Patterns to Develop and Evolve Architecture During a Agile Software Project

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

Abstract: *The architecture design during an agile project is an activity that should take place in all phases of a project lifecycle. Even given an initial architectural for a project, it is important to continue to evolve the architecture in order to keep it suitable for the application's current needs. This paper documents some patterns for working on and continue to evolve the architecture while using agile techniques.*

1. Introduction

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

The editorial "In Search of Agile Architecture" (Binstock 2014) states that agile software development has gaps, and one of them is related to software architecture. According to this editorial, since architecture is important to enterprises, these gaps can prevent the usage of agile methods by some enterprises. Additionally, there was research performed by (Breivold et. al 2010) which looked for statements made regarding the relationship between agile development and software architecture. This research revealed that there is a lack of scientific support for many of the claims that are concerned with agile and architecture. In a grounded theory study involving 44 participants (Waterman et. al 2012), one of the findings noted that reducing the up-front design too much can lead to an accidental architecture that does not support the team's ability to develop functionality and fails to meet requirements. As a result, this work recommends that a team must find an appropriate trade-off between a full up-front architecture design and a totally emergent design. These conclusions are aligned with the patterns presented in this paper.

When architectural tasks can be predicted, it is important to include **Architecture in the Backlog**. However, when it is not possible to know if a feature will be needed in the architecture, an **Architectural Trigger** can help to determine a responsible moment to implement it. If the team is insecure about how something should work and how some architectural feature should be implemented, an **Architectural Spike** can be used to explore different options and search for the most appropriate solution. Since unexpected issues can

appear and some non-optimal solutions might be chosen for other reasons, **Technical Debt Management** is important to control what problems the architecture currently has, how this impacts the project, and how much interest the team needs to pay because of them.

The target audience for these patterns are software architects and software developers who work on projects with architectural challenges. We assume some familiarity with some terms from the agile software development community. Despite the patterns does not assume that Scrum is being used as a methodology, it borrows some terms from it, such as backlog, sprint, etc.

The goal of this paper is to present patterns identified for architectural design on agile projects that are appropriate throughout a project. These patterns should be considered to be used on projects where the reference architecture used is not enough to solve all architectural issues. For instance, in a simple Ruby on Rails project for information management these patterns might not be used since the reference framework provides solutions to the application structure and to handle its main issues. As another example, if this Ruby on Rails application needs to handle a very high amount of requests and deal with a huge set of databases, these patterns can help to manage the architectural evolution during the project to address these issues appropriately.

The paper is organized as follows: section 2 presents some previous existing patterns; from sections 3 to 6 the patterns are presented; and, finally, section 7 concludes the paper.

2. Existing Related Patterns

These patterns belong to a larger pattern collection the authors are working on which focuses on initial architectural design (Guerra et. al 2015). The following summarizes those patterns:

- **Climbing on the Shoulders of Giants**: How can you quickly define the basic application architecture and the main component types that will satisfy the requirements? Use an existing reference compatible with the application platform and suitable to its needs as a starting point.

- **Find Where it Hurts**: How can you identify relevant points where the architectural design should focus? Early on, identify the challenging technical requirements that are important for the project, so they can be handled at the optimal time.

- **Plan for Responsible Moments**: How can you handle the technical challenges in the beginning of the project without a full architectural design upfront? 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.

