

Designing for Non-Functional Requirements

Seattle University, 2007

Vuillemot, Ward

Wong, Wai

Yager, David

Abstract

Improving software quality involves reducing the quantity of defects within the final product and identifying the remaining defects as early as possible. In fact, defects found earlier in the development lifecycle cost dramatically less to repair than those found later. However, engineers cannot address non-functional quality requirements such as reliability, security, performance and usability early in the lifecycle using the same tools and processes that they use after coding and at later phases. Approaches such as stress testing for reliability, measuring performance and gauging user response to determine usability are inherently post-integration techniques. Accordingly, defects found with these tools are more disruptive and costly to fix.

The goal of this paper is to gain an understanding of where in the development lifecycle companies address non-functional requirements and what methods, if any, they are taking to address these requirements earlier. This research highlights the sporadic industry acceptance of some popular methods for designing for non-functional requirements and suggests some practical approaches that are applicable for companies that also must consider the demands of schedule and cost.

1. Introduction

Software projects are subject to numerous external pressures and constraints. As the cost of development, testing and maintenance increases, software quality improvements that reduce the amount of rework and production defects become integral to the success of projects. Traditionally, software teams address software quality requirements, sometimes called non-functional requirements (NFRs), using product-centric [1] methods. These methods are curative [2] and focus on gathering metrics and testing to examine a product after construction to determine whether it is within certain quality constraints. Dromey [2] has identified a number of deficiencies with the exclusive use of the product-centric approach:

- Testing is a non-productive part of software development and can consume 50% or more of the cost of a project. Reducing the amount of testing and related rework required can directly reduce cost.

- Reducing the number of defects or addressing them before the product is created is much cheaper than repairing them after construction. This is supported by Davis' findings [3] that indicate the relative cost of fixing a defect during each phase of a project.
- There are many well-researched methods with known benefits such as prototyping and identifying personas that can be employed to address quality requirements before software construction.

Mylopoulos, Chung and Nixon [1] proposed another approach to addressing NFRs. They call this the process-oriented approach. In this preventative approach, the goal is to prevent problems with quality from being injected into the product during the requirements or design phases. Depending on the type of requirement, this can be achieved with different tools from critical path and design reviews for performance, to managed-code architectures for security. The primary benefit of this approach is the reduction in defects found later in the project and therefore a reduction in cost to the project. Some [4] have criticized this method due to the difficulty of finding many defects without viewing the product from the user's perspective. And certainly, it is easier to measure a product's properties instead of investing effort into design approaches that achieve improvements that are difficult to quantify. Nonetheless, since the cost of finding a defect during testing can be 50 times the cost of finding it during requirements [3], the benefits of preventative quality management cannot be ignored.

The goal of this paper is to highlight the methods that the industry is currently using to address NFRs and to identify whether these methods could benefit from the process-oriented approach or whether they are already there.

We have limited our discussion to four distinct quality measures: performance, usability, reliability and security. We discuss each of these in separate sections. There are dozens of other quality measures in addition to these four, but the fruits of a preventative approach can be beneficial for all non-functional requirements. For each of these measures we discuss what our research indicates teams are doing to address the requirement and we compare this with the preventative approach proposed by researchers. Then we discuss what else teams could do to move towards a process-oriented approach, or in some cases why that is not possible.

Each section contains a graph divided into three separate measurements: Design, System and User. This division represents the phase of the project responsible for addressing the NFR. "Design" represents requirements gathering, documentation and functional specification. "System" represents integration, compilation and testing. "User" represents post-implementation error reporting and other forms of user feedback. For each of these phases we provide three measurements. The first is what a preventative method could be (Preventative

Approach). The second is what the respondent is currently doing (Current Approach), and the third is what the respondent feels is the most important approach (Desired Approach). Unfortunately, there is no recommended distribution of effort for projects wishing to follow a preventative approach. Instead we simply have the recommendation of “the earlier the better”. So to allow comparison with the curative and desired measurements, we distributed the preventative measurement into 60% for design, 30% for system and 10% for user.

