Using Automated Tests for Communicating and Verifying Non-functional Requirements

Robert Lagerstedt
Global System Management
Sony Mobile Communications AB
Lund, Sweden
robert.lagerstedt@sonymobile.com

Abstract—**In software development the code often must comply to a number of non-functional requirements, like architectural requirements. These requirements are often communicated and verified by writing guidelines and creating reports of the non-compliance. This way of communicating and verifying non-functional requirements is very costly since all developers needs to understand all requirements. It is also very hard for a developer to remember all requirements and it is easy to make mistakes. In software development much of the work is done in a tool-chain. The tool-chain contains tools like text editors, compilers, linkers, static analysis tools, automatic test frameworks etc. An alternative way to communicate and verify non-functional requirements is to add them to the tool-chain as automated tests and checkers, so the developers get fast automated feedback when they do mistakes. I have worked many years as a software architect defining and writing architectural requirements and my observations and experiences shows that the productivity is increased and number of non-compliant non-functional requirements is lower using the tool-chain feedback instead of using guidelines and reports.**

Index Terms—**Architecture, Non-functional requirements, Guidelines, Communication, Tool-chain, Automatic tests.**

I. Introduction

In software development, the code must often comply with a number of non-functional requirements. Many non-functional requirements are aiming to increase quality aspects of the software like scalability, maintainability, portability, safety etc. Some of these requirements are controlling the architecture of the software and how the code should be written, called architectural requirements. Some examples of architectural requirements are:

- "No dependencies from the lower parts of the architecture to the upper parts, since this decreases the possibility to reuse the lower parts within different products."
- "Class *Database* must only be used within the code package *DataLayer*, since this might increase cost when switching database technology."
- "Intent *ACTION_BOOT_COMPLETED* must only be used from the applications that are approved by the product architect, since this will increase the startup time of the product."
- "Never use the method *eval()* since it creates a security hole."

The architectural requirements often control the ability of the software to be reused and maintained in a cost efficient way. The amount of architectural requirements is often very large. The programming language itself has many built in requirements and most languages have additional set of standard requirements. Also many software platforms and operating systems have a set of requirements. In addition to these rules most companies have a number of company specific requirements and different parts of the software often have design documents containing a number of requirements. The company internal requirements are often written by architects, as me, inside the company.

When a developer is writing code they often focus on implementing one functional requirement at the time. This requirement can be verified using automated tests and is often located in an isolated part of the software. Even if one functional requirement can be implemented at the time the developer must consider many different non-functional requirements, including the architectural requirements. It is hard for the developer to focus on all non-functional requirements at the same time as they must be creative enough to implement the functional requirement. Ramesh et al. [2] identified non-functional requirements as a major challenge for agile development and that quality factors such as scalability, maintainability, portability, safety and performance are often weakly defined or completely ignored."

These non-functional requirements must be communicated in a good way so all developers and code reviewers understand the importance of them otherwise this might lead to quality problems in late phases of the projects or coming projects, as shown in study by Bjarnason et al. [3]. A common way of communicating the non-functional requirements is by writing guidelines for the developers. The compliances of the requirements are often verified by creating different reports.

An alternative way to communicate and verify non-functional requirements is to add them to the tool-chain as automated tests so the developers get fast automated feedback when they do mistakes, since in software development much of the work is done in a tool-chain. The tool-chain contains tools like text editors, compilers, linkers, static analysis tools, automatic test frameworks etc.

II. Problems with Guidelines and Reports

Even if many non-functional requirements are verified using automated tests, a common way of communicating and verifying non-functional architectural requirements is by writing guidelines. A company often has a set of common guidelines that all code must comply to and often the architecture of different software parts are written in design documents. In best case these guidelines are located in a central place like a wiki but often they are spread out in different document systems. Especially the company specific rules and rules from software platforms and operating systems are rarely located at the same place.

If all code must comply to the architectural requirements all developers must be educated. This is usually made by courses or presentations by architects. The developers should learn all requirements of the programming languages, the software platforms and the operating systems. This education can be held outside of the company by education providers or in-house. The developers should also learn all the company internal guidelines and this is often made as presentations by architects. This is time consuming and the architects are many times fully occupied with the roll-out of guidelines. When the roll-out is done all developers are expected to follow all the guidelines, but since mistakes are made, reports containing different measurements are created.

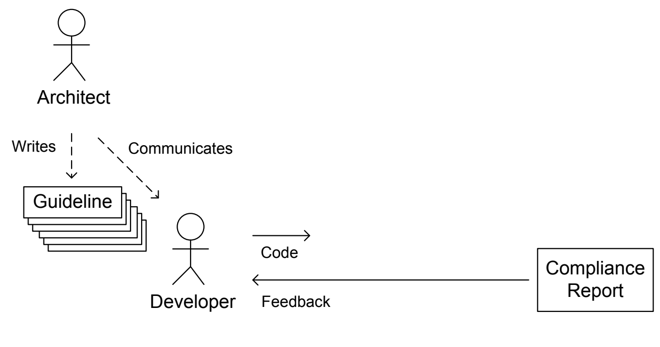

Fig. 1. The common way of communicating and verifying the architectural requirements. The architect is writing guidelines containing the requirements and the developer is using guidelines and gets feedback by compliance reports.

These reports can contain non-compliances of the architecture requirements or the guidelines, for example the number of dependencies from the lower parts of the architecture to the upper parts. The reports can also contain higher level of effects due to non-compliance, for example the effort needed to change to a new database technology. To understand the connection to the actual requirements, a root-cause analysis can be made from these reports. If many breaches are found within an area the action is usually new education for the developers.

