Patterns for Initial Architectural Design on Agile Projects

Eduardo Guerra, National Institute of Space Research (INPE), Brazil
Rebecca Wirfs-Brock, Wirfs-Brock Associates, Inc, USA
Joseph Yoder, The Refactory, Inc., USA

Abstract: *Agile methods have brought to the software development industry a wave of new ideas and better approaches for delivering more value to the clients in less time. The focus of agile methodologies on software design is evidenced by accepted practices such as test-driven development (TDD) and refactoring. However, currently there are no well-recognized and agreed upon approaches for architectural design. In fact, the most recognized approach is to simply let the architecture "emerge" and refine it over the life of the project. Consequently, mature teams have their own approaches for handling architecture, while teams that are new to agile development do not have a good understanding about how best to handle architecture. This paper is the beginning of an effort to address this issue by identifying patterns about practices for architectural design on agile projects. The set of patterns presented in this paper focus on the initial architectural design to be applied in the beginning of a software project.*

1. Introduction

Agile methods are currently established as an important approach for software development in industry. They have popularized among developers some influential design techniques, including test-driven development and refactoring. However, agile methodologies have a lack of practices for architectural design as shown through industry reports (Binstock 2014), by a systematic review (Breivold et. al 2010) and by a grounded theory study (Waterman et. al 2012).

The goal of this paper is to present the first patterns identified for architectural design on agile projects. These patterns focus on the practices that can be applied at the beginning of the project for the initial architectural design. The target audience for these patterns are software architects and software developers that work on projects with non-trivial architectural problems.

2. Climbing on the Shoulders of Giants

Also known as **Start With a Reference, Reference for the Architecture**

"If I have seen further it is by standing on the shoulders of giants." - Issac Newton

On agile projects it is desirable to start the implementation as fast as possible, but there are several things that need to be defined about the architecture before starting to write code. For instance, it is important to define the basic component types that are going to interact with the application resources and how they are going to be combined to implement basic functionality.

* * *

How can you quickly define the basic application architecture and the main component types that will satisfy the requirements?

There are several companies that focus on a type of application or architecture and specialize on it, having experience and a reference to be used in further projects.

There are several well-documented frameworks and platforms that present solutions for several common problems to the type of application that it is designed for.

Agile methodologies do not have a long inception phase dedicated to architectural design, and they value starting implementation activities as soon as possible.

Having some architectural definitions, especially on how the components are going to be divided and organized, is important in order to enable the rapid implementation of the initial user stories.

Therefore:

Use an existing reference compatible with the application platform and suitable to its needs as a starting point.

This reference can be a reference architecture (Avgeriou 2003), such as Service Oriented Architecture (SOA) or Client-Server, that defines the main kinds of components and how they collaborate to fulfill the requirements. Additionally there are reference implementations (that is a skeleton of what should be implemented) and a framework-based implementation that provides support of aspects of a reference architecture. For example the reference can also be supported by frameworks, such as Java EE components (JCP 2009) or Ruby on Rails (Fenandez & Faustino, 2014), which provide implementations of architectural elements, or by implementations of similar projects developed by the same team or by the same company.

When there is no reference from an existing platform or a previous project, a pattern or a set of patterns can be used to represent the main idea of the architecture. For instance, you can use ***MVC*** (Model-View-Controller) as a reference for interactions between the system and users and ***Master-Slave*** as a reference to how distributed processing should work. It is important to note that behind the reference architectures and the frameworks there are a lot of patterns concepts, however references are prefered since they provide a more concrete way to implement them. A ***Metaphor***, one of the core Extreme Programing practices in its first edition (Beck 1999), is another approach to define an architectural reference.

It is important to remember that you don't always need to use all the reference ideas and you can also complement it in the future with additional capabilities in support of other application-specific requirements.

A benefit of using an existing reference is that all team members will have knowledge of how the software should be structured. Even if they don't have direct experience with such an architecture, there are lots of documentation and code example references of how this architecture can work. This allows for a fast start to implementation, that can follow the reference and adapt it where needed.