1.1 Research Method

Our project consisted of a survey submitted to a software company and the IT department of an aerospace company. With half of the respondents, follow-up email interviews were conducted to collect additional information. Respondents held many roles including developers, developer managers, project managers, analysts, architects and testers.

2. Performance

2.1 The Problem

Since Connie Smith introduced the term Software Performance Engineering and pointed out the fix-it-later approach that software engineering adopted for performance requirements in her paper published in 1981, it has captured lots of attention in academia [5]. Since then, research has provided many possible methods and solutions for handling performance requirements early in the design stage.

Those researchers who focus on addressing performance requirements early in the design stages have recommended many approaches. For example, Kanchana and Sarma propose applying the Taguchi methods on the software design process to maximize the performance of the software[6]. Israr, Lau, Franks, and Woodside came up with a light-weight performance model called Software Architecture and Model Extraction (SAME) to help identify performance problems early during software design[7]. Floch, Hallsteinsen, Stav, Eliassen, Lund, and Gjorven also recommended another architecture model called MADAM (mobility- and adaptation-enabling middleware) which aims for improving the performance of mobile computing software design[8].

In contrast to the preventative approach proposed by Mylopoulos, Chung and Nixon, some scholars still stress the importance of testing and validating performance requirements after implementation. For example, Gregoriades and Sutcliffe propose a scenario-based assessment tool called System Requirements Analyzer (SRA) tool to validate the software performance requirements and identify the problem areas and performance bottlenecks in the software [9].

2.2 Our Findings

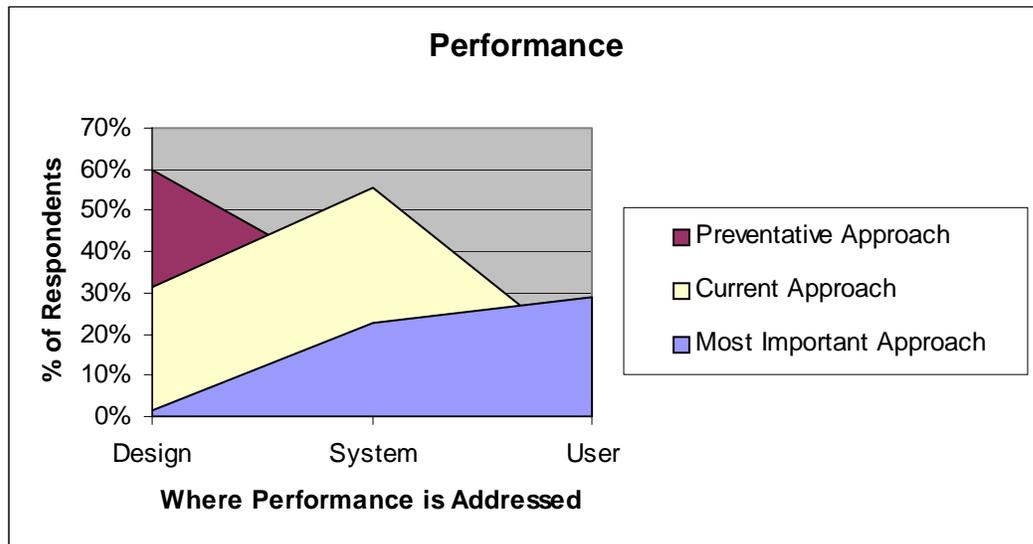


Figure 1: This figure describes where in a product's lifecycle (at design, at system test or when the user receives the product) that our respondents address performance contrasted with an arbitrary preventative approach

Our survey shows that 69% of the respondents address the software performance requirements after the requirements and design phase of their project. Only 13% of the effort for improving performance is spent during requirements gathering and 18% during design and documentation. Of all the individual areas, the most effort (24%) is spent in internal testing and measurement.