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

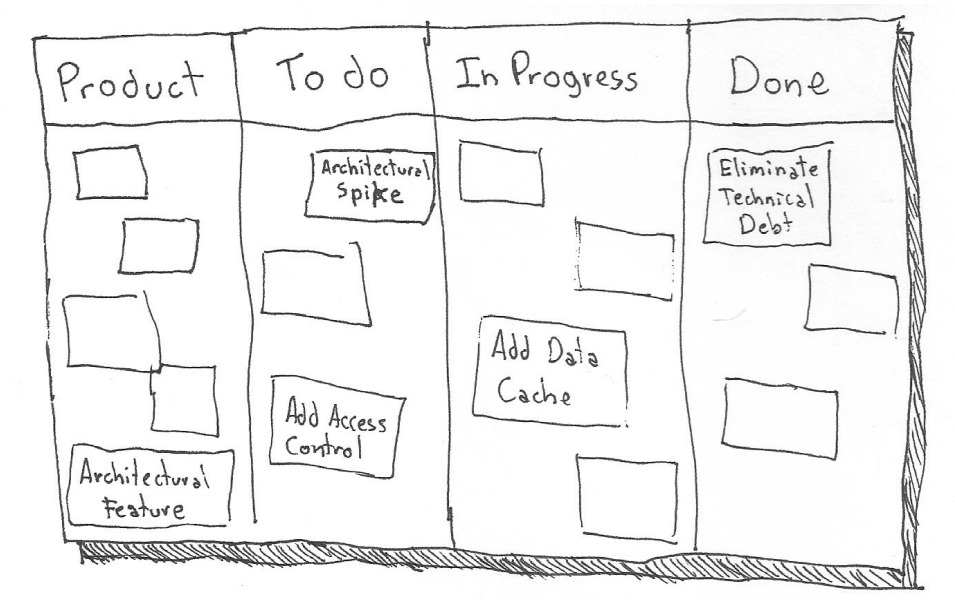
- **Test Architecture**: How can you define how architectural components should be tested? Define the test approach for each kind of component, considering its scope, technique, and kind of tests and tools that are going to be used.

Besides these patterns, the **Continuous Inspection** pattern [Merson et. al 2013], can be also considered to be part of this pattern collection. The following briefly describes this pattern in order to give the readers context:

- **Continuous Inspection:** How can you detect architecture and code problems as soon as possible? Use available automated tools to continuously inspect code, generate a report on the overall code health, and point out if any violation was detected. These tools can be executed locally on the developer's machine alone and by having the system communicate with a continuous integration server that builds the code at specific time intervals, or upon each code commit.

These patterns complement the patterns identified above by adding practices on how to evolve architecture during the development cycles. In the future, the authors intend to put these patterns together in a pattern language, however we chose to focus on the patterns individually in this paper.

3. Architecture in the Backlog



Agile projects usually start with architectural capabilities sufficient only to enable the development of features in the first few iterations. The focus is on delivering functionality with the assumption that emerging architecture will be sufficient. Often, the architecture needs to continue to be enhanced and expanded upon in order to meet continued software requirements.

How can you ensure that the architecture will adequately evolve during the project, meeting the necessary capabilities to fulfill the software requirements?

Work on agile projects is scheduled based upon a priority backlog of work items. Often tasks related to architecture evolution are deferred, given low priority, if any at all. This is especially true when the backlog is composed primarily of functional user stories.

Defining a minimum architecture at the beginning of the project is not sufficient to support all the architecture capabilities you will eventually need. Refactoring in order to improve the design may not improve the architecture. Good architecture doesn't just magically emerge, it requires attention.

Often an architecture capability supports multiple functional user stories. It can be difficult to estimate the effort to implement a particular user story when changes are required to the architecture. When changes to the architecture are buried in the detailed tasks for a particular user story, they can become invisible. But they still need to be done.

On the other hand, if the team focuses only on developing architectural features in the first few iterations, there can be a temptation to add complexity to the architecture that might not be needed thus leading to overdesign. It is import to control the amount of effort dedicated to the

architecture and to balance implementing architecture capabilities with implementing the functionality that depends them.

Therefore:

Add important architectural capabilities to the backlog as they become apparent to make sure that they are prioritized and implemented at the *Most Responsible Moments*.

There are different approaches for adding tasks related to the architecture to a backlog. They can be added as independent tasks to the existing backlog, associated with a specific user stories as acceptance criteria, or even maintained as separate tasks in a separate architectural backlog that supports a functional user story backlog.

Associating an architectural requirement directly with a User Story is advisable when the product owner is not able to understand the importance of an architectural task or schedule them separately or when it is specific to that story. Instead of having to prioritize a technical task, the product owner can evaluate the value of an architectural requirement based on a story that it impacts. However, some tasks related to the architecture can be hard to associate directly with any specific User Story since they may impact many different stories, for example overall system performance. In the case where they affect the overall system, it is a good idea to create a separate architectural story to describe the specific needs of the system.

