our teams, their profiles, and the roles they play. And, different from what we have seen in the previous section, we aim not only to reflect societal diversity, but we should strive to improve it in terms of equality and representation. Diverse teams and people from unrepresented groups in positions of decision-making and power can be our most strategic business advantage.

Operational. How and why we organize the internal functioning of our organization will have an impact in our products and their quality. Identifying the essential capabilities (or lack thereof), performance measures, and actions to be taken for improvement cannot be done without explicitly accounting for diversity and work-life balance, which in turn are key to workplace satisfaction and commitment. If so, we risk coding into the so-called "company culture" a set of barriers for employees that "are not the norm." But why, if we have agreed there is not one "normal user," should we assume there is one "normal employee"? The answer is easy: we should definitely not.

Development. Among the different product development methodologies and life cycles that have been defined in software engineering, there is arguably no silver bullet. It is more a matter of finding the most suitable match between product requirements, business needs, and operational structure. When referring to the literature on software development, this is often remarked as being the case, but then we straightaway proceed to talk about iterations, sprints, stakeholders, minimum viable products, etc. without ever relating these concepts to the composition of our development teams. Software products are made by people and same as shoes or clothing, hardly ever one size fits all. In other words, the best software development approach will be that in which all of our diverse (see operational requirements) development team can be most productive at.

2.3 Advancing external requirements

In this last section of this chapter, we turn to external factors. We do this because none of us technology makers live in the vacuum, and our actions are subjected to societal norms and laws and in turn influence how societal norms and laws evolve. This means we share a double responsibility: on the one hand, diligently complying with the former, and on the other hand, challenging the status quo when it is necessary and advancing it.

Standards. Whether they dictate norms, conventions or requirements for data format, storage or exchange, or for service provision, interfacing, or requirements, standards play a fundamental role in software interoperation, especially if they are internationally recognized and publicly available (i.e., open). As software creators, we are responsible not only for being aware of which standards affect the areas we are deploying our software on and/or the activities it provides support to. What is more, implication of software agents of all sizes, small included, in standardization processes, is very much needed in a world in which, more often than not, conventions are imposed by big players or agreed upon among few of them behind closed doors. Paradoxically, non-functional product requirements are not usually enforced for standards themselves, which have a reputation for lacking usability, for instance. When referring to executable or interactive elements, standards or standardization efforts should always be accompanied by software tests. Implementing tests for standard specifications is a way of disambiguating them and stress-testing them. And of course, whenever a standard features or refers to any aspect of what a persona is, the assumptions that may be underlying need to be contrasted against the widest definition possible.

Ethical. When we discussed safety, we mentioned that introducing new technology just for the sake of it could not be considered *safe*. Hence, nor can it be considered *ethical*, we add now. More and more it is the case that CS studies feature courses that introduce future professionals to the concepts of professional ethics in the context of technology and software. However, many universities and academic institutions still do not offer them, and it is not reasonable to assume that every professional involved in technology and software creation has or will have a university background. The lack of a universal ethics, so to speak, is likely to make this the most subjective and controversial non-functional requirement of all. However, it is undeniable that we cannot look at software as a mere tool anymore, but rather a piece of technology that embodies the ethical commitments of those who make it and those who decide it should be made and used. The ethical aspects surrounding software products have two aspects to them: (a) whether the development of the product itself is contextually right or wrong or (b) whether the development of the product significantly affects the life or balance of power of or between individuals. Focusing on the latter case, we here advocate once more for a holistic approach to

what individuals we bear in mind when analyzing the issue. As for the former, there are two main perspectives: (a) use software in particular, and technology in general, to eliminate or reduce suffering and maximize well-being and happiness for the greatest number of people and (b) use software in particular, and technology in general, to follow society's universal rules. The first, however, poses a great opportunity that the second misses: the chance to challenge the status quo and advance society's rules themselves, by pursuing the common and greater good. Paraphrasing a renowned Sci-Fi series, "the need of the many outweigh the need of the few," but the need of the few cannot be consistently overlooked: that's the way minorities are forever discriminated. Our non-functional ethic requirements need to address whether our software is *right*, but also whether it is *just* and *fair*.

Financial regulations. The prevalence of technology, when applied to software that runs as a service, means that we might be providing services to an international use base before we actually give some thoughts to the financial implications of this in terms of tax declarations, client rights, anti-monopoly legislation, etc. These are much specific issues than those of ethical non-functional requirements but still need us to explicitly identify and decide how to take care of them, from the different possibilities that we might have before us.

Security regulations. In line with the financial non-functional requirements, user-privacy and user-data preservation laws might affect our software products regardless of whether we operate beyond the scope of a single country or not. Furthermore, cybersecurity and privacy awareness is on the rise, so the context and actual rules we might need to oblige to or enforce are subjected to far greater dynamism that those of financial nature. This needs to be considered in terms of maintenance and product life.

