Patterns to Develop and Evolve Architecture During an Agile Software Project

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

The design of the architecture during an agile project is an ongoing activity that takes place in all phases of a project lifecycle. It is important to continue to evolve the architecture in order to keep it suitable for the software system's current needs. This paper documents four patterns for working on and evolving a system's architecture using agile techniques.

Categories and Subject Descriptors
• **Software and its engineering~Agile software development**
• Software and its engineering~Software design techniques • Software and its engineering~Risk management
• Software and its engineering~Software evolution • Software and its engineering~Patterns

General Terms
Agile, Architecture, Patterns, Software Qualities, Agile Methodology

Additional Keywords and Phrases
Agile Architecture, Backlog, Technical Backlog, System Qualities, Patterns, Agile Software Development, Technical Debt, Software Evolution, Architectural Spike, Design Spike, Refactoring

ACM Reference Format:
Wirfs-Brock, R., Yoder, J.W., and Guerra, E. 2015. Patterns to Develop and Evolve Architecture During an Agile Software Project. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 18 pages.

1. Introduction
Agile teams generally don't follow a common set of architectural design practices as evidenced by industry reports [Bin], a systematic review [BSWL], and a grounded theory study [WNA]. Research [BSWL] into the relationship between agile development and software architecture reveals a lack of empirical evidence for many of the claims about agile processes and architecture. In the grounded theory study involving 44 participants [WNA], one of the findings was that reducing up-front design too much can lead to an accidental architecture which does not necessarily support the team's ability to develop functionality and fails to meet requirements.

Author's email address: joe@refactory.com, rebecca@wirfs-brock.com, guerraem@gmail.com

More recent agile methods such as SAFe [Lef] or Disciplined Agile Delivery [AL] address agile at scale. They recommend several architecture practices which have been adopted by some larger organizations. But there still is a lack of consensus around agile architecture practices. A question to be answered on agile projects is how much architecture definition is needed to start development. When the project is running, the challenge is to keep the architecture good enough to support the current features under development and capable of evolving as needed. Gaps or inconsistent agile architecture practices can lead to a lot of problems, such as technical debt or the inability to sustain ongoing development. Agile teams need to find an appropriate balance between full up-front architecture design and totally emergent design.

This paper presents four patterns for architectural design on agile projects: **Architecture in the Backlog, Architectural Trigger, Architectural Spike,** and **Technical Debt Management**. When architectural tasks are known, include **Architecture in the Backlog**. However, when it is uncertain whether a new architecture capability will be needed, an **Architectural Trigger** can help to determine a responsible moment to analyze the situation and determine what actions should be taken. If the team is uncertain how something should work or how some architectural feature should be implemented, an **Architectural Spike** can explore different options and search for the appropriate solution. Since architecture shortcomings can be discovered throughout development and non-optimal architectural solutions might need rework, **Technical Debt Management** helps clarify problems and deficiencies with the current architecture, how these weaknesses impact ongoing development, and costs associated with either reducing technical flaws or deciding to live with them.

These patterns should be considered for use on projects where there is significant architecture and design work required. Although the target audience for these patterns are primarily software architects and developers who work on agile projects, these patterns may be useful to a broader audience of architects working on non-agile projects. Another target audience for these patterns are product owners and project managers that need to understand when and how architecture decisions can be made, and how architecture evolution can be managed throughout the project lifecycle. We assume familiarity with some terms from the agile software development community.

The paper is organized as follows: section 2 presents related patterns that are relevant to this current work; sections 3 to 6 presents our patterns; and, section 7 summarizes our work and identifies some additional patterns.

2. Related Patterns

These patterns belong to a larger pattern collection the authors are writing which focuses on architectural design practices for agile projects [WYG]. The following is a summary of patterns previously documented that focused on architectural practices to be performed in the beginning of an agile project:

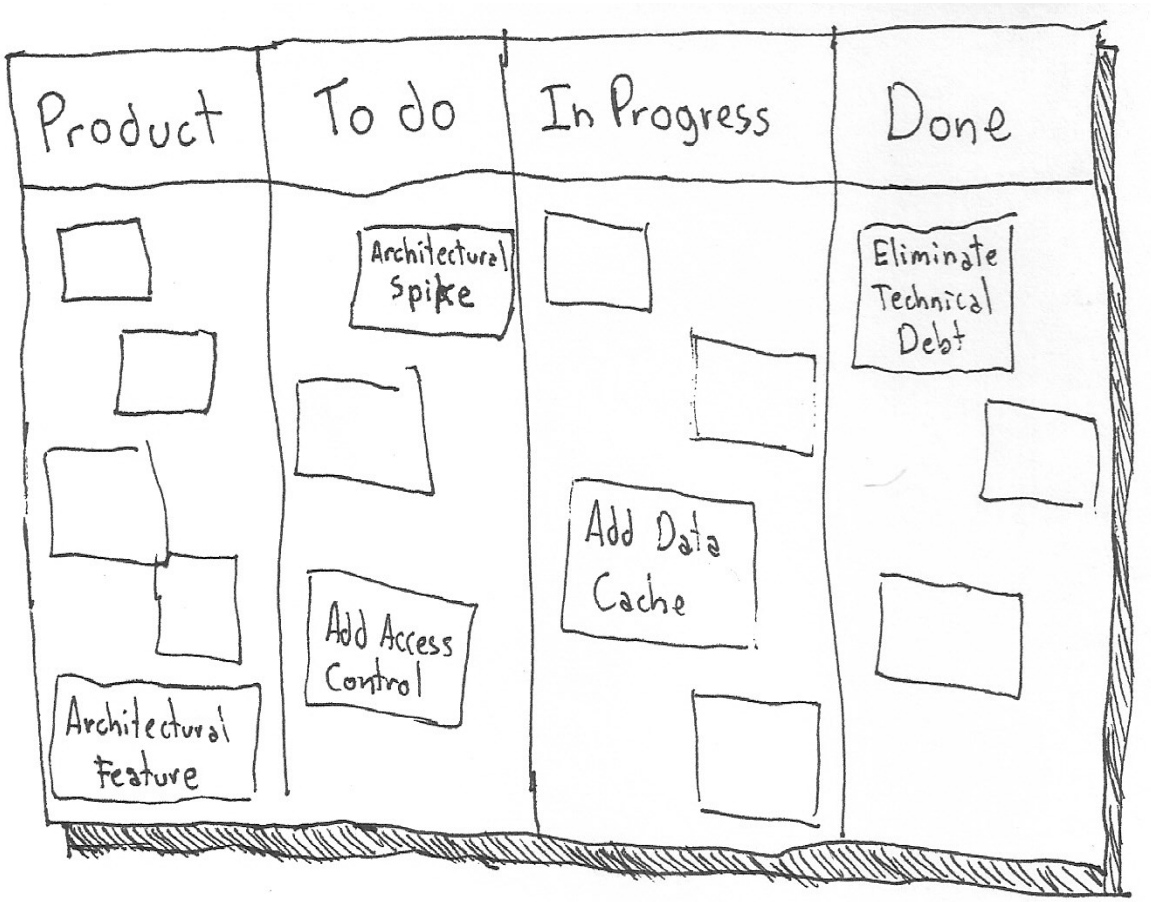
- **Climbing on the Shoulders of Giants**: Use an existing reference compatible with the application platform and suitable to its needs as a starting point.
- **Find Where it Hurts**: Early on, identify the challenging technical requirements that are important for the project, so they can be handled at the optimal time.
- **Plan for Responsible Moments**: Create a technical plan for how and when to handle each of the technical challenges and evolve that throughout the project. Such a plan needs to define how to identify important responsible moments and circumstances when it is appropriate to address technical challenges and work on the architecture.
- **Tracer Bullets**: Select the smallest set of architecturally relevant user stories and implement them as a reference implementation to learn from and for upcoming functionality. Use this implementation as an example solution for technical challenges targeted for successive iterations.
- **Test Architecture**: 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, **Continuous Inspection** [MYGA], can be also considered part of this pattern collection:

- **Continuous Inspection:** 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 or by having the system communicate with a continuous integration server that builds the code at specific time intervals, or upon each code commit.

Our patterns complement these patterns by addressing how to evolve and sustain the architecture during development. In the future, we intend to put these patterns together into a pattern language.

3. Architecture in the Backlog



Agile projects usually start by implementing architectural features that support development of backlog items that will be delivered in the first few iterations. The initial focus is on delivering functionality. Most likely the architecture will need to change in order to meet evolving software requirements. Even with legacy software, it can be useful to consider activities for improving or shoring up the existing architecture as you incorporate new functionality.