A drawback of this pattern is that the team can become tied to the reference, and miss seeing beyond it to find more appropriate architecture. Sticking with a reference architecture can also prevent developers from solving problems that are not targeted by the used reference. Another problem occurs when developers use the reference blindly, not understanding why it is being used on the project. So while they may use it, they do so in naive ways that compromise the architecture.

* * *

It is very dangerous to use this pattern alone, without **Finding Where it Hurts** for technical requirements that are out of the reference scope. This can lead to an architectural technical debt that can be hard to repay in the future.

In the patterns for starting a TDD session (Guerra et al. 2014), the reference architecture used can be used to ***Understand Class Role in Architecture*** of a class that is going to be developed; and then to **Know Your Neighbourhood** to find out what are the expected class interactions.

In project LEONA, Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America, the application used as a reference the architecture of previous projects in the same institute. However, because of the nature of the data, in the beginning of the project we decided that the reference was not suitable for many of the requirements. Despite the team changing a lot of things to support these requirements, the initial reference still was useful as a starting point.

In the development of the system SADE (Perillo et al. 2011), the architecture was based on Java EE with Struts and Spring framework. Despite other additions made to the architecture *during the project, this reference helped in the beginning of the project in the implementation of the initial user stories.*

3. Find Where it Hurts

Also known as **Identify Technical Challenges**

While a reference architecture provides a solution to several common requirements, it does not cover all of them. It is important for an architecture to handle challenging technical requirements and address specific architecture qualities. Simply selecting an appropriate reference architecture will not guarantee success. Most projects have some unique characteristics that will require additional attention to the architecture.

*   *   *

How can you identify relevant points where the architectural design should focus?

Despite the fact that a reference architecture provides key solutions to the architectural design, rarely does it support all the important architectural requirements.

Some projects are focused on rebuilding existing systems that are difficult to maintain and evolve and currently do not fulfill all the requirements concerning system quality attributes.

Experienced developers and software architects are good at identifying some requirements of the software that are likely be challenging to implement using the chosen reference.

Not all quality attributes requirements are relevant to a project.

On agile projects it is desirable to optimize the time spent in the initial requirements analysis, especially related to non-functional requirements that are speculative or haven't been verified. But you do need to identify the significant threats and failures to/in the architecture as early as possible.

If an architectural requirement is addressed too late in the project, however it can require a lot of rework to fix and cause the project to be delayed or fail.

Therefore:

Early on, identify the challenging technical requirements that are important for the project, so they can be handled at the optimal time.

It is important to focus on the important technical requirements that can bring high risks to the project. It is advisable to not try to identify and immediately address all architecture challenges, because that might take a long time in the beginning without returning any value to the project. Also, perceived technical challenges may be irrelevant to the functionality that is actually implemented.

For instance, if the client has a previous system that will be discontinued, which has performance problems due to the high amount of data, that will be one of the important issues that the architecture needs to address. As another example, in a system where it is critical for the customers to have a high availability, the team needs to make sure that the architecture has the appropriate mechanisms to support this requirement.

It is important to share and discuss these requirements with the team and the clients, so everyone is aware of the technical challenges that are going to be faced during the project and their importance to the delivery of business value. This list of technical challenges should not be static; items can be introduced and eliminated throughout the project and be included as part of the roadmap.

Using security as an example, you are only as strong as your weakest link. Security is an emerging property that cannot be defined locally (Fernandez 2001). It is hard to just simply take a single framework and instantly get good security. But the security requirements on a given project may be not be critical, or there may already be well-established solutions. In those cases, the implementation can evolve to address security concerns during the project iterations, and do not need a big initial architectural effort.

* * *

The challenges found in this pattern should not include things that are handled by the chosen reference, because when you use a **Reference Architecture**, you can see beyond the problems that it already solved. For each challenge identified, you need to have a **Technical Plan** to handle it.