When we asked respondents how they address the performance requirements, most of them indicated that they identify scenarios and use cases that are most important to their project and then focus on improving the performance of these features. The next most selected method is measuring the overall performance of the software after implementation. The least used method was to identify performance bottlenecks during integration and focus on those.

2.3 Where to go from here

A recent research survey by Balsamo, Marco, Inverardi and Simeoni shows that even though there is no universal methodology for addressing performance requirements early in the design, the model-based software performance prediction is mature enough for the software industry to adopt [10]. However, it is clear from our survey that these performance prediction tools are not widely adopted in the industry. Our respondents still rely heavily on simply measuring the performance of their software and focusing efforts on improving the flow of the critical performance path through their system.

Our interviews indicate that project cost and schedule constraints are the primary reasons the industry favors traditional measurement methods to the newer

prediction tools. Assuming the customer finds initial performance acceptable, our respondents are much more comfortable addressing performance problems as they appear. This is especially true when competition causes the profitability of a product to be dependent on the time to market.

Our respondents exert substantial effort testing and collecting measurement of their products' performance. From these results, they examine the critical path of their product's execution to determine if opportunities for improvement exist. Unfortunately, because they perform these activities after code construction, there is no assurance that the critical path was efficient to begin with. Research also shows that designing performance requirements early in the project can greatly reduce the amount of design and code changes in the later stage of the project, and thus, increase the software maintainability [11].

One way to drive performance considerations earlier in the product's design is to couple performance requirement definition and specification with usability studies. Since our survey indicates that customer perception is a ubiquitous "requirement", this would provide a chance for designers to test the performance of their design prior to construction.

While customer satisfaction really drives the performance requirements in the industry, cost remains to be the biggest concern in the design of performance requirements. From our data, the industry seems to show concern only to those performance requirements that have direct impact to customer satisfaction. Therefore, measuring the product after it has been implemented favors the industry in pin-pointing only those ill-performed areas that may upset the customers. The industry seems to be eager to solve the performance issues by any means early in the design phase. However, due to cost and schedule concerns, they would also like to address only those performance issues that will impact the customers. Therefore, academia should focus on research that will solve these two problems simultaneously for the industry.

A preventative approach to design performance does not mean that no curative methods should be taken. Certainly measuring a software product is integral to ensuring that the product meets the performance constraints established by the customer. But this approach should be performed in concert with preventative methods proposed by research.

3. Usability

3.1 The Problem

The usability of a product refers to how easily users learn to use the product, how efficient they are at performing tasks once they have learned it and how many mistakes they make when they use the product. Usability also encompasses vague measurements such as perception and satisfaction. What tests must a

product pass to be deemed usable? Designers can quantify some aspects of a user interface, such as number of clicks or keystrokes, number of complaints or even the amount of time it takes for the user to become as productive on the product as they were on the previous product. And if analysts establish goals for these measures during the design phase, they can be an instructive method of determining usability and preventing problems with usability before construction begins. But other usability measures such as perception and satisfaction over time have just as much impact on the usability of the product. They are qualitative in nature and therefore difficult to target.

This is the point that our respondents start to rely on the skills of the team members. They indicated that an experienced analyst is better able to determine what makes a product “easy to use” than an inexperienced one, and so our respondents made sure their project teams always included as many experienced members as possible. In fact, the presence of one or more well-qualified members had the power to offset other issues with funding such as poorly supported testing.

Throughout this paper you will see that this approach is typical. When preventative methods are not available or unused and curative methods are insufficient, teams eventually fall back on experience to ensure that they meet the constraints of the NFR. There are a number of issues with this approach. We will discuss these later in the paper.