How can the architecture adequately evolve to best meet the changing stream of software requirements?

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

Defining a minimum architecture at the beginning of the project is insufficient 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 magically emerge, it requires ongoing attention.

It can be difficult to estimate the effort to implement a User Story when significant changes are required to the architecture. When changes to the architecture are buried in the details of User Stories, they can be overlooked or slighted. But they still need to be done.

If the team implements architectural features in the first few iterations which are not immediately required by any User Story, the architecture might become overly complex. It is import to balance implementing architecture capabilities with the functionality that depends on them.

There can be a temptation to make big changes to the architecture all at once. This is risky and can undermine system stability. Sometimes the existing architecture is “good enough.” It is difficult to decide how much or how little to change, and when to make those changes. To keep the architecture as it currently is, even when it could be improved, is a valid design decision.

Therefore:

Add new architectural capabilities to the backlog to ensure that they are prioritized and implemented at the *Most Responsible Moments*.

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

There are different approaches for adding architecture work items to a backlog. They can be added as independent items, associated with a specific User Stories as acceptance criteria, or even maintained in a separate architectural backlog that supports a functional User Story backlog. Each approach has its benefits and drawbacks.

Associating an architectural requirement directly with a User Story is advisable when the architecture work is specific to that story. If the Product Owner does not understand the significance of an architectural task, it may better to define appropriate acceptance criteria for the story that include architecture-specific quality requirements. Instead of having to prioritize a separate architectural work item, the Product Owner can evaluate the value of User Story which will also include an architectural requirement.

An architecture feature often supports multiple User Stories. For example the architecture feature to develop an overall caching strategy can impact many User Stories. When the architecture work affects the system as a whole, it is a good idea to create a separate architectural story to describe the specific architecture requirements.

Having a separate architectural backlog is one way to ensure that there is always part of the effort focused on architecture evolution. By maintaining and prioritizing this architecture backlog,

you can define and prioritize how much effort should be dedicated to architecture tasks in each iteration. Another way to make architecture tasks stand out is to tag the backlog items related to architecture or technical debt reduction with different colors [Kru].

When the project has multiple teams, a separate team might be dedicated to working on the architecture backlog in support of other teams working on system functionality. However, maintaining a separate architecture backlog has some drawbacks. It is harder to see the dependences between the User Stories and their related architectural tasks. Regardless, the prioritization of User Stories and architectural tasks needs to be coordinated.

* * *

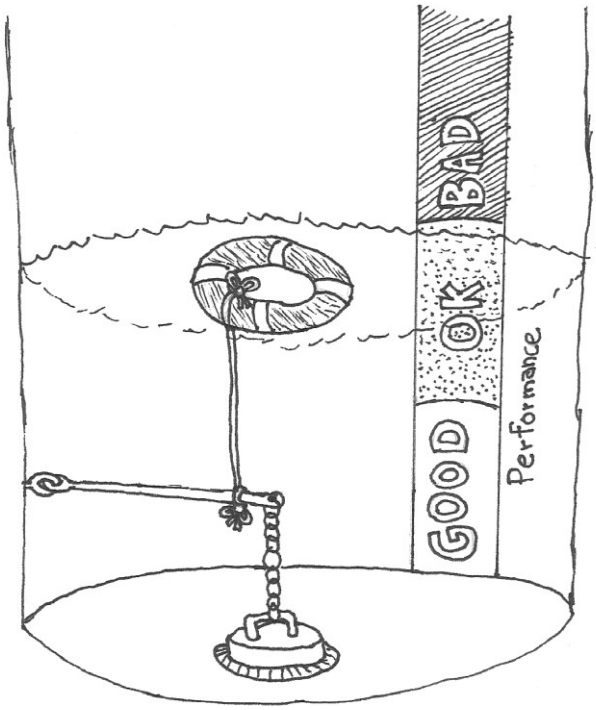
When the team **Plans for Responsible Moments** some tasks related to architecture evolution may be identified and added to the backlog.

New architectural requirements and controls that appear due to an **Architectural Trigger** or the **Continuous Inspection** will be added to the backlog and considered for subsequent iterations.