//I think we can reference the patterns related to discussing the quality attributes

//TODO: known uses

4. Plan for Responsible Moments

Also known as **Technical Plan**

In agile projects it is important to avoid unnecessary upfront design, and leave decisions to the "most responsible moment" (Wirfs-Brock 2011). However, there may be several technical challenges that, once identified, need to be handled by the architecture in order to sustain ongoing development of functionality.

*   *   *

How can you handle the technical challenges in the beginning of the project without a full architectural design upfront?

There are some important requirements that need to be handled by the architecture that put all business in risk if they were not achieved. There are different levels of technical risk and different approaches for handling each risk.

While some approaches to deal with technical challenges influences the way that all the software need to be created, others can be easily encapsulated.

It is not desirable to have a long period of upfront architectural design in the beginning of an agile project, because that can postpone the implementation of functionality that provides business value.

As with other requirements, technical requirements can also be uncertain, and handling them prematurely can be a waste of resources.

Therefore:

Create a technical plan for how and when to handle each of the technical challenges and evolve it throughout the project. This plan needs to define how to identify these important responsible moments and circumstances when it is appropriate address these technical challenges.

This plan, which can and will evolve, does not need to present the solution that necessarily is going to be implemented. However, it should state when the solution should be considered for implementation, or even ways to identify if and when that challenge needs to be faced. For instance, for a critical issue on which other features may depend, the plan could be to address it in the first iterations and have feedback about it as soon as possible. However, as another example, for a technical challenge related to performance, the plan can be to perform measurements to determine the current performance and then identify when something should be done to improve on it. This technical plan should try to prepare the team and the system to the most responsible moment for implementing an architectural feature.

Even when something is going to be handled later, it can be part of the initial plan to create interfaces, decouple components, or create hotspots (reference needed here?) early on. The effort to create a structure that allows for future evolution that implements the solution later can be considerably lower cost than prematurely implementing unnecessary architectural functionality. So this plan can include several actions that should be performed on different phases of the project.

The technical plan should be based on the technicals risk that are based on the requirements of the project. The effort required, the influence on other functionalities and the uncertainty on its constraints are other important things to consider. Sometimes it is necessary to include in the plan an architectural spike [http://www.scaledagileframework.com/spikes/] that can reveal more information to enable the definition of appropriate next steps.

A benefit of applying this pattern is that it allows the project to start fast, but without neglecting important architectural aspects. It leaves for later what can be handled later, and it puts in the beginning of the project architecture tasks which have a high risk or a high influence on the rest of the system. Another benefit is to make sure that all the team is aligned with how the technical challenges will be handled, allowing them to contribute, give feedback, and bring new inputs as soon as they are available.

A drawback of this pattern is that the technical plan identified for some challenge can be inadequate or inappropriate, and the most responsible moment to handle it without disruption can pass. Lack of feedback on the plan can aggravate the risk of this happening. Because of that, when the team feels that it is necessary, especially when dealing with new domains and platforms, it is important to perform architectural spikes to prove with code some assumptions on which the plan is based on.

* * *

When you decide to use a **Reference Architecture**, there are several things that the chosen reference should already handle, so you do not need to repeat it and document plans for problems that are already solved. When you **Find Where Hurts**, you need to **Plan Responsible Moments** for each technical challenge considered relevant.

//TODO: known uses

5. Tracer Bullets

Also Known as **Walking Skeleton**

Despite the architecture being based on a reference, there can be several low-level decisions that should be made, such as what frameworks should be used, how it should be instantiated and configured, and how the architectural layers are going to be organized and connected. These decisions are important to identify in order to define a standard to be followed during the implementation of the remaining system functionality, in order to achieve design consistency.

* * *

How can you define low-level details about the architecture without spending a lot of time upfront on a detailed investigation?

A high-level architecture defined by the reference usually does not contain enough detail to start the implementation.

There are several approaches to instantiate the same reference, and if a standard way to do that is not defined in the beginning of the project, several parallel approaches to do the same thing can be used on the software.

The integration between the reference and the solutions on the technical plans may be not be clear or trivial.

There are some assumptions in the technical plans that may need to be verified with initial implementations to eliminate some risks at the beginning of the project.