I have seen several problems with this approach:

- Even if all the guidelines are written in a good structured way in a wiki, the developers tend to forget many details when they have to read a large number of guidelines.

- The time from when a developer makes the non-compliant code until feedback is received is very long. Especially if a root-cause analysis of a report must be made first. When the developer gets the feedback, long time has passed since the code was created and the developer needs time to recap.

- The productivity is decreased for a senior developer with too many guidelines. It is also hard to focus on all guidelines simultaneous and often the focus is changing towards the area where the latest non-compliances were found. This increases the risk of non-compliance in other areas.

- Since all guidelines must be communicated to all new developers, the time until they start being productive increases.

III. Solution using Automatic Tests with Fast Feedback

It is common to develop software in an iterative way meaning that a set of code is created and then sent to a tool-chain. The tool-chain contains tools like text editors, compilers, linkers, static analysis tools, automatic test frameworks etc. The tool-chain analyzes and transforms of the code in many different ways and the tools can give fast feedback to the developer about the progress. This means that the tool-chain can be used to prevent mistakes from the developer. Poppendieck et al. [1] means that “Mistakes are not the fault of the person making them; they are the fault of the system that fails to mistake-proof the places where the mistakes can occur.

The tool-chain already contains a set of pre-defined requirements, like the compiler do syntactic and semantic checks of the code and creates compiler errors when the requirements are not complied. There are also many static analysis tools that have pre-defined set of requirements. The tool-chain can be extended with additional checkers. This means that many company internal guidelines and design document can be added as checkers and automated tests, verifying the compliance of the requirements. This also makes it possible to give fast feedback to the developers about the non-compliance of architectural requirements and other non functional requirements.

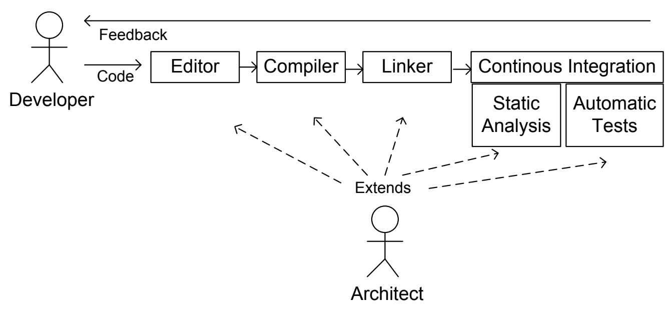

Fig. 2. The suggested solution of communicating and verifying the architectural requirements. The architect extends the tool-chain and the developer is using the feedback from the tool-chain to learn and verify the architectural requirements.

There are many tools in the tool-chain that can be extended with checkers. Some example of extending the tools to verify architecture requirements are:

- The programming language itself contains the possibility to add rules like using public, private etc.
- The static analysis tools can be extended with rules that checks static dependencies etc.
- Custom checker scripts that scans the code or code diff and analyze use of not allowed methods.
- Automated tests that verify backward compatibility of interfaces.

The checkers should be written in a way so the developers can learn from the feedback. This can be made by a short text and a link to the description and justification of the requirement. In this way a learning feedback loop is created, decreasing the need of formal educations and presentations. This is similar to a common way of learning the requirements of a programming language. The developer writes the source code and then learns by the feedback from the compiler instead of reading a complete book from first to last page.

One concrete example of a checker is to prevent that an application is receiving the *ACTION_BOOT_COMPLETED* intent. In the tool-chain a review system is handling code review and all reviewers have to approve before the code can be merged. The checker also acts as reviewer and approves the code if the code diff do not contain the specific intent.

I have seen many advantages by the approach:

- As mentioned, the need of formal education is decreased since it creates a learning loop using the tool-chain feedback.
- The feedback about the non-compliance is given to the developer when he is working with the code that was non-compliant. This means that the need of recap is limited.
- The rules can be adapted continuously. This is especially good, for instance, when a rule contains a list of approved objects that can be added to a white-list.

There are cases of architectural requirements that have challenges preventing the implementation of a checker. Some cases the requirements are not written in a concrete way but more as a vision, like “The documentation of a class must be made in a way so it is easy to understand the purpose of the class”. Some cases are when the analysis in the tool-chain takes very long time and would slow down the development and instead of blocking the code change, a notification is sent.

IV. Conclusion

I have worked many years as a software architect defining and writing architectural guidelines and I have done and observed the communication of non-functional requirements by using guidelines in both small and large organizations. A number of these guidelines have been verified using reports about non-compliance. I have also implemented and observed the use of automatic verification of the non-functional requirements by using the tool-chain feedback in a number of cases. My observation and experiences shows that the productivity is increased, especially for experienced developers, and number of non-compliant architectural requirements is lower using the tool-chain feedback instead of using guidelines and reports. I also see a decreased amount of time spent on communicating the architectural requirements, and a decreased amount of time spent on correcting the code that do not comply to the requirements.

References

[1] M. Poppendieck, T.Poppendieck, “Implementing lean software development: from concept to cash” Addison Wesley, p197

[2] B. Ramesh, L. Cao, R. Baskerville, Agile requirements engineering practices and challenges: an empirical study, Inform. Syst. J. 20 (2007) 449–480

[3] E. Bjarnason, K. Wnuk, B. Regnell, Requirements are slipping through the gaps – a case study on causes & effects of communication gaps in large-scale software development, in: 19th IEEE International Requirements Engineering Conference, IEEE Computer Society, 2011, pp. 37–46