Having a separate architectural backlog is another approach to ensure that there is always part of the effort focused on architecture evolution. By using this approach you can define how much effort should be dedicated to each of them. When the project has multiple teams, there can be a separate team dedicated to working on the architecture backlog in support of other teams who are working on system functionality. The advantage of having a separate architecture backlog is that it is easier to prioritize and manage the amount of effort dedicated to architectural tasks in each iteration.

It is important is to know what should be implemented in the architecture and be sure that it is implemented according to understood priorities. The technical plan to implement architectural requirements can explicitly define tasks to be performed during the iterations. These tasks are usually related to architectural capabilities that can be more easily predicted and worked on as discrete development tasks, such as security control or an integration mechanism.

* * *

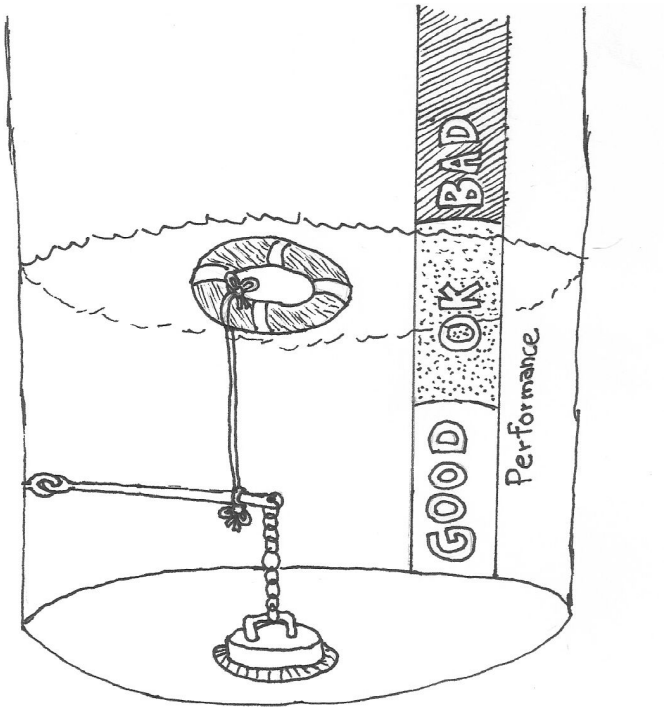
When the team **Plans for Responsible Moments** some tasks related to the architecture evolution will be identified that can be added to the backlog.

It is important to highlight that new architectural requirements and controls can appear due to an **Architectural Trigger** or the **Continuous Inspection**. In this case, they should be entered into the backlog and be considered for subsequent iterations.

The usage of a dedicated architectural backlog was reported in Madison (2010) and Medtronic Experience Report (Agile 2006?). A report from Adobe (Green 2013) states that they consider architectural layers to divide the tasks from a user story.

On a project hold on National Institute of Space Research in Brazil, the project LEONA, each story on the backlog was broken on functional and non-functional tasks. When choosing the tasks for a given iteration, the team paid attention to always add some amount of tasks related to architectural features, not too little and not too much.

4. Architectural Trigger



During development, new architectural capabilities are sometimes needed for specific features. For instance, it can be necessary to implement some memory optimization due to performance degradation. Or a refactoring introducing a design pattern might be needed when the functionality in a single class becomes too complex. These architectural tasks are necessary to evolve a system's structure, however it is not always clear what they are or when they will be required.

How can the team know when current conditions should cause work on the architecture?

Premature optimization (Knuth 1974) or premature abstraction (Ward's Wiki) can generate unnecessary work and even add complexity to future tasks. Working on improving some aspect of your design might not even have the improvements you expect. However, you want to work on keeping the architecture in line with emerging requirements.

Doing too much to prevent architectural degradation can result in overdesign or can overly restrict acceptable design solutions. You want to focus on implementing user functionality while at the same time adequately evolving your architecture. It can be difficult to predict when you need to shift emphasis to working on improving your architecture.

