How Design Trade-Offs Influence Software Testing

Anil Chakravarthy (anil_chakravarthy@mcafee.com)
Pramod Sharma (pramod_sharma@mcafee.com)
Sudeep Das (sudeep_das@mcafee.com)

Abstract:

While designing and implementing software, architects and implementers make many different design and implementation choices. These choices reflect the trade-offs they make in order to solve a software engineering problem. Understanding those trade-offs is important to ensure that testing is planned and executed effectively. A trade-off implies that one attribute of the software is sacrificed to strengthen another. Testers need to make sure to treat the strong and weak areas, such as performance, availability, maintainability, security etc. in appropriate ways. Understanding trade-offs helps testers use correct assumptions about the software behaviour. It also helps them to set right level of expectations about the software behaviour.

In this paper, we look at the various design and implementation trade-offs made by architects and implementers. To illustrate our point, we use real world examples of the products that we worked on. We then suggest how such trade-offs should be handled effectively to make the testing more effective.

Biography:

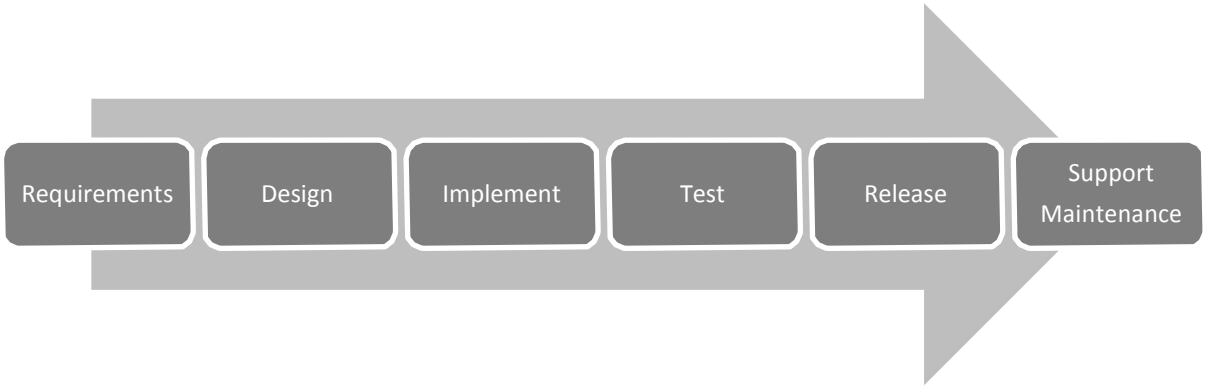
Anil Chakravarthy is a Technical Lead at McAfee, with more than seven years of software development experience. Highly passionate about quality, he strives to lookout for ways and methods to improve software reliability and usability. As an inventor of key technology solutions, his interests include security management, content distribution and updating.

Pramod Sharma is a Technical Specialist at McAfee, with more than eight years of industry experience. An acknowledged technologist, Pramod has eight patent pending inventions to his credit. Passionate about software quality and process improvements, Pramod is a vocal champion of adopting changes, and learning from the industry. His interests span security management for embedded and mobile devices, and scalable architectures for distributed systems.

Sudeep Das is a Software Architect at McAfee, with more than ten years of experience designing and implementing software. He started with a three person team and over time has seen it to grow to more than twenty, while improving upon development and testing processes incrementally over the years. His areas of interest span security management, content updating mechanisms, data protection and virtualization.

1. Introduction:

In general, a software development life cycle consists of requirements gathering, design, implementation, testing, release, support and maintenance phases. Each of these functions has significant impact on the quality of the software and on user experience.



Much of current software engineering discipline is focused on getting each of these functions right. It is also important to understand the influence of each of these functions on one another.

Each of these functions involves different sets of people with different areas of expertise. People involved in these functions need to understand the needs and limitations of people involved with other functions. For example, designers, while choosing a technology, need to ensure that the implementation team has sufficient expertise on the technology. Implementers need to ensure that the software allows for white box and automation hooks where necessary.