A dedicated architectural backlog was reported in [Mad] and a Medtronic Experience Report [Spe]. A report from Adobe [Gre] states that they used architectural layers to divide separate architecture tasks from User Stories.

On a project at the National Institute of Space Research in Brazil, project LEONA, each story on the backlog was broken into functional and non-functional tasks. When choosing 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



Agile teams value responding to change over following a plan. It is common to evolve and refine your architecture during development, but the appropriate time to make significant architecture changes isn't always clear. For instance, it can be necessary to implement memory optimization due to performance degradation. Or, refactoring part of the system to introduce a new service or threading model could cause some architectural rework. However, it is not always clear what will lead you to revisit your architecture or when changes will be required.

How can the team know when to rework or evolve the architecture?

Premature optimization [Knu] and premature abstraction can generate unnecessary work and add complexity to future tasks. Attempting to improve some aspect of your design might not have the payoff you expected. However, it is important to keep the architecture in synch with and supportive of emerging requirements.

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 from implementing functionality to working on improving your architecture. Being aware of architectural considerations is important, however you don't want to waste time worrying over things that might not be relevant.

If certain architectural issues are not addressed in a timely manner, it can be expensive to rework the architecture. Not addressing the architecture regularly can lead to an architecture that is hard to change and to validate.

Therefore:

Define conditions that trigger architectural investigations which may lead to adding tasks to the backlog.

The triggers can be based on when values of certain system qualities are not being met. These triggers might be based on criteria established in an **Agile Landing Zone** [YW], or by **Continuous Inspection** tools for code quality conditions, or conditions defined by the team.

Agile Landing Zones define acceptable minimal values for system quality attributes. An architecture trigger should be defined for when a landing zone attribute drops below the minimum. The process of **Continuous Inspection** can also reveal parts of the architecture that need attention. It is impossible to predict when these triggers will appear. Whether or not you decide to work on the architecture, however, depends on your diagnosis of why the condition was triggered.

There are different ways to monitor trigger conditions. For example, a team might use a **System Quality Dashboard** or a **System Quality Radiator** to get regular feedback about important system qualities and be notified when a problem is detected [YW]. When a triggering condition occurs, the team will need to determine whether there is an easy fix or whether you need to define 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 triggering conditions can be hard to quantity. 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 refactoring the code. This trigger can be noted 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 component, class, or isolated function. For instance, if one class is growing too big, that should be considered a design problem. However if a set of related classes in the same layer is growing too big, then it might be considered an architectural issue because it can affect the design integrity of the entire layer. This distinction is important because the introduction of a new cross-cutting architectural feature has a higher impact on the system as a whole than does a change to an isolated class.

The actions to be performed when the trigger conditions are met are often hard to precisely define. It may be clear that some specific refactoring should be performed, but the scope of that task may require some investigation before it is added to the backlog. When the path to follow is not clear, the team can use an **Architectural Spike** to investigate appropriate actions to take.

For example, 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 minimum acceptable value, a task to improve the performance should be added in the backlog and considered for the next iteration. You can also define easily implemented micro-benchmarks to measure the performance of simple, common system behaviors [Pro] and trigger further investigation when performance degrades.