Not all quality attributes that demand attention from the architectural point of view are easily measured and monitored automatically. Because of that, sometimes it is not possible to have a fast feedback on whether the architecture if fulfilling some requirement.

It takes time to plan for architectural issues and how to react when certain conditions arise. Being aware of architectural considerations is important, however you don't want to waste time worrying over things that might not ever happen.

Therefore:

Define conditions or scenarios in the architecture that if they happen can trigger the addition of one or more architectural tasks to the backlog.

This trigger can be based on criteria established in an **Agile Landing Zone** (Yoder and Wirfs-Brock 2014) for conditions based on quality attributes, or by **Continuous Inspection** tools for code quality conditions. **Agile Landing Zones** can reveal a degradation to the architecture where essential quality attributes no longer meet acceptable minimal values. In fact an architecture trigger should be established for whenever a landing zone attribute drops below the minimum. The process of **Continuous Inspection** can reveal parts of the architecture with deficiencies that need attention. It is impossible to predict when these conditions will appear. Whether or not you decide to work on the architecture, however, may depend on your diagnosis of why the condition was triggered.

There are different ways to monitor these conditions. For example a team might use a **System Quality Dashboard** or a **System Quality Radiator** to get regular feedback of important system qualities (Yoder and Wirfs-Brock 2014) and be notified when a problem is detected. When triggering conditions occur, then you will need to determine whether there is an easy fix or whether you need identify relevant architectural tasks and prioritize them in the backlog. The technical plan may prescribe options that can be taken in case the trigger is activated and define the tasks that should be added to the backlog.

Some conditions can be hard to quantity, such as "the logging is not good enough". However there should be an agreement by the team on how to determine when these hard to quantify conditions should trigger an architectural task. For instance, a discussion between developers can determine that when an unacceptable number of complexity or code smells are present, they will consider to refactor the code. This trigger can be registered and discussed by the team when someone believes that a triggering condition has been reached.

The architectural trigger should be defined for characteristics that affect the system as a whole and not a single class or isolated function. For instance, if one class is growing too big, that can be considered a design problem. However if this is happening to a set of related classes in the same layer, then it might be considered an architectural issue because it might affect the design integrity of the entire layer. That distinction is important because the introduction of a new cross-cutting architectural feature has a higher impact on the system as a whole than a change to an isolated set of classes.

The actions to be performed when the trigger conditions are met are often hard to be precisely defined. If it is clear that some specific refactoring should be performed, that task can be added directly to the backlog. However, when the path to follow is not clear, the team can use an **Architectural Spike** to find the better solution.

An example of context where the trigger can be used is for performance requirements. A measurement can be performed frequently to verify if a performance attribute is inside the respective **Agile Landing Zone**. If it is found that the measurement is below the range of the values represented in the landing zone, a task to improve the performance should be added in the backlog and considered for the next iteration.

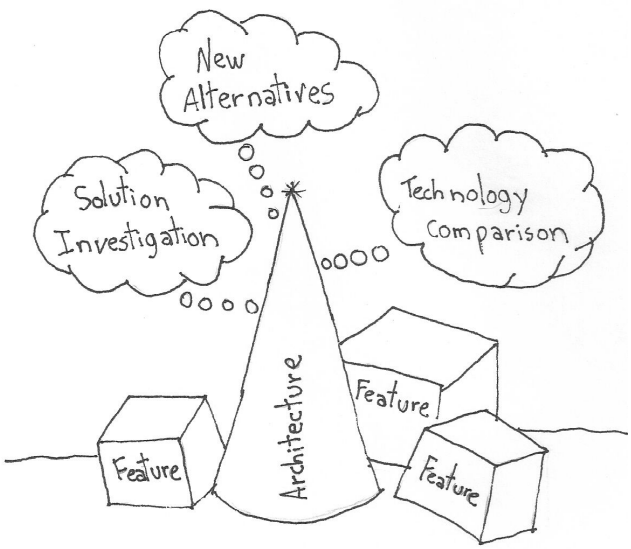
* * *

The identification of the architecture trigger can be part of the **Plan for Responsible Moments** defined at the beginning of the project. In response to the trigger, an associated action should be added representing **Architecture in the Backlog**.

