How Architecture Decisions Solve Quality Aspects

Anil Z Chakravarthy

(Anil.Z.Chakravarthy@Intel.com)

Charulatha Dhandapani

(charulatha.dhandapani@gmail.com)

Abstract

Software architecture is about making fundamental and crucial decisions on individual elements which could prove to be costly to change, once they are implemented. Solving quality aspects such as performance, reliability, and security as part of the software design can be daunting and challenging.

In this paper, we will dwell on some software architecture decisions and their impact on achieving quality software products.

- Single threaded vs multi-threaded programming
- Blocking calls vs non-blocking calls
- Reuse of the components (no need to reinvent the wheel)
- The technology we choose to solve the bigger problem
- Phoenix- kill the current design and redesign

Biography

Anil Z Chakravarthy is an Engineering Manager at Intel (McAfee now part of Intel Security), with more than eleven 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.

Charulatha Dhandapani is Software Engineer at Intel (McAfee now part of Intel Security), with more than ten years of software development experience. She is a very passionate about working on emerging and innovative technologies and always strives to find innovative ways to improve quality and reliability.

1 Introduction

Quality aspects have a system-wide impact and hence their implementation should also be system-wide. Quality aspects are the factors which have impact on the runtime behavior, user experience, and security. These can be further classified into modularity, testability, maintainability, robustness, scalability, throughput, etc. One of the most challenging aspects of the software design is to create a system which has built-in quality aspects needed by the end user.

This demands that an architect have a great deal of understanding on the quality aspects, their impact on the system and approaches to satisfy these in the architecture of the system. Architecture patterns addresses various issues in software engineering with an impact on many quality aspects of the system.

In this paper we present our experiences from our architecture design decisions we had made to resolve some of the above quality aspects.

2 Architectural Decisions & Quality Aspects

2.1 Architectural Decisions

Architecture is the blueprint 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. Every application/software has different requirements and goals which need to be considered in the design phase. One of which is to identify the actions that the application is going to do in its lifetime and also most importantly identify the resources that the application would need to perform its action/job. Each of these choices has pros and cons which need to be considered to make a right choice at the design phase that would create the best possible outcome.

In this section, we list out some of these design decisions that software architects make. For each of these design decisions, we will provide a description of the decision and then dwell into identifying the quality aspects of the decision. We will finally share the decision we have made and the quality aspect we tried to address.

2.2 Quality aspects

In this section, we will list the quality aspects that can be addressed by the architectural decisions that are made. There are many trade-offs for each approach which deals directly with some of the quality aspects like

- Performance & concurrency
- Generating code
- Code maintenance
- Validation
- Porting
- scalability

The following sections talk about each architectural decision and the quality aspects it impacts.

2.3 Single threaded vs multi-threaded programming

Based on the requirements, an architect has an option to choose one of the below options in his / her architecture:

- single-threaded,
- large thread-pool or
- fixed thread count

2.3.1 Quality Aspect

If an application/software is meant to perform one action at a given time or if it is acceptable to queue the actions to be sequential, then we would ideally go for a **single threaded** application. This would by design resolve the traditional **synchronization/concurrency** problems that come with multithreaded applications/software.

Another quality aspect that we need to consider is **Generating code**. To have a cross platform application/software we need to write code to either have a wrapper which would abstract the threading APIs (platform specific code) and then start using these in the code. But, if an application is a single threaded application, we would avoid use of any threading modules, making it easier to develop platform agnostic code.

2.3.2 Example

In one of our Projects, we developed a single code base which runs on a variety of platforms. This presented a challenge to ensure we chose the right architecture and the language that can be built across these heterogeneous platforms. One of the areas that we identified was that our software didn't need multiple threads and all the actions can be performed sequentially. We adopted the single threaded programming approach and have seen in our experience that most of the issues related to concurrency, synchronization as already addressed by design. We also have benefited by not generating code for cross platform threading module, or using/including any 3rd party code which is common across platforms.

2.4 Blocking calls vs non-blocking calls

Traditionally in multi-threading programming we use locks to synchronize access to shared resources. The thread that attempts to acquire a lock that is already held by another thread, it will be blocked until the lock is released. Blocking a thread is undesirable for some of the below reasons:

- A blocked thread cannot accomplish anything.
- If the blocked thread had been performing a resource intensive task, those resources are now blocked from being used by other process threads
- Possibilities for deadlock; the app or process comes to a halt (effectively crashes)

2.4.1 Quality Aspect

Unlike the above, non-blocking calls do not suffer from these downsides. Non-blocking in conjunction with single threaded programming allows us to resolve these issues by design.

Asynchronous programming allows more parallelism. Such a programming model relies on mechanisms such as callbacks and events to trigger/initiate a task or signal task completion.

2.4.2 Example

An asynchronous programming model allows an application to respond to service requests that don't need to be entirely processed immediately. In this scenario, it is best suited for the user to just initiate the operation and get notified when completed. This resolves many of the quality aspects like **resource utilization**, **deadlock**, and unnecessary **wait**.

In a network application, where the user has to be notified when there is data available, we will need a mechanism to register for such a notification and go ahead with other operations. As and when the data is available the callback will be called and an appropriate action can be taken.

2.5 Reuse of the components (no need to reinvent the wheel)

In most business software applications, the workflow or the actual solution implementation can be the business logic and rest of the application can be implemented using standard infrastructure components.