3.2 Our Findings

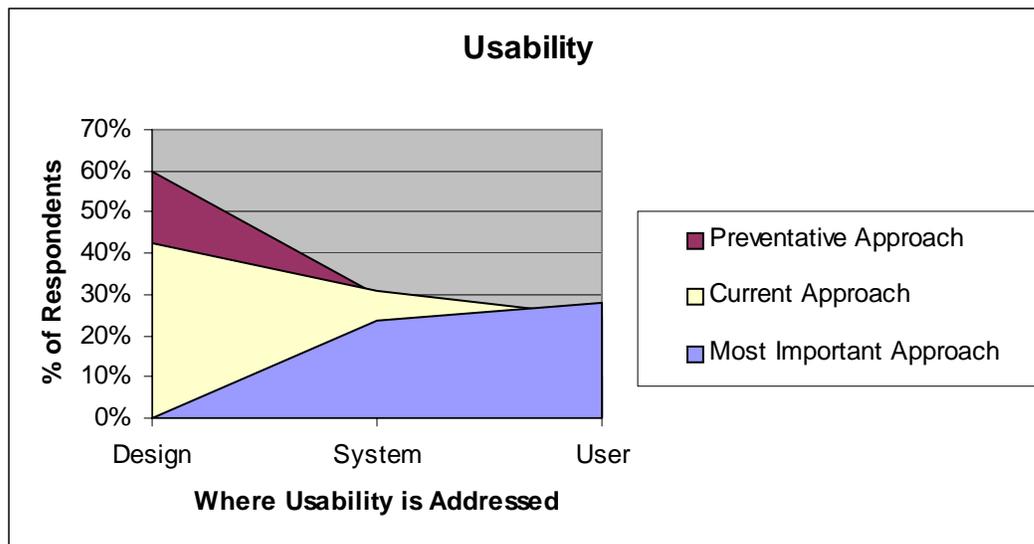


Figure 2: This figure describes where in a product’s lifecycle (at design, at system test or when the user receives the product) that our respondents address usability contrasted with an arbitrary preventative approach

The process of building a usable product lends itself well to the process-oriented approach. We found that many of our respondents address usability early in

their development process. Specifically, 42% of the respondents rely on methods such as prototyping, personas and other forms of customer feedback to get an early handle on usability constraints and goals. While still not a majority, this is more than the pre-coding consideration given to performance (31%), security (39%) and reliability (negligible). Part of the reason for this is that usability is so important to marketability of a product. Methods such as prototyping provide customers and investors a chance to immediately respond to the product and give indication to the designers where weaknesses with the design lie. This gives them confidence for continued funding and support. As a result, projects tend to use these tools early and consistently.

Another reason for success of preventative methods is the relative maturity of the tools themselves. Prototyping in particular, is ubiquitous, though teams apply it in different ways. Some respondents mentioned the ease of use of low fidelity prototypes such as sketches. Simply drawing different versions of a user interface is an inexpensive method of testing user response to different solutions. But respondents indicate that relying on such prototypes exclusively can leave subtle functionality open to interpretation. The gap between the business knowledge of even the most harmonious customer and designer leads to assumptions, misinterpretations and eventually, usability design flaws. And considering the falling costs of higher-fidelity prototypes that provide most of the user interface functionality without requiring a full design specification, respondents are increasingly focusing more effort on enhancing their pre-coding building efforts.

This does not mean that teams use evolutionary prototypes that become the base for the final code. Although 27% of our respondents do use this method, others indicated that code built during the prototyping stage is not created with the same controls as production code and should therefore be discarded. Furthermore, use of code generated during design would actually violate the preventative model, since our goal here is to focus on methods that occur before coding.

3.3 Where to go from here

Despite the acceptance of preventative methods of addressing usability requirements, respondents overwhelmingly indicated that the customer has the final say in whether the product meets usability requirements. Our respondents are a good example of how a preventative approach can be used successfully. Their customers are involved early through the use of interviews or personas to define usability requirements. Then after coding, tools such as testing, surveys and use studies are used to verify that they have met usability requirements. Addressing these requirements is important for ensuring that their products meet qualitative measurements such as user perception.