In the project SADE, for an emergency response system (Perillo et. al 2011), an architectural trigger was developed for the reverse AJAX mechanism. If measurements reported that it was not fulfilling the performance requirements, some architectural task should be done to optimize it. Despite defining this trigger, the initial solution was enough for the requirements and no changes needed to be made.

Some platforms, such as Mezuro (Meirelles 2013) and SonarQ (Bellingard 2011) provide an environment that can be used to send notifications to development team according to configuration criterias based on the source code metrics. If the metrics value exceeds a configured threshold, the team is notified that they should do something about it. These can be considered architectural triggers for code quality.

5. Architectural Spike



Architectural decisions are not always easily made. For example, the seemingly simple decision to add a component to the architecture might demand tests to verify if it is in conformance with the quality attributes requirements. Dealing with the issues takes time and the team might not have expertise in some technology needed for some part of the system. They may need to gain experience before they can confidently proceed. Also, it may not be clear which architectural decision will best meet the current needs. Consequently, some investigation might be needed before any decision is made.

How can you explore and decide on architectural issues that are not completely understood by the team?

Any architecture decision involves making tradeoffs. You need to understand and analyze tradeoffs in order to make a decision on which solution best meets the goals of the product. This may require some experiments or investigation.

Some architectural tasks are difficult to understand and to estimate. Trying to force estimates on poorly understand tasks leads to inaccurate guesses. Expectations generated by these imprecise estimates can lead to false assumptions about what progress can be made and at what cost.

It can be hard to estimate based on a technologies or components that are not completely understood by the team. Even if you have information about similar efforts from other teams, you can still be misled. Your project might use a technology or component differently.

Not all decisions related to the architecture are made at the beginning of a project. When these later decisions are made the architecture has evolved quite a bit and it can be hard to know what the best solution might be and how it will best fit and affect the current architecture.

Some features that are on the project roadmap may or may not be feasible. Sometimes experimenting with different architecture solutions in support of a new feature will result in adjusting the roadmap.

Therefore:

When more information is needed, add an Architecture Spike task to your backlog to perform some study, test or alternatives investigation that support architectural decisions.

Determine a number hours that can be dedicated to that **Architectural Spike** and halt that task when that limit is reached. The time dedicated to any architecture spike needs to be bounded. If necessary, because you haven't made enough progress towards making a decision, continue working on the spike for another bounded period of time. Even when the Architectural Spike does not result in a final solution, there can be some advances, such as the identification of new alternatives or the elimination of alternatives. When the path to follow is clear, a **Plan for Responsible Moments** can be made to focus on the implementation of that architectural feature.

An important consideration for **Architectural Spikes** is to know when to work on them. When a need is perceived, add an **Architectural Spike** to the backlog. When the prioritization for the next iteration is performed, It is important to observe further tasks that depend on the respective architectural decision and the impact of delaying that decision according to the **Technical Debt Management**.

The main goal of an **Architectural Spike** is to provide information for an architectural decision. Based on that goal, there are differents tasks that might be performed depending on what information is needed. If the spike aims to confirm that a technology is suitable for the requirements, some tests or even a prototype might be developed and evaluated. Another possible question to be answered by the spike is what technology is more suitable. In this situation, some metrics and criteria should be defined based on the requirements, and the different options should be explored. Finally, the spike can also search for a solution to a problem that the team has no clue how to solve. In this case, the spike can be used to research alternative solutions, raise the positive and negative points of each, and assess which possible solutions are best for the project.

The difference between an architecture spike and a design spike is that the architecture spike involves decisions that are potentially difficult or costly to reverse, while a design spike happens when someone does not know how best to implement some functionality. For example, deciding

how to refactor a large, complicated conditional logic or whether the use of the interpreter pattern help solve a problem is a design spike. Choosing whether to buy or build a framework is an architectural spike. Similarly, how to improve application performance is likely an architectural spike.

The result of an architectural spike can be a solution, however it is not mandatory. It can just produce information that can be used to make decisions. For instance, a spike can reveal that a solution alternative is not viable or suitable to the problem.