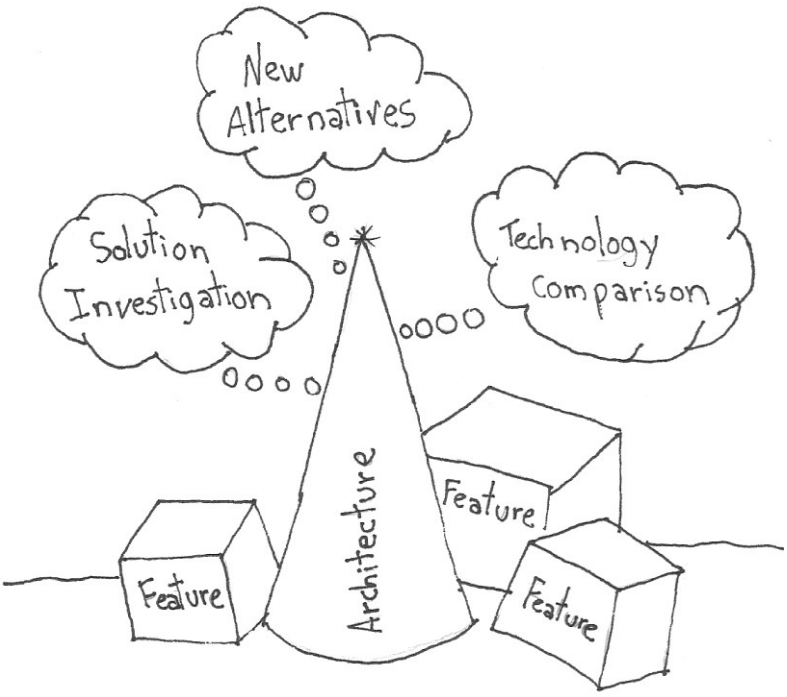
* * *

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

*In the project SADE, for an emergency response system [*PSVG]*, an architectural trigger was developed for the reverse AJAX mechanism. If measurements reported that it was not fulfilling the performance requirements, an optimization architectural task should be added to the backlog. Despite defining this trigger, the initial solution proved adequate and no additional optimization was required.*

Platforms such as Mezuro [Mei] and SonarQ [Bel] provide support for sending notifications to development teams according to configuration criteria 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 attribute requirements. Dealing with these issues takes time and the team might not have expertise in some technology. They may need to gain experience before they can confidently proceed. Also, it may not be clear which architectural option will best meet current or future needs. Consequently, some investigation might be needed before meaningful decisions are made.

How can you make architecture decisions when their impacts 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 about 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. It can be hard to estimate based on technologies or components that are not completely understood. 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. Trying to force estimates for poorly understand architecture 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.

As your software is deployed and put to use, your initial architecture may not scale. Originally your architecture might have addressed the business needs with simple solutions. However, with continued success some factors such as reliability, performance, and security may no longer be adequate. It is important to figure out how to evolve your architecture to address increasing demands.

Therefore:

When you discover that the current architecture is inadequate and you do not know how to address it, add an Architecture Spike task to your backlog to perform some study, test or alternative investigation that supports architectural decisions.

The main goal of an **Architectural Spike** is to provide information for an architectural decision. Different tasks might be performed during the spike, depending on what information is needed. If the spike aims to confirm that a particular technology is appropriate, some tests or even a prototype might be developed and evaluated. Another possible question answered by a spike is which option is better. In this situation, some metrics and criteria can be defined based on the requirements, and during the spike alternative solutions can be investigated. A spike can also search for a solution to an immediate problem that the team has no clue how to solve.

Sometimes the architecture becomes brittle and increasingly difficult to reliably and predictably add new functionality. In this case you might perform an architectural spike to investigate what can be improved in the architecture and at what cost. The outcome of this spike may be recommendations for architecture rework or areas that need further investigation.

The result of an architectural spike can be a solution, however it is not mandatory. Sometimes an architecture spike can reveal that a potential solution is not viable, or that the costs or risks of making an architecture change are greater than expected. The outcome of an architectural spike may be evidence in support of a decision to adjust the project roadmap. Even when the Architectural Spike does not result in a suitable solution or clear decision, there can be positive outcomes, such as the identification of new alternatives or the elimination of others. When the path to follow is clear, a plan can be made and tasks prioritized to change the architecture.

The difference between an architecture spike and a design spike is that an architecture spike involves decisions about potential architecture changes that can be difficult or costly to reverse, while a design spike is declared when someone does not know how to implement some functionality. For example, choosing whether to buy or build a framework is an architectural spike. Similarly, identifying approaches to improve application performance is likely an architectural spike. Deciding how to refactor complicated conditional logic, or whether the use of the **Interpreter** pattern will help solve a problem is a design spike.

A design spike may not be visible outside the development team, but an architecture spike should be apparent to the Product Owner and other stakeholders. An important consideration

for **Architectural Spikes** is to know when to work on them. When a need for architecture investigation is perceived, add an **Architectural Spike** to the backlog. When the prioritization for the next iteration is performed, it is important to understand what backlog items depend on the outcome of that **Architectural Spike** and the impact that delaying that decision would have on **Technical Debt Management**.

The time dedicated to any architecture spike needs to be bounded, usually to the length of a sprint. Determine the time to dedicate to an **Architectural Spike** and halt it when the limit is reached. If you still need more information to make a decision, then make a conscious decision to continue working on the spike for another bounded period of time.