High bandwidth communication between the software engineering functions is key to getting the functional inter-dependencies right. The communication needs to be relevant and right information needs to be exchanged. In general, such communication is found to be between functions that are logically closely related. For example, communication between architects and implementers, between implementers and testers, between testers and support engineers etc.

However, it is also important for *non-immediate functional groups* to communicate effectively[1]. For example, testers and architects need to communicate effectively. This is one area where communication seems to be lacking. While the need to communicate is well understood, what exactly to talk about isn't. That's one reason why tester's participation in design discussions and architect's participation in test planning are often seen as mere formalities.

To demonstrate this shortcoming, and how it can be overcome, we take the example of one key aspect of software design, namely design trade-offs.

In this paper, we start with a high level view of software design trade-offs to familiarise the reader with the concept. We then take a look at some well-known software design trade-offs, and how they influence testing. We also look at the quality consequences that may occur when testers do not take into account these trade-offs. We conclude with some additional insights and recommendations to quality practitioners who may want to try out the discussed design trade-offs in their new projects

Our Intention is to get people to think about all aspects of software design, think about how those aspects influence software testing, and finally, think about what relevant aspects should form the basis for communication with the testing group. We also would like testers to think about how fore knowledge of various design facets can help plan and execute tests more effectively.

[1] Functional groups that immediately precede or succeed each other in a development cycle are said to be immediate functional groups. For example, implementation and design are immediate functional groups, but design and testing are not.

We believe this paper is to be of primary interest to architects, test planners, and project managers. However, all stakeholders in the software development process would find this paper beneficial.

2. What is a Design Trade-Off?

Architecture is the blue print of software. Depending on the constraints and operational environment of the software, various choices about the software's functionality, about the implementations, and about capabilities are made by architects. Requirements and constraints on the software force an architect to, at times, settle for a solution that gets the task accomplished but may not create the best possible outcome. Architects may also try to strike a balance to reconcile conflicting requirements. Such a balancing act sacrifices and weakens one aspect of the software to strengthen another. This is what we refer to as a design trade-off.

3. Trade Offs and Their Implications:

In this section, we list out the actual design trade-offs that software architects make. For each trade off we list here, we first provide a brief description of it. We then describe the test plan implications and test execution implications of the trade-off. We also provide a brief guideline on good practices that can help handle the testing impact of the trade-off effectively. We finally support the point with a real world example.

3.1. Robustness vs. Size:

Software, while operating, will run into error conditions. Robust software should handle such errors, do its best to recover from them and let the user continue on with the task. How well the software handles errors and recovers from them is an indicator of its robustness.

3.1.1. The Trade-off:

Robustness requires error handling and recovery. Error handling and recovery needs additional code to be written and additional code translates to additional binary size. When constrained for size, architects may sacrifice error recovery to achieve the size requirements. One way is to adopt the "fail fast fail safely" strategy, where any condition other than the "happy path" causes a controlled, handled but unrecoverable failure.

3.1.2. Testing Implications:

Before planning the test cases, testers should clarify with the architects if this design trade-off exists. They should also check with the requirements team about such a trade-off being acceptable and if so, to what extent. At times, requirements and error handling behaviour may need to drill down into details of what is acceptable and what is not.

Without this three way communication between architects, testers and requirements team, and with incorrect assumptions about error recovery, testers will raise significant number of defects for "as designed" cases. A sudden spike in the number of defects alarms management stakeholders about software quality. The concern is aggravated when those defects are marked "As designed". Considerable amount of time and resources are expended in handling the lack of communication.

3.1.3. Example:

We had to create a web installer for our product, a small application that would be downloaded by the end user and then executed to download and install the rest of the product. The web installer had severe size constraints, and so, for various error conditions, the design and implementation simply

printed an error, and exited.

There was lack of early communication between testers, architects and product manager on this aspect. The testing expectation was to have the installer recover from errors and allow users to continue. This was contrary to observed behaviour and caused debates, arguments and escalations about the product behaviour.