2.5.1 Quality Aspect

Generating, building, or validating such common infrastructure components pose a lot of inherent quality aspects that must be considered. Below are a few:

- Need to build the expertise in generating/building/validating these components
- This is not the core area of the solution one would like to offer to their customers
- Stability issues as part of build these components in house
- Adhering to the regular updates that are published to these components
- Ensuring these components adhere to security and performance constraints

Instead when we reuse infrastructure components, we realize these benefits:

- **Assured quality,** as the reused component is written and delivered by SMEs (Subject Matter Experts)
- **Adhering to standards** is a given as these are developed by SMEs as per open standards
- **Increased velocity** in developing the core business logic one would want to deliver to their customer

2.5.2 Example

As a multiple platform software/application, we had multiple requirements like:

- Cross platform support (e.g.: Windows, Linux, Mac OSX, etc.)
- Need for a network module to perform network operations such as secure communication
- Packing/unpacking module to perform zipping/unzipping operations
- And so on

In order to meet these requirements, we either had to

- start designing/developing/validating these modules
    - This is not our core competency and also not the actual solution that we would want to deliver to our customers
- (or) re-use these infrastructure components developed by 3rd party in order
    - **Increase the velocity** of development of the product and also be **assured of the quality** which is already tried and tested in the field by many others.

Reuse of components helped **lower our costs (smaller team)** and **faster development** to deliver the product to the customer/market **on-time**.

For example: if we have to develop a network library, which handles HTTP, HTTPS, FTP, Proxy with all the authentication mechanisms like NTLM, Digest, Kerberos, etc., below will be the challenges and costs.

- Additional resources
    - Will need a SME in the network domain
    - Need a SME in a crypto domain
- Time

    - Development of these modules will take time, before we venture into the business logic (which is the core competency)
    - Validation towards functionality of these modules
    - Validation towards complying to the standards
    - Different network topologies to be setup for validation
- Risks
    - This is a new 1.0 source which needs field time (feedback) to stabilize over a period of time
    - Will be engaged with customers on these issues than the feedback on the actual business logic or workflows

Versus, if we choose cURL 3rd party open source, all of the above are a given.

cURL is developed and supported by open source,

- No need to hire SMEs
- No need to validate this library
- Focus on the business logic
- Feedback from customers will be more on the business logic

2.6 Phoenix- kill the current design and redesign

As a successful product, customers might be happy with the current version\design which is stable and is good enough for the customers to resolve their needs. But, the product development cycle and team would know best when they think the time is right to redesign \ re-implement \ re-architect a legacy product which has been in the field.

2.6.1 Quality Aspect

During the development and maintenance of a software/application, there are changes/fixes/improvements for the betterment of the product in the field. Most of these changes\fixes are due to one or more of the following:

- Feedback on the workflow
- Performance (in-the-field adoption)
- High Priority / Severity issues
- And so on

As part of customer feedback over a period of time, we also learn how customers intend to use certain features of the product. We also learn about the workflow of the entire solution stack. This is valuable feedback, which can be considered for our future implementations/design.

2.6.2 Example

In our experience, we have an existing product which has been successful in the field for more than a decade. Product had evolved over this period of time based on the feedback from the customers and also was enhanced with new features based on requests from multiple customers.

There was a need for the product to evolve further to serve customer needs with new infrastructure components and also resolve a list of design quality aspects. The re-design / re-architecture of the current product updated workflows which also considered the seamless transition between the legacy and next generation versions.

For example: we had a product/application which was mainly supporting Windows platform and was designed and developed over couple of decades ago with the technology which was prominent in that time period.

The product had served best in the field with great quality and with more net promoters. Having said that, when we envisioned the future possibilities of the product we realized that the current/legacy design/architecture/implementation is not going to serve or with stand another decode or more.

We realized that we had to re-design/re-architect the application to meet our vision and future possibilities. This was accomplished and the results were highly positive.

Along with the above mentioned architectural decisions, we also had few more considerations (like, choice of language, technology, etc.,) that added more value in terms of improved quality and led to a remarkable reduction in delivery time.

3 Conclusion

We had a unique opportunity to **re-invent** our product/solution which had been in the field for more than a decade. In this process, we considered the design patterns or architectural decisions which would influence and impact the quality of the product/solution/software. This paper presents a few of those concepts that we had considered and implemented in our project. Basing our redesign on all of the quality aspects discussed in this paper, we saw a **significant improvement** in the **quality** and the **velocity of the development**.

These architectural decisions helped us achieve our vision without impacting the quality and time aspects that are critical to meet customer requirements and to be in the market at the right time.

References

Web Sites:

IASA SPAIN – An association for all IT architects.
http://www.tutorialspoint.com/software_architecture_design/introduction.htm (accessed July 7, 2016).

Wikipedia – Architectural Patterns. https://en.wikipedia.org/wiki/Architectural_pattern (accessed July 7, 2016).

Tutorials Points – Software Architecture and Design Tutorials.
http://www.tutorialspoint.com/software_architecture_design/introduction.htm (accessed July 7, 2016).

Anil Z Chakravarthy 2012. How Design Trade-offs Influence Software Testing.
http://www.uploads.pnsqc.org/2012/papers/t-79_Chakravarthy_paper.pdf (accessed July 7, 2016).