4. Reliability

4.1 The Problem

How do you know your system is reliable? Specifically, how do you ensure that your system meets the reliability requirements of the customer? There are numerous metrics for determining reliability: mean time to failure, defect reports and counts, resource consumption, stability, uptime percentage and even customer perception. Our respondents indicated that they use the following methods equally to flush out these metrics:

- Static Code Analysis
- Maximum Test Coverage
- Load Testing
- Stress Testing
- Automated or Manual Error Collection

The problem with all of these approaches is that they are curative. The only way to ensure software will be reliable using curative methods is to make sure that they test every situation that could cause reliability issues using real world constraints. One of our respondents simplified this statement by saying that “anything that CAN go wrong in a program is a potential cause of a reliability issue”. Not only is it costly to write test cases for so many scenarios, the execution of the tests themselves could take years. And that assumes you have discovered all of the test cases. There is no way to prove that all tests have been accounted for and therefore no way to know when you are done testing.

4.2 Our Findings

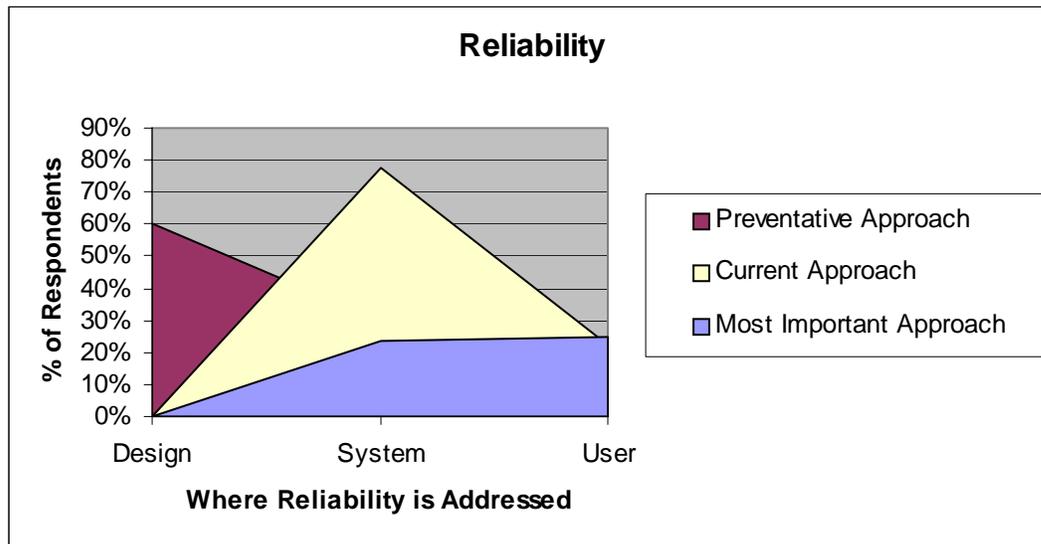


Figure 3: This figure describes where in a product's lifecycle (at design, at system test or when the user receives the product) that our respondents address reliability contrasted with an arbitrary preventative approach

Our respondents approached reliability with decidedly curative approaches and from figure 3, it is clear that they like it this way. Part of this may be due to the lack of tools for building software designs that will result in reliable software. Outside of selecting architectures or algorithms that are known to be reliable for the given domain, there are few ways of ensuring reliability before writing a line of code.

More than any other method, our respondents again rely on the skills of their teams. Instead of writing test cases for every possible failure, they rely on their analysts to design solutions that encourage reliability. They rely on their engineers to select the appropriate architectures and algorithms. And they rely on their developers to write code that won't fail. To offset this non-scientific approach to reliability they enforce rigorous testing procedures using methods like static code analysis and stress testing. Then when the product is in the customer's hands, they automatically gather the remaining failures and address them as they occur.