One consequence of declaring an **Architectural Spike** is that any task that depends on its outcome is blocked. A chain of **Architectural Spikes** that do not result in a viable solution can negatively impact progress. In this case, the problem may be tackled with a more dedicated effort, such as asking for specialized help or performing an iteration where the entire team works on solving that specific architecture problem.

* * *

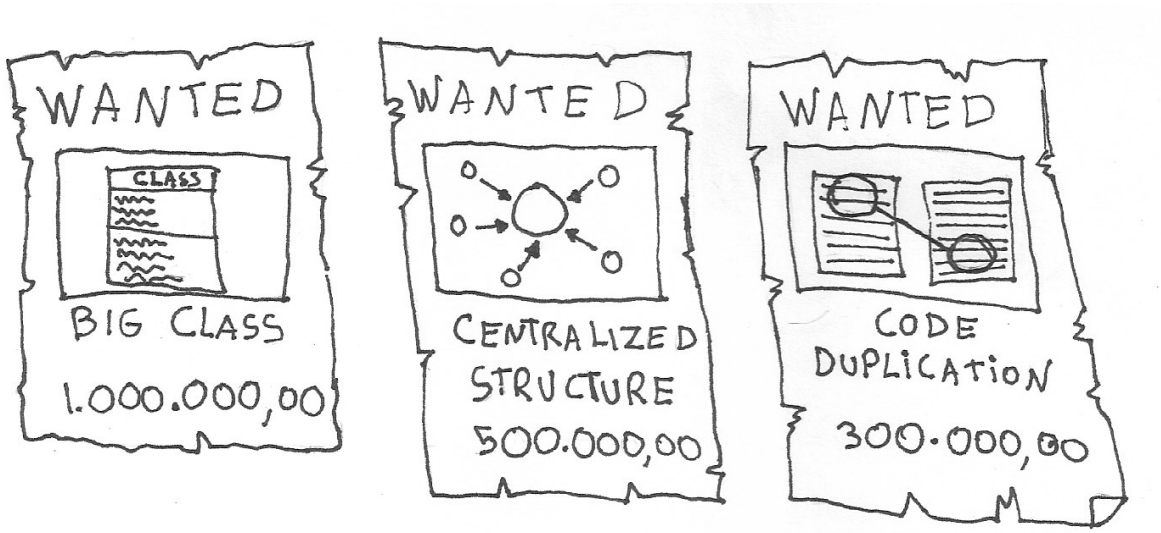
Plan for Responsible Moments can use **Architectural Spikes** to address uncertain architectural requirements. **Architectural Spikes** can be included in the plan and included as part of the **Architecture in the Backlog**.

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

The Refactory used an ***Architectural Spike*** *to design and build a solution that could generically handle imports of data for various clients in an Invoicing and Order Processing system. Many clients had similar types of orders with minor differences which led to a lot of duplication. A spike was used to create a framework to process 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. 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



Technical debt [Cun] can accumulate, if you don't take time to rework and improve the design of your system. Technical debt is not a simple issue. The cost of accruing significant technical debt makes it more difficult to rapidly change and evolve your system. Fixing designs problems is often more costly the longer you delay [ZSSS]. A recent study revealed that the management of existing debt can help avoid its uncontrolled growth [PTGO].

Accruing some technical debt can be prudent [YMS] and even desirable. One needs to balance the cost of making any design improvement with implementing new functionality. You may consciously decide to take on technical debt in order to learn something about your system and its use before improving your design. However, inconsistencies and inadequate design choices can become overwhelming if not addressed. Technical debt can compound over time if it is not managed.

How can you protect your architecture from the ravages of uncontrolled technical debt?

Some debt happens unintentionally through misunderstanding or errors in judgment. Technical debt is inevitable on any project. There isn't enough time to always improve functionality that "works well enough." On the other hand, if you do not work towards a **Sustainable Architecture** [WY], technical debt can grow and cause your system to devolve in a poorly architected **Big Ball of Mud** [FY].

It is often hard to know what appropriate action to take to reduce some technical debt. The long term consequences of a particular design choice and its accumulated effects on the overall system can be hard to understand. Deferring design rework until after you have learned more can have a huge payoff.