Once the iterations that deliver value to customers begin, there are more pressures on the team, and the time to explore different solutions becomes more limited.

Therefore:

Select the smallest set of architectural relevant user stories and implement them as references for upcoming functionality. Use this implementation to face technical challenges that were planned to be targeted before the project iterations.

These user stories should implement real user requirements and explore the basic architectural layers. During this implementation the frameworks can be introduced to provide functionality and the team can validate its suitability to the problem. During this implementation, if necessary, several options can be evaluated. After implementing the whole functionality, the code should be reviewed for refactoring opportunities, because it will be used as a reference and even a small problem here has a high potential to be propagated to the whole project.

Other user stories that that focus on architecture features need validation at the beginning of the project can also be included when you **Plan Responsible Moments**. These implementations should explore different possibilities and some design or architectural spikes can also be performed to fulfil this task. At the end of this implementation of these stories, the team should take time to evaluate the solution and make sure that “standard ways” of implementing architecture details are identified and understand by all.

The functionality developed by early **Tracer Bullets** should not be definitive. It should represent a viable start from where the application design may evolve. During the iterations, the design reference for the project may evolve, incorporate new ideas and solutions, and every implemented story should be developed in a way that it can be used as a reference for the next one.

A benefit of this pattern is that it provides a concrete way to design the basic application architecture, and, besides it is not the focus, the implemented story can still add value to the customers. It also provides a way to prove with code the core concepts of the architecture and solutions proposed to some technical challenges.

The main drawback of this pattern may happen when the developers start to struggle with some technical problems, which can take away the focus from business and delay the start of the iterations. The Tracing Bullet should not be considered an opportunity to develop complex frameworks and structures in the beginning of the project.

*   *   *

The **Tracer Bullet** can be considered a concrete and more low level definition of the reference chosen for the **Reference Architecture**. For some technical challenges, especially the ones with high risk and high impact, its validation by the **Tracer Bullets** can be the best way to **Plan Responsible Moments**.

This pattern is related to the practice **Prove with Code** (Agile Architecture 2015), but this pattern provides a concrete way to do that. This pattern is not new, and it is first description was made by Hunt & Thomas (1999). This is similar in some ways to a **Walking Skeleton** (Cockburn 2004) although you might implement **Tracer Bullets** as part of the core architecture rather than as a prototype.

//TODO known uses

6. Test Architecture

Agile projects have a huge focus on test automation, however to enable more low level testing, such as at unit or component level tests, the architecture should provide for testability. Sometimes there needs to be additional interfaces defined specifically to support testing. Even when it is decided to not test something, it should be a conscious decision by the team.

*   *   *

How can you define how architectural components should be tested?

It is desirable that the same test technique and approach is applied for similar architectural components.

Some kinds of components are hard to test and need experienced developers to develop a test automation solution.

You may also decide that some component is going to be tested manually or not tested at all.

When the tests of several similar components follow the same approach, it is easier to reuse code among them, making the next test easier to create.

The initial architecture solution for a component may be not suitable for test automation.

Therefore:

Define the test approach for each kind of component, considering its scope, technique, kind of test and tools that are going to be used.

Since the **Tracer Bullets** include the main architectural components, they can also be used as a reference to develop tests. Define the tests that should be developed for each system functionality, including its approach, the components that it covers, and the tools that are going to be used. These tests should be used as a reference for the tests of further functionality.

For instance, imagine a classic Java web application architecture that has an MVC architecture and uses DAOs for database access. The test architecture can define that DAOs are going to be unit tested by including the database, and by inserting data and verifying its state using the DBUnit testing framework. The business classes are also going to be unit tested, by creating a mock object for the DAO class. Finally, it is decided that the controller and the HTML interface, which has some JavaScript logic, will not be tested individually. So the team decides to create functional tests by using the test tool Selenium to test the whole system, including classes from the two layers that were not unit tested. The tests were developed for the User Registration user story, which was chosen to be the **Tracer Bullet**, and further used as reference for other functionalities.