When asked to identify the most important tool for addressing reliability, our respondents consistently expressed responding to measurements such as crashes over time, uptime and more qualitative measurements such as customer perception or satisfaction. Although they expressed the need for more preventative methods, there was no such method that they knew of or currently perform that would be more important than the curative measures they already take.

4.3 Where to go from here

Unfortunately, relying on the selection of an appropriate architecture is still a passive preventative approach to addressing reliability. Given one weak analyst, engineer or developer, a project will be at risk of experiencing reliability issues.

Dromey [2] introduces one of the more promising approaches. First, with help from the analyst, the customer must define reliability constraints. This means identifying the possible conditions or inputs that a system will be required to handle, identifying the time or environmental constraints under which the system must complete its functionality, and validating that the chosen design will solve the right problem, correctly. Knowing these constraints, the designer should now create functionality that checks for the satisfaction of them and performs some action when they are not met. This approach is similar to the practice of adding asserts to warn developers that an assumed constant is not correct. Unlike asserts, this functionality is visible to the user and is not removed during compilation.

It will be difficult to rationalize to project sponsors expending effort to produce what is essentially non-value added functionality. Furthermore, this approach is inherently dependent on the correct and complete specification of reliability constraints. Nonetheless the added functionality proactively addresses reliability before coding and gives designers a tool to impact the reliability of the software.

5. Security

5.1 The Problem

Security has in recent years received much attention in both popular media and academia. Kuper [12] notes that a significant amount of IT spending has been focused on reactive, defensive “perimeter-related security” when instead a more proactive approach to defending the data itself yields better results in terms of security. Skalka [13] advocates that more than mere type safety and judicious programming practices proposed to safeguard C, C++ *et al.* from buffer overflows et cetera that we need to more strenuously march toward “language safety” inherent in managed code such as C# and Java. McGraw [14] brings to light the differences in the traditional “application security” that post facto looks to mitigate security issues, versus the emerging trend of “software security” that looks to engineer security into the design of the software itself; effectively minimizing the necessity for “application security.” Another argument by McGraw is that we need to appreciate the subtle difference in software security versus security software. Secure software (software security) must be fundamental to the architecture, not itself an adjunct (security software) we include at the perimeter of our systems, deployed to protect ourselves from our defective products.

While the general consensus is that we need to better architect our systems to be secure, there remains the issue of managing the diametrically opposed requirement of usability [15-20]. Take as example the existence of “low-friction” installers from Microsoft and others some years back. The intent was to provide a superior user-experience by having the system easily update itself with whatever extensions and/or drivers were mandated by the user’s immediate needs. However, this approach exposed a number of vectors into the system by unscrupulous persons thereby creating opportunity for exploitation. In the intervening years we have seen a reversal of opinion with more emphasis on explicit management of system security by the user. Microsoft Vista’s new security policies prompt the user for confirmation whenever a potentially compromising action is undertaken – even now there appears to be ongoing tweaking by Microsoft to better manage the user experience on this front.