3. Conclusion

Technology is not neutral. A biased development team, organization, societal context, etc. will most likely produce biased software. Biased technology perpetuates damaging stereotypes, hinders the empowerment of minorities and underrepresented groups, and ultimately delays innovation and progress. This can hardly be considered successful [18]. So far, the most effective ways of fighting biases in technology that we know are (a) being aware of said biases, in all their shapes and forms, and (b) striving to have as much diverse development teams and organizations. However, future possibilities may include the perspective of automated testing of fairness [19].

From the perspective of software architecture, one of the critical stages in software development, we can work toward the construction of less biased software by carefully analyzing the non-functional requirements that are relevant to our product. By first extending the traditional taxonomy of non-functional requirements (much focused on product requirements alone) and then (re)visiting it one by one, we have shed some light on how a software architect can contribute in this very important endeavor. We have provided a sort of exhaustive but high-level checklist that (a) practitioners and future professionals can use when analyzing and designing their systems and applications and (b) users can use to empower themselves in claiming that the whole software industry evolves to a higher level of responsibility toward not only innovation but progress.

Author details

Laura M. Castro
Universidade da Coruña, A Coruña, Spain

*Address all correspondence to: lcastro@udc.es

IntechOpen

References

[1] Progress [Internet]. En.wikipedia.org. 2019. Available from: https://en.wikipedia.org/wiki/Progress [Accessed: 12 April 2019]

[2] Amazon scraps secret AI recruiting tool that showed bias against women [Internet]. U.S. 2019. Available from: https://www.reuters.com/article/us-amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G [Accessed: 12 April 2019]

[3] UK government probes algorithm bias in crime, recruitment, and finance [Internet]. Uk.finance.yahoo.com. 2019. Available from: https://uk.finance.yahoo.com/news/uk-government-probes-algorithm-bias-crime-recruitment-finance-000153694.html [Accessed: 12 April 2019]

[4] Google Photos labeled black people 'gorillas' [Internet]. Eu.usatoday.com. 2019. Available from: https://eu.usatoday.com/story/tech/2015/07/01/google-apologizes-after-photos-identify-black-people-as-gorillas/29567465/ [Accessed: 12 April 2019]

[5] Simonite T. Study Suggests Google's Ad-Targeting System May Discriminate [Internet]. MIT Technology Review. 2019. Available from: https://www.technologyreview.com/s/539021/probing-the-dark-side-of-googles-ad-targeting-system/ [Accessed: 12 April 2019]

[6] 5 Apps That Have Rampant Discrimination Built In [Internet]. Cracked.com. 2019. Available from: https://www.cracked.com/blog/5-apps-that-have-rampant-discrimination-built-in/ [Accessed: 12 April 2019]

[7] Why Software Fails [Internet]. IEEE Spectrum: Technology, Engineering, and Science News. 2019. Available from: https://spectrum.ieee.org/computing/software/why-software-fails [Accessed: 12 April 2019]

[8] IEEE 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology [Internet]. Standards.ieee.org. 2019. Available from: https://standards.ieee.org/standard/610_12-1990.html [Accessed: 12 April 2019]

[9] Fulton R, Vandermolen R. Airborne Electronic Hardware Design Assurance. Oakville, Ontario (Canada): CRC Press; 2017

[10] Perry D, Wolf A. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes. 1992;**17**(4):40-52

[11] Chen L, Ali Babar M, Nuseibeh B. Characterizing architecturally significant requirements. IEEE Software. 2013;**30**(2):38-45

[12] "FAQ—Edition 2.0: E) Key concepts". IEC 61508—Functional Safety. International Electrotechnical Commission

[13] Sommerville I. Software Engineering. Boston, Massachusetts (USA): Pearson Education; 2015

[14] Horbach J, Rammer C, Rennings K. Determinants of eco-innovations by type of environmental impact—The role of regulatory push/pull, technology push and market pull. Ecological Economics. 2012;**78**:112-122

[15] Dietz T, Rosa E. Rethinking the environmental impacts of population. Affluence and Technology. Human Ecology Review. 1994;**1**(2):277-300

[16] HP Face-Tracking Webcams Don't Recognize Black People [Internet]. Gizmodo.com. 2019. Available from: https://gizmodo.com/hp-face-tracking-webcams-dont-recognize-

black-people-5431190 [Accessed: 12 April 2019]

[17] Bowles N. 'I think my blackness is interfering': Does facial recognition show racial bias? [Internet]. The Guardian. 2019. Available from: https://www.theguardian.com/technology/2016/apr/08/facial-recognition-technology-racial-bias-police [Accessed: 12 April 2019]

[18] Ralph P, Kelly P. The dimensions of software engineering success. In: Proceedings of the 36th International Conference on Software Engineering (ICSE); 2014; 2014

[19] Galhotra S, Brun Y, Meliou A. Fairness testing: testing software for discrimination. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE) 2017; 2017