One consequence of an **Architectural Spike** is that all tasks that depend on its architectural decision need to wait. The lack of any guaranteed solution may not be acceptable, depending on the criticality or importance of tasks that depend on it. A chain of **Architectural Spikes** in consecutive iterations that does not yield a viable solution can negatively impact progress. In this case, the problem may be tackled with a more dedicated effort, such as asking a specialist for help or performing an iteration with the participation of all team dedicated to solve that specific architecture problem.

* * *

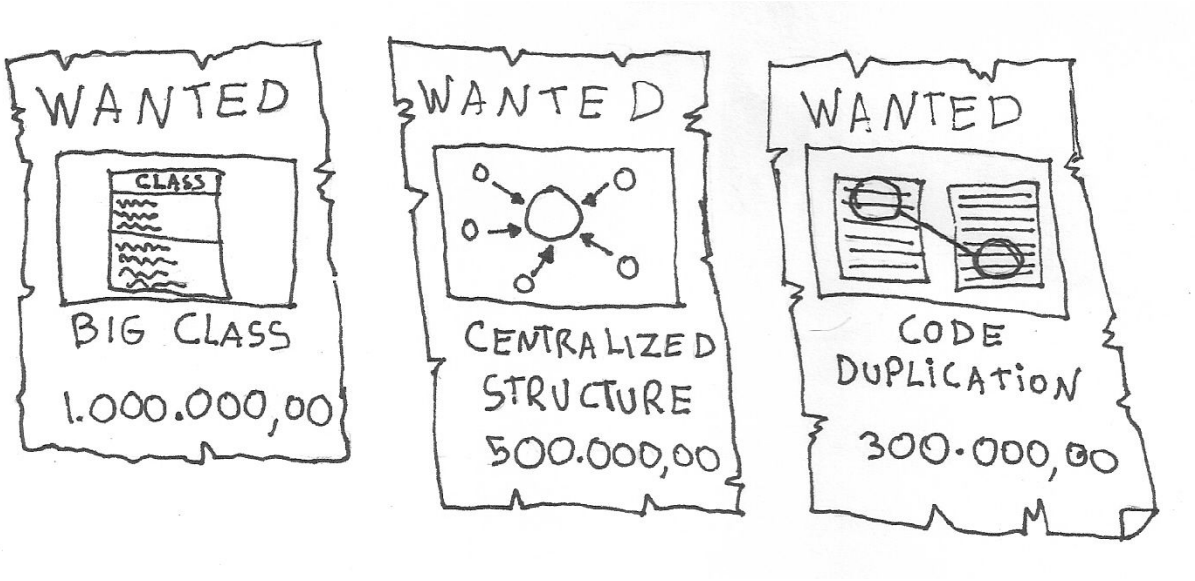
Plan for Responsible Moments can use **Architectural Spikes** to address uncertain architectural requirements. **Architectural Spikes** can be included in the plan and included as part of the **Architecture in the Backlog**.

*On a project at the National Institute of Space Research in Brazil, project LEONA, an **Architecture Spike** was used for several tasks that the team didn't know how to perform. Examples of such tasks were: to verify how an applet should be configured to present a video streaming, and to choose an acceptable protocol for video streaming.*

*The Refactory used **Architectural Spikes** for an Invoicing and Order Processing system which involved building the generic processing of the imports of data for various clients. Many of the clients had similar types of orders with minor differences which led to a lot of duplication. A spike was used to create a framework for these order imports.*

***Architecture Spikes** were used for building the infrastructure of medical rules and the persistent layer for systems used by the Illinois Department of Public Health (IDPH) system. In the IDPH systems the original persistent layer was cumbersome, difficult to use, and error prone, thus a spike was used to investigate better ways to handle the persistence layer. This led to the development of a persistence framework, resulting in a persistence mechanism that was easier to use and less error prone.*

6. Technical Debt Management



Sometimes, when an **Architectural Trigger** demands some action by the team, it is not a priority to handle it. In others cases, some unanticipated issues related to the architecture or the application structure arises in the middle of the project. These problems can grow over time, if not addressed, thus increasing the cost to execute other tasks and to fix that issue, which is represented by a metaphor called “technical debt” (Cunningham 1992).