5.2 Our Findings

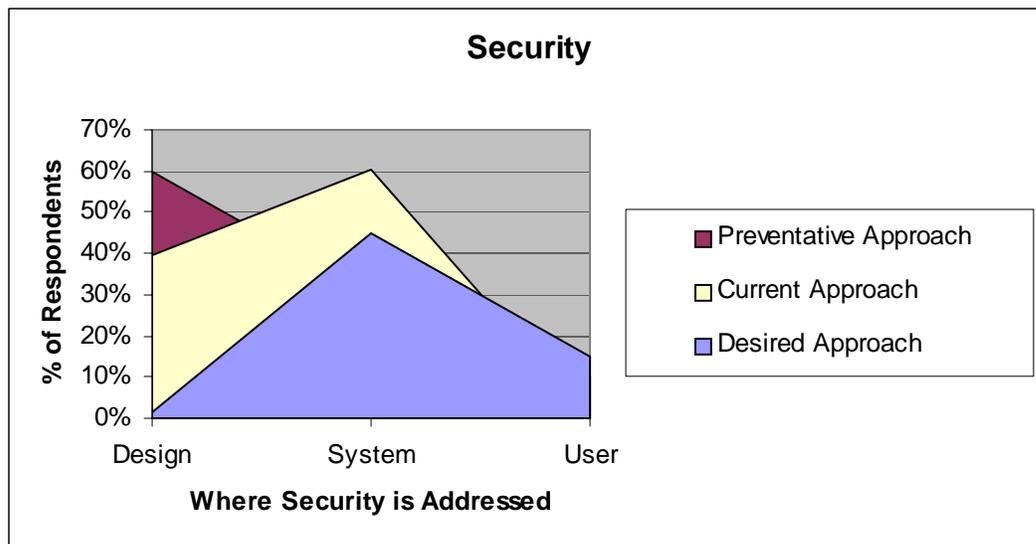


Figure 4: This figure describes where in a product’s lifecycle (at design, at system test or when the user receives the product) that our respondents address security contrasted with an arbitrary preventative approach

In our survey we attempted to determine how, and through extension when, in their process teams address security requirements.

45.5% of our respondents chose the following options:

- During design we analyze our data flow to determine vulnerable areas that could comprise security or data integrity.
- We use static code analyzers to identify potential security issues.

The least chosen option (15.87%) amongst our respondents is:

- We use managed-code (C#, Java, etc.) so that security issues are addressed [21] by the compiler in a consistently secure manner.

We see that nearly half of the activity of identifying security breaches occurs in the design and system stages of development through the initial use of data flows and a final shakedown using static analysis. If we include managed-code then we see that a fairly thorough portion of the software from design to deployment is ensured to be secure.

Moreover, just as the current practices indicate a healthy reliance on upstream processes to catch and fix security defects before they flow downstream to the user, approximately 45% of our survey respondents indicated that, given no other alternatives, they would ultimately measure the system rather than rely upon users to ensure they satisfy security requirements.

5.3 Where to go from here

The companies we surveyed publicly announced secure code development as a major initiative. Given the responses, we can conclude that this has been more than mere hyperbole. Since this research does not indicate whether this has had the intended effect, an opportunity for continued research exists. Additionally, respondents are equally applying the best practices of: abuse cases; security requirements; risk analysis; static analysis; external audits; et cetera at enterprise-level operations.

We believe the growing trend in managed code (C# and Java) along with ready availability of static analysis already provides much needed support toward engineering secure software. However, there appears to be opportunity to provide more robust means of data flow analysis to aid software engineers in evaluating the risks of proposed architectures.

6. Conclusions

Throughout our research we found that teams invariably fall back on the skills of their teams to ensure that their product meets quality requirements. Although this can be a remarkably successful method of ensuring quality, there are inherent flaws with this approach. First, it is costly. In an era where management will invest enormous effort to implement outsourcing strategies that break even only after several years of involvement, it is difficult to justify the extra expense of more experienced personnel. This is because it is impossible to determine the savings that was gained from having that person there. Of course, if a project is well funded to begin with, this may be of little consequence.

Secondly, with performance, our respondents found that the intuition of even the most experienced engineer is fallible. It is okay to discover that your “hunch” about the non-functional aspects of your system was wrong, as long as there is

still time to fix it. But it would be far more cost effective and risk-conscious if these hypotheses could be confirmed while the design is still on paper.

The third and most devastating problem with relying on skill to mitigate quality risks is that resources are by definition transient. You may have the most effective project manager or the most experienced engineer, but what happens when a better-funded or possibly emergent project comes along and entices your mitigation resources away with compensation that you cannot compete with?

Given these issues, projects should not exclusively rely on the experience of their staff to mitigate non-functional requirement issues. This does not mean projects should not seek and employ the most talented people for the job. What it means is that the presence of those resources should not be the only thing ensuring that projects meet performance, usability, reliability and security requirements.