Some debt is urgent and easy to fix. Some debt may be costly and hard to fix. Some debt may be easy to remedy but isn't that significant. Prioritizing debt reduction efforts can be difficult.

Some technical debt does not cause problems as it is in a part of the system which is fairly stable and isn't changing much. Trying to pay off that debt may be more costly than any benefit that comes from reducing it. However, the economics may change in the future, when that part of the system needs significant enhancements. Subsequently, it can become worthwhile 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 proactively for business reasons.

Therefore:

Identify and manage the technical debt present in the project along with the respective effort to fix it.

To manage technical debt, consider the effort to fix it, the consequences that debt has on the current users of the system and ongoing development, and how much that debt is projected to compound if not fixed. **Technical Debt Management** takes time and requires discipline from the team. However, reducing debt need not consume all of a team's energies. The most critical problems should be prioritized first. Not all debt is equally important to reduce. For example, it may be more critical to fix a performance issue that is slowing down every user than to re-implement a service in order to use the standard logging mechanism.

The amount of technical debt and architecture work can be made apparent by coloring backlog items to make them stand out: architecture features including quality-related items can be colored yellow and technical debt reduction items colored black [Kru].

If the technical debt is extensive it can be difficult to balance debt reduction with new feature development. Sometimes a product owner may not want debt reduction items on the main product backlog because their main focus is on delivery of new features. In that case, create a separate technical debt reduction backlog which is prioritized separately. One issue to resolve when having separate backlogs is how to coordinate the work and manage dependencies between User Stories and debt reduction tasks. While some believe there should be only one backlog, there can be benefits to having a separate technical backlog. A separate team might be dedicated to reducing technical debt while addressing architecture tasks. Do what works best to keep the proper focus and priorities clear.

Technical debt reduction efforts need monitoring and metrics can be gathered by using existing code analysis tools. **Continuous Inspection** can be used to examine debt and monitor its impact. Inspection tools can be used to verify how much the debt is growing.

There are different approaches for reducing technical debt. Refactoring the system to eliminate the debt is the most straightforward approach. However, depending on scope of the refactoring effort, it can require a lot of investment that might not pay off.

Another option for containing debt instead of reducing it, is to isolate debt ridden parts of a system to minimize their impact on the rest of the system by applying patterns such as **Sweeping it Under the Rug** [FY], **Wiping Your Feet at The Door** [WY] and **Anti-Corruption Layer** [Eva].

* * *

When the team does not know how to address some specific technical debt, it can use an **Architectural Spike** to explore possible solutions. An **Architectural Trigger** can be used to identify when technical debt has become a problem that needs to be handled.

Pires et. al [PSVG] conducted a study of three teams at a large company which revealed that visibility of problems and their consequences was increased by managing technical debt.

Noman [Noa] found that reducing clutter in code by making simple refactorings first, makes it easier to make more architecturally significant refactorings. He also found that if debt reduction occupies more than 10% of their overall workload, teams become weary of debt reduction and that the amount of time a team spends on debt reduction has diminishing returns.

The Refactory collaborated with a development team that scheduled 1 out of every 4 sprints dedicated to technical debt management. This was an agreement between their customer and the development team. They did not always use the sprint for technical debt management but had it in the plan if needed.

7. Summary

This paper presented four patterns for developing and sustaining architecture that can be used in the context of agile projects having complex architectural requirements. These patterns focused on practices to be used during project development and extend those previously written for defining the initial architecture for agile projects [WYG].

We have brainstormed additional candidate patterns during various writers' workshops and discussions with colleagues. These candidate patterns are: *Architectural Sprint* (similar to **Quality Focused Sprint** [YW]), *Share Architectural Knowledge and Responsibility*, *Build one to Throw Away*, *Minimize Architectural Risk*, *Review Plans and Assumptions*, and *Don't Overdesign or Defer Decisions*. Some of these candidates might become full-fledged patterns while some may not.

Although these patterns presented in this paper have relations to other patterns, our collection of patterns is not yet a pattern language. In the future we intend to investigate the relationship among various 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 thank our shepherd Stefan Sobernig for his valuable comments and feedback during the PLoP 2015 shepherding process. We also thank our 2015 PLoP Writers Workshop Group, Michael John, Wiebe Wiersema, Cees de Groot, and James Thorpe, for their valuable comments.

References

[AL] Ambler S., Lines A., *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterpris*e, IBM Press, 2012.