Technical debt is not a simple problem, but a problem whose cost to resolve increases over time. This increase can be related to the amount of work for solving that problem or to the extra effort required to perform other tasks that depend on debt-ridden code. It is important to keep in mind that fixing problems in the future will be more expensive than fixing them today (Zazworka et. al 2011).

Some technical debt is intentional (Yli-Huumo et. al 2014). It might even be forgotten if it is not managed. For example, if you decide to cut corners on an implementation in order to meet deadlines, there can be inconsistencies in how you implement similar functionality. You can also discover after implementing some functionality that there is a better way to do it. Inconsistencies and inadequate design choices can become overwhelming across development cycles if not addressed.

How the team can avoid an uncontrolled growth of technical debt?

Some debt happens unintentionally through misunderstanding or errors in judgement. Even the abstract design used by a team, such as a reference architecture, may induce some debt on the system.

A little technical debt is inevitable on any project. There isn't time to always improve code that "works well enough." On the other hand, if you do not work towards a **Sustainable Architecture** (Wirfs-Brock and Yoder, 2012), this debt can grow and cause your system to devolve in a poorly architected **Big Ball of Mud** (Foote and Yoder, 1998).

It is often hard to know what the right decision is for handling technical debt. Some debt is easy to fix and urgent, other problems are important to address, but occur infrequently. Some debt may be costly and hard to fix. Some debt may be easy to remedy but not that important. Prioritizing debt reduction efforts can be difficult.

The consequences of a particular design choice and its its accumulated effects can be difficult to understand. Deferring decisions until after you have learned enough can have a huge payoff.

Some debt does not cause problems as that part of the system is fairly stable and isn't changing much. Trying to pay off the debt may be more costly than any benefit that comes from reducing it. However, this situation may change in the future, and it can subsequently become important to pay off the debt.

Some short term debt can be good for tactical reasons. It may be more valuable to release the software and get feedback on new features rather than clean up some existing technical debt. Sometimes long term debt can be taken on for strategic reasons or proactively for business reasons.

Therefore:

Identify and manage the technical debt present in the project along with the respective effort to fix and the interest that accrues if the debt is not paid off.

How to address technical debt should consider the effort to fix it, the consequences the debt has the current users of its system and continued development, and how much that debt will grow if not fixed. A recent study revealed that the management of existing debt can help to avoid its uncontrolled growth (Torin 2014). New debt items can be added to the list at any time if the team agrees that it is relevant.

Continuous Inspection can be used to examine debt and monitor whether it has an impact. If the team decide to not handle the debt, the inspection tools can be used to verify how much is it growing. This can lead to an **Architectural Trigger** for human intervention to analyze and decide if there is some technical debt that should be managed.

When managing technical debt, the most critical problems should be prioritized first. Selecting which debt is most critical should be based on what impacts it has on current users and your ability to sustain ongoing development. For example, it may be more critical to fix a performance

issue that is slowing down every user than it is to re-implement a service to use the standard logging mechanisms.

It is important to make the list of debts public and accessible to the team, so everyone can be aware of the existing problems and their consequences over the short and longer term. When these items need to be addressed, they can be added to the backlog which might include **Architecture in the Backlog**. It is also important to highlight the ongoing impact of the debt, and consequently its priority, which may vary over time and need to be reevaluated.

There are different approaches to handle technical debt. Refactor the system to eliminate the debt is the most definitive approach, however depending on scope of the effort, it can require lot of investment that might not be worth the effort. Another option is to isolate the problem in a way so it has minimal implementing the rest of the system, implementing patterns such as **Sweeping it Under the Rug** (Foote and Yoder, 1998), **Wiping Your Feet at The Door** (Wirfs-Brock and Yoder, 2012) and **Anti-Corruption Layer** (Evans 2003). For debt related to quality attributes, new architectural controls and components might be introduced to solve the problem and to avoid it accumulating.

Technical Debt Management takes time and requires discipline from the team to maintain the technical debt list. The introduction of the tasks related to debt management into the usual team meetings, like fast updates into stand up meetings and their discussion at planning meetings, can help the team to incorporate debt management into their routine.