Controlling the non-functional quality aspects of software projects is imperative for today's software teams. Despite the problems with purely curative approaches to ensuring quality, we found that the software industry does not have a common methodology for addressing quality requirements early in the design process. There are some requirement-specific approaches that move towards a preventative model, such as personas for usability and assert-like functionality for reliability, but things like the budget of the project, experience of the team, and customer feedback impact how the team implements these tools. From our survey it is clear that not only are preventative methods not in wide use, but most teams are not interested in using them, or not aware of their availability. In fact the most persistent method of ensuring non-functional requirements is simply relying on team experience. The importance of meeting non-functional requirements demands that more repeatable solutions be found.

References

1. Mylopoulos, J., L. Chung, and B. Nixon, *Representing and using nonfunctional requirements: a process-oriented approach*. Software Engineering, IEEE Transactions on, 1992. **18**(6): p. 483-497.
2. Dromey, R.G., *Software Quality--Prevention versus Cure?* Software Quality Journal, 2003. **11**(3): p. 197.
3. Davis, A.M. and D.A. Leffingwell, *Using Requirements Management to Speed Delivery of Higher Quality Applications*. Rational Software Corporation, 1995.
4. Lauesen, S. and H. Younessi, *Is software quality visible in the code*. Software, IEEE, 1998. **15**(4): p. 69-73.

5. Smith, C., *Increasing Information Systems Productivity by Software Performance Engineering*. Proceedings CMG XII International Conference, 1981.
6. Kanchana, B. and V.V.S. Sarma. *Software quality enhancement through software process optimization using Taguchi methods*. 1999.
7. Israr, T.A., et al., *Automatic Generation of Layered Queuing Software Performance Models from Commonly Available Traces*.
8. Jacqueline, F., et al., *Using Architecture Models for Runtime Adaptability*. 2006. p. 62-70.
9. Andreas, G. and S. Alistair, *Scenario-Based Assessment of Nonfunctional Requirements*. IEEE Transactions on Software Engineering, 2005. **31**(5): p. 392-409.
10. Balsamo, S., et al., *Model-Based Performance Prediction in Software Development: A Survey*. IEEE Transactions on Software Engineering, 2004. **30**(5).
11. Smith, C. and L.G. Williams, *Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives*. IEEE Transactions on Software Engineering, 1993. **19**(7).
12. Kuper, P., *The state of security*. Security & Privacy Magazine, IEEE, 2005. **3**(5): p. 51-53.
13. Skalka, C., *Programming Languages and Systems Security*. Security & Privacy Magazine, IEEE, 2005. **3**(3): p. 80-83.
14. McGraw, G., *Software security*. Security & Privacy Magazine, IEEE, 2004. **2**(2): p. 80-83.
15. Balfanz, D., et al., *In search of usable security: five lessons from the field*. Security & Privacy Magazine, IEEE, 2004. **2**(5): p. 19-24.
16. Cranor, L.F. and S. Garfinkel, *Guest Editors' Introduction: Secure or Usable?* Security & Privacy Magazine, IEEE, 2004. **2**(5): p. 16-18.
17. Gutmann, P. and I. Grigg, *Security Usability*. Security & Privacy Magazine, IEEE, 2005. **3**(4): p. 56-58.
18. Ka-Ping, Y., *Aligning security and usability*. Security & Privacy Magazine, IEEE, 2004. **2**(5): p. 48-55.
19. Hamed, H. and E. Al-Shaer, *Taxonomy of conflicts in network security policies*. Communications Magazine, IEEE, 2006. **44**(3): p. 134-141.

20. Smith, S.W., *Humans in the loop: human-computer interaction and security*. Security & Privacy Magazine, IEEE, 2003. **1**(3): p. 75-79.
21. Wagner, S. and T. Seifert, *Software quality economics for defect-detection techniques using failure prediction*, in *Proceedings of the third workshop on Software quality*. 2005, ACM Press: St. Louis, Missouri.