3.2. Make vs. Buy:

3.2.1. The Trade-off:

Architects, at times, may choose to use third party code or libraries instead of spending significant resources on creating supporting technology from scratch. Frequently, this third party code is open source. The flipside of this is that the software now has code on which the development team may not have expertise. The third party code may have its own share of defects that the development team may be ill equipped to fix.

3.2.2. Testing Implications:

Testing team members should clarify upfront the usage of third party code. If there is third party code involved, then all stakeholders should agree on the following:

a. Treatment of defects encountered in third party code:

   The possible options are

   a. Agree to fix by the development team: This is not recommended in practice unless the development team has expertise on third party code.
   b. Fix by third party code providers: This is a feasible option, and is highly recommended. There is a risk that some of the third party code may not be fixed by the time the software needs to be released.
   c. Ignore: This is not recommended as this implies deliberately shipping buggy software.

b. Adjustment of code coverage metrics:

   The possible options are:

   a. Completely ignore: In code analysis and coverage metrics, completely ignore any metrics from third party code from coverage reports. If the third party code is known to be robust, stable and well regarded, then this is the recommended approach. This is so because certain well regarded third-party open source components like Boost (C++ libraries), libcurl (URL transfer library) etc. follow well established development processes and are known to be mature and stable. As such, there isn't much gain in establishing coverage metrics for such code.

   b. Adjust for functionality: For not so mature third party components, it may help to map the parts in third code that will actually be used by the product being developed, and include them in the coverage metrics. While somewhat difficult in practice, a recommended way is as follows:

      Consider a third party i/o library that has exposes a read() and a write() function. Let's say the read() internally calls read_usb() , read_sd(), and write() internally calls write_sd() and write_usb().

      i. Identify all API (Application Programming Interfaces) in the third party code that are ACTUALLY used by the product being developed. In this example, it is read() that will actually be used.
      ii. Evaluate a code flow graph of only those APIs of the above step in third party code.

This will provide all functions and classes that will be in the code path of the API of third party code. In this example, the flow graph for read will traverse read_usb() and read_sd(), but will not touch the write functions.

iii. Map code corresponding to only those third party functional code in coverage metrics.. For this example, you will include only code for the read, read_sd(), and read_usb() in your metrics

Not being aware of this trade off and not planning for it may have undesirable consequences as below:

Inaccurate and misleading code coverage metrics: The application may be using only a small part of the third party library, and hence only that part will be covered in coverage metric. For example, if a program with 1000 LOC (Lines of Code) uses a third party component with another 1000 LOC, but exercises only 200 lines in there, then coverage metrics will indicate only 60% coverage or 1200 out of 2000 total LOC.

Misleading static analysis metrics: Static analysis applications like Coverity and Fortify will analyse the third party code as well, and will indicate problems with them.

3.2.3. Example:

In one of our products, there was need for significant usage of a particular compression algorithm and http(s) capabilities. Building them by ourselves would have been at a significant cost, and longer time to market, so a decision was made to use third party components that, in addition to what we needed, provided a lot of other capabilities that we didn't need. Code coverage and static analysis raised red flags after analysis, and considerable time was spent later to tailor the results to reflect the true state of the code.

3.3. Feature Parity vs. Platform Disparity:

3.3.1. The Trade-off:

In many instances, the same software may need to be supported on multiple platforms and operating environments. Different operating environments and systems have different capabilities. Depending on the feature set of the software and the platforms it is expected to be operational on, software designers may choose to:

a. Support a feature set that is limited but uniform across all platforms, based on lowest common denominator of platform capabilities.

b. Add platform specific code to support different environments.

3.3.2. Testing Implications:

Product managers, architects and other stakeholders should evolve consensus on handling platform disparities. From a testing perspective, it's a common error to create test cases for one platform and then extend it to all platforms. Testers then test the features on platforms that don't support it, and then create defects for them. This gets repeated for each additional platform that the software is supported on.