[Bel] Bellingard F., Blog post, "Effective Code Review with Sonar," October 20, 2011, http://www.sonarqube.org/effective-code-review- with-sonar.

[Bin] Binstock A., Blog post, “In Search of Agile Architecture”, posted on November 04,2014, http://www.drdobbs.com/architecture-and-design/in-search-of-agile - architecture/ 240169245.

[BSWL] Breivold, Sundmark, Wallin, & Larsson. What Does Research Say about Agile and Architecture?” Paper presented at the Fifth International Conference on Software Engineering Advances (ICSEA), August 22-27, 2010.

[Cun] Cunningham W., Dec. The Wycash Portfolio Management System, SIGPLAN OOPS Messenger, 1992. Vol. 4 (2), pp. 29–30.

[Ev] Evans E., *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

[FY] Foote B., Yoder J., “Big Ball of Mud,” *Pattern Languages of Programs Design 4* Harrison N., Foote B., and Rohnert H., editors. Addison Wesley, 2000.

[Gre] Green P., Blog post, “Splitting stories into small, vertical slices”, Sept 27, 2013, http://blogs.adobe.com/agile/2013/09/27//splitting-stories-into-small-vertical-slices.

[HSWL] Hongyu P., Sundmark D., Wallin P., Larsson S., “What Does Research Say about Agile and Architecture?” The Fifth International Conference on Software Engineering Advances, ICSEA 2010, 22-27 August 2010, Nice, France.

[Knu] Knuth D., 1974, “Computer programming as an art,” Communications of the ACM, 1974, Vol. 17, No. 12, pp. 667-673.

[Kru] Kruchten P., Blog post, “The Missing Value of Software Architecture,” http://philippe.kruchten.com/2013/12/11//the-missing-value-of-software-architecture, 2013.

[Lef] Leffingwell D., *Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007.

[Mad] Madison J., "Agile Architecture Interactions," IEEE Software, vol. 27, no. 2, pp.41-48.

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

[MYGA] Merson P., Yoder J., Guerra E., Aguiar A., “Continuous Inspection - A Pattern for Keeping your Code Healthy and Aligned to the Architecture,” AsianPLoP, 2013.

[Noa] Noaman A., “Refactoring as a Lifeline: Lessons Learned from Refactoring,” Agile

2013 Conference Experience Report.

[Pro] Proegler E., “Performance Testing in Agile Contexts,” presentation at Agile 2015, https://submissions-production.s3.amazonaws.com/attachments/attachments/000/001/671/original/Eric_Proegler_Agile2015_Performance_Testing_In_Agile_Contexts.pdf?AWSAccessKeyId=AKIAIKK35CR3V3P7N25A&Expires=1455771430&Signature=%2BWzTmpaAo%2BaRHSsnMvQ5jki3utk%3D.

[PSVG] Perillo, R., Silva, J. R. B., Varga, R., and 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.

[PTGO] Pires R., Tonin G., Goldman A., Oliveira, R. “The effects of a model for identifying and managing Technical Debt in software projects,” Workshop Brasileiro de Métodos Ágeis 2014 (WBMA 2014).

[Spe] Spence J. “There Has to Be a Better Way,” Agile 2005 Conference Proceedings.

[WNA] Waterman M., Noble J., Allan G., "How Much Architecture? A Grounded Theory of Agile Architecture" AGILE India (AGILE INDIA), 2012.

[WY] Wirfs-Brock R., Yoder J., "Patterns for Sustaining Muddy Architectures", PLoP'2012, October 19-21, 2012, Tucson, Arizona, USA.

[WYG] Wirfs-Brock R., Yoder J., Guerra E., “Patterns for Initial Architectural Design on Agile Projects,” AsianPLoP, 2015.

[YMS] 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.

[YW] Yoder J., Wirfs-Brock R. "QA to AQ Part Two Shifting from Quality Assurance to Agile Quality - Measuring and Monitoring Quality”, PLoP'2014, September 14-17, Allerton, Illinois USA.

[ZSSS] Zazworka N., Shaw M. A., Shull F., Seaman C., “Investigating the impact of design debt on software quality,” Proceedings of the 2nd Workshop on Managing Technical Debt. MTD ’11, 2011, New York, NY, USA, pp. 17–23.