Sometimes it is necessary to refactor the **Tracer Bullets** to improve testability and enable the creation of tests. Two valid approaches are to develop the tests after the **Tracer Bullet** or in parallel with it. Following this, the **Test Architecture** gives feedback to the architecture design. By applying this pattern, it is possible to address both how to build good tests for the architecture and how the architecture will affect how you test the system. This **Test Architecture** should evolve and be refined through the project as the system architecture evolves.

It is important to state that it is not everything need to have automated or unit tests. Some parts of the system may be tested using integration or functional tests. Even manual testing is a valid strategy, especially for components where it is difficult to automate, such as ones that interact with hardware. However, it is important to be aware of the consequences of such decisions.

A benefit to applying the **Test Architecture** pattern is that the team will have a reference for how tests should be developed for all system layers. This is especially useful to developers without much test automation experience. It will also standardize the approach for creating tests, which can help on test code reuse. A drawback is that the development of the **Test Architecture** can take precious time at the beginning of the project, and delay the start of the first iteration.

* * *

The reference used for the architecture when you **Climb on the Shoulders of Giants**, should be used as the base for the test automation. The **Tracer Bullets** can support the implementation of this pattern, by providing a implementation of the architecture used to validate the test implementation. The **Plan for Responsible Moments** can include tests to measure quality attributes, such as execution time or memory consumption, as reminders to trigger future tasks if necessary.

//TODO known uses

7. References

Agile Architecture: Strategies for Scaling Agile Development - Available at http://www.agilemodeling.com/essays/agileArchitecture.htm#ProveIt accessed on Jan 25 2015

Avgeriou, P., “Describing, instantiating and evaluating a reference architecture: A case study”, Enterprise Architecture Journal, June 2003.

Andrew Binstock. "In Search of Agile Architecture", posted on November 04, 2014, available on
http://www.drdobbs.com/architecture-and-design/in-search-of-agile-architecture/240169245

Alistair Cockburn. 2004. Crystal Clear a Human-Powered Methodology for Small Teams (First ed.). Addison-Wesley Professional.

Andrew Hunt, David Thomas. The Pragmatic Programmer: From Journeyman to Master, 1999, Addison-Wesley Professional.

Kent Beck, Extreme Programming Explained: Embrace Change, 1999, Addison-Wesley Professional.

E.B.Fernandez, "Building complex object-oriented systems with patterns and XP", Procs. of XP Universe (International Conference on Extreme Programming), Raleigh, NC, July 23-25, 2001.

Guerra, E., Mauricio, A., Yoder, J., Gerosa, M. "Preparing for a Test Driven Development Session", 20th Conference on Pattern Languages of Programs (PLoP), PLoP'14, September 14-17, Monticello, Illinois.

Java Community Process (JCP), JSR 316: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification, 2009, available at https://jcp.org/en/jsr/detail?id=316

PERILLO, J. and SILVA, J. and VARGA, R. and GUERRA, E. 2011. SADE – Sistema de Atendimento de Despacho de Emergências em Santa Catarina. In Proceedings of XIII Simpósio de Aplicações Operacionais em Áreas de Defesa (XIII SIGE).

Hongyu Pei Breivold, Daniel Sundmark, Peter Wallin, Stig Larsson. "What Does Research Say about Agile and Architecture?" The Fifth International Conference on Software Engineering Advances, ICSEA 2010, 22-27 August 2010, Nice, France

Obie Fernandez and Kevin Faustino. "The Rails 4 Way", 3rd Edition, Addison-Wesley Professional, 2014.

Waterman, M.; Noble, J.; Allan, G., "How Much Architecture? Reducing the Up-Front Effort," AGILE India (AGILE INDIA), 2012 , vol., no., pp.56,59, 17-19 Feb. 2012

Wirfs-Brock, R., "Agile Architecture Myths #2 Architecture Decisions Should Be Made At the Last Responsible Moment", 2011, Available at http://wirfs-brock.com/blog/2011/01/18/agile-architecture-myths-2-architecture-decisions-should-be-made-at-the-last-responsible-moment/

//more to be added