Testing professionals should not assume feature parity across platforms. While creating test

cases, they need to ensure that feature disparities across platforms are taken into account. If this is not done, then midway through the development cycle, test cases will need to be revisited, altered and re-executed to account for platform disparities. Planning for this trade-off upfront is important.

3.3.3. Example:

McAfee's products support multiple platforms. For example, on Windows, a feature set exists that is tied to getting OS notifications about user dial up connection. Not having a reliable way to do this on Linux, Solaris, AIX and HPUX the feature was not available on these platforms. This caused significant user experience and consistency concerns to the testing team who assumed feature parity across platforms.

3.4. Security vs. Usability

3.4.1. The Trade-off:

Different software systems have different security needs. For example an apartment doesn't need Fort Knox security, and prison system cannot depend on padlock security.

Although more security checks improve security assurance, they may negatively impact user experience by making it harder to use the system. The degradation may be seen in performance for example, when dealing with large amounts of cryptography, and in user experience where a user may be forced to establish credentials at every operation. Given that security can never be fool proof, designers can only raise the bar. Both Security and usability are competing requirements, designing software with right balance based on acceptable levels is the key trade-off.

3.4.2. Testing Implications:

Testers responsible for security testing should discuss this aspect with architects before planning their testing. The discussion should also include product managers to ensure that the level of security and usability is in line with user expectations.

Security testing tools should be configured based on the security design of the product. Testers should exercise reasonable judgement on the scope of security testing, rather than running all possible exploit tools on the product.

Similarly, testers validating usability need to be aware of the security aspects that may limit usability, and plan and execute their tests accordingly.

3.4.3. Example:

One of McAfee's products is an enterprise client server product, but is an on-premises solution where both client and server are inside the enterprise network boundary. As such a few security measures relevant to internet facing systems were dropped, because the server was not me ant to be internet facing. Because of this misunderstanding the security testing team ran a significant number of tests in that area, and came up with defects that were not applicable to the situation.

3.5. Memory vs. Performance:

3.5.1. The Trade-off:

Another classic trade-off is that performance drops if less memory is available and improves with

more memory. Depending on the resource profile of the target system, designers may choose algorithms that are either resource intensive or resource constrained.

3.5.2. Testing Implications:

Performance testers should discuss this aspect with architects and product managers before planning their performance testing. Depending on requirements, architects may replace entire algorithms while keeping functionality intact. This may cause performance metrics to be outside expected bounds. By communicating with architects about this aspect, performance testers can ensure that their performance testing metrics are an accurate reflection of application performance.

3.5.3. Example:

We had a product that was originally intended for a general purpose computing environment, and used a resource intensive algorithm to deliver performance. In a subsequent version, the product was required to support mobile devices as well, and the algorithms were changed to resource conservative ones. The product in the newer version had a resource conservative algorithm across the board. When the general purpose compute environment was taken up for performance testing, the performance results were below the previous version benchmarks. The performance testing team raised slow performance related issues, as a result time and effort was spent to discover the source of the problem. After consultation with the product manager, it turned out that a slow performance was expected because of new the algorithms. The testing team was unaware of this design change, tested against the original benchmarks, leading to significant waste of resources.

4. Conclusion:

Effective communication across different functions is fundamental and a key success factor in delivering quality software. Tri-partite communication between architects, testers and product owners (managers) is one such example. The effectiveness of such communication depends on the right set of questions being raised and the right set of problems being addressed. The purpose of such communication is not to make architects out of testers or testers out of architects. The purpose is to encourage early collaboration, exchange of information about the roles and responsibilities of each of the functions, how it is intended to be done, and how it may affect other functions.

We believe that the various trade-offs presented here will encourage architects to discuss in-depth the trade-offs during their interaction with testing teams. We also believe that testers will actively ask questions about various design choices instead of being observers in design meetings. There are many more design trade-offs than the ones discussed here we hope software professionals add to this list over time from their experiences

Knowledge of design trade-offs helps testers to plan software test activities wisely. It sets the correct expectation about what the software is intended to do. Mismatched expectations on software behaviour causes significant time to be lost in unproductive testing activities, and may also require significant rework.