* * *

When the team does not know how to handle a technical debt, it can use an **Architectural Spike** to explore the different possible solutions. An **Architectural Trigger** can be used to identify when a technical debt is becoming a problem that needs to be handled.

On Pires et. al (2014), a study conducted on three teams from a large company revealed that visibility of problems and their consequences was increased by explicitly managing the technical debt.

7. Conclusions

This paper presents four patterns for architectural design that can be used in the context of agile projects that need to handle complex architectural requirements. These patterns focus on practices to be used during project development. Although these patterns have many relations to other patterns, our collection of patterns is not yet a pattern language. Future efforts will further investigate the relationship among these practices, and explore whether there is a pattern language for evolving architectures during an agile project.

Acknowledgements

We acknowledge the support of CNPq (grant 445562/2014-5) and FAPESP (grant 2015/16487-1). We also thank a lot our shepherd, Stefan Sobernig, for given a valuable feedback during the shepherding period.

References

Bellingard, Fabrice. "Effective Code Review with Sonar", posted on October 20, 2011, available at http://www.sonarqube.org/effective-code-review-with-sonar/

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

Eduardo Guerra, Rebecca Wirfs-Brock, Joseph Yoder. Patterns for Initial Architectural Design on Agile Projects, AsianPLoP, 2015.

Green, Peter. "Splitting stories into small, vertical slices.", September 27, 2013, available at http://blogs.adobe.com/agile/2013/09/27/splitting-stories-into-small-vertical-slices/

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

Paulo Merson, Joseph Yoder, Eduardo Guerra, Ademar Aguiar. Continuous Inspection - A Pattern for Keeping your Code Healthy and Aligned to the Architecture, AsianPLoP, 2013.

PERILLO, R. ; SILVA, J. R. B. ; VARGA, R. ; GUERRA, E. M. . SADE Sistema de Atendimento de Despacho de Emergências em Santa Catarina. In: XIII Simpósio de Aplicações Operacionais em Áreas de Defesa, 2011, São José dos Campos

Meirelles, P. "Monitoramento de métricas de código-fonte em projetos de software livre", PhD thesis, Universidade de São Paulo, USP, Brasil, 2013.

Madison, J., "Agile Architecture Interactions," *Software, IEEE* , vol.27, no.2, pp.41,48, March-April 2010

Donald E. Knuth. 1974. Computer programming as an art. Commun. ACM 17, 12 (December 1974), 667-673.

Waterman, M.; Noble, J.; Allan, G., "How Much Architecture? A Grounded Theory of Agile Architecture" AGILE India (AGILE INDIA), 2012 , vol., no., pp.56,59, 17-19 Feb. 2012

Joseph W. Yoder and Rebecca Wirfs-Brock, "QA to AQ Part Two Shifting from Quality Assurance to Agile Quality - Measuring and Monitoring Quality", PLoP'2014, September 14-17, Allerton, Illinois USA

Cunningham, W., Dec. 1992. The wycash portfolio management system. SIGPLAN OOPS Mess. 4 (2), 29–30.

Yli-Huumo, J., Maglyas, A., Smolander, K., 2014. The sources and approaches to management of technical debt: A case study of two product lines in a middle-size Finnish software company. In: Jedlitschka, A., Kuva ja, P., Kuhrmann, M., Mnnist, T., Mnch, J., Raatikainen, M. (Eds.), Product-Focused Software Process Improvement. Vol. 8892 of Lecture Notes in Computer Science. Springer International Publishing, pp. 93–107.

Zazworka, N., Shaw, M. A., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt. MTD '11. ACM, New York, NY, USA, pp. 17–23.

Rebecca Wirfs-Brock and Joseph Yoder, "Patterns for Sustaining Muddy Architectures", PLoP'2012, October 19-21, Tucson, Arizona, USA

Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

Brian Foote and Joseph W. Yoder Big Ball of Mud Fourth Conference on Patterns Languages of Programs (PLoP '97) Monticello, Illinois, September 1997. Technical report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997. Pattern Languages of Programs Design 4 edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison Wesley, 2000