Software Architecture in the Agile Life Cycle

Diego Fontdevila[1], Martín Salías[2]

[1]M.Sc. Software Engineering Management, Carnegie Mellon University, United States
[2]Senior Architect at Southworks, United States

Abstract

This article proposes a set of techniques and practices to leverage the agile approach to software architecture—increasing overall quality, streamlining development practices, and providing business value as a constant flow.

The article describes issues that are related to component API design and behavior-driven design, continuous measurement of complexity, automated quality-attribute evaluation, and design rationale recording. The reader should take away from the article several techniques to research and try, a basic development life cycle, and some leads for further investigation

Keywords: Agile approach to software architecture

1. Introduction

Even while agile methodologies are getting widely accepted in the development world, there is still a lot of debate about how to apply them to the architectural space. One of the most conflictive issues stems around "big design upfront," which is strongly discouraged by agile practitioners, and the traditional approach to architectural design.

This article proposes a set of team dynamics, conceptual practices, and specific technologies to embed software architecture within the agile approach—keeping up the shared goals of technical excellence, streamlined development practices, and a constant and ever-increasing flow of business.

It is the hope of the authors that readers can later compare our experiences with their own and provide further discussion, so as to keep improving our professional corpus.

2. Architectural Dynamics in Agile Teams

One of the 12 principles of the Agile Manifesto states that "the best architectures, requirements, and designs emerge from self-organizing teams." [1] We take this to heart—especially, the reference to our shared specialization.

While architecture is an activity that is historically performed with an emphasis on the early stages of a project, the main focus of agile development is on emergent design and iterative production— creating a series of interesting challenges down the road.

First of all, agile makes a big push toward shared responsibility and, thus, dilutes the traditional role of the architect as the one who "defines" the higher-level design of a solution. In this new approach, architecture (as most other development activities) is something that is performed by the whole team—preserving its multidisciplinary nature. This does not imply that the architect profile goes away, as with all the other roles; it means that while someone contributes with a broader and probably more experienced perspective (usually leading in this aspect), the whole team participates and understands the implications of the design decisions that it makes, and continuously evaluates them.

In our experience, key considerations—such as the modularity strategy, how communication is handled within and outside the application, and how data and services are accessed and abstracted—are successfully defined and implemented when the whole development team establishes a consensus about these issues. In this way, team members fully understand the consequences of the selected alternatives, remain aware of their initial assumptions thorough the solution life cycle, and quickly raise concerns when their validity is affected.

Most of these challenges are usually tackled by folding architectural discussion and revision into the regular meetings that take place over the course of an iteration—such as planning and review meetings, and frequent sync-ups and design meetings with plenty of white boarding and open talk. It is also worthwhile to have the most important guidelines permanently exposed in an informative space, including diagrams, checklists or reference charts around the walls and semi permanent flip charts that are used as posters.

Published at: The Architecture Journal - MSDN - Microsoft

This article does not cover in detail specific techniques that apply to coordinating several sub teams; mainly, it mirrors the standard guidelines about the "Scrum of Scrums"[2]. The addition to such activities is a stronger focus on the preservation of conceptual integrity—thus, planning frequent high-level design meetings between teams. Again, these meetings should avoid becoming architect meetings; while the contribution of team members who have a stronger architectural background is obviously important, it is very important for other members to participate. Even the less experienced team members can provide a somewhat naïve perspective to some discussion—promptly flagging complexity excesses that are a professional malady among us architects.

To close on the team dynamics, as the agile perspective goes over the standard view of the development team and extends to customers, operations personnel, and other stakeholders, expectation management is a big deal also for the solution architecture. As the next section shows, there is a strong emphasis on mapping the needs and goals of these actors to the architectural constraints and converting the most important into strong metrics to be evaluated.

3. Agile Architecture Patterns and Practices

There are several common approaches to support the previously described dynamics and keep the agile principles of high customer involvement and feedback, continuous delivery of working software, and attention to technical quality, among others.

3.1 Sashimi

There are several common approaches to support the previously described dynamics and keep the agile principles of high customer involvement and feedback, continuous delivery of working software, and attention to technical quality, among others.

One of the most common patterns that we use to avoid the perils of big design up front is the "sashimi" approach to the architectural definition. In this approach, instead of spending a lot of time designing and implementing the different moving parts around layers and tiers, crosscutting concerns, and so on, we build the minimal amount of code that is needed to connect all of the pieces and start building the actual functionality on top—providing an early end-to-end experience of the results. Indeed, the focus is more on the API level of the infrastructure, and not the actual implementation, which is usually mocked up for the first little iteration.

The main purpose is to avoid building architecture components that are hard to use or tying the business logic and other high-level abstractions to the underlying implementation. As iterations progress, the actual implementation is incrementally completed, following the needs of the functional part of the application. At some point, such things as load or stress testing that is performed over the functional side of the solution will even require fine-tuning of these components for robustness, increased performance, resource consumption, and so on.

To be able to support this emergent implementation over architectural pieces, definition of a highly decoupled API is the most critical factor. Whenever implementation details permeate outside the API—hence, coupling with its consumers—refactoring the architectural components becomes a nightmare. That is why API design becomes a key activity in the earlier stages, and why starting with no implementation at all is a better approach.

This practice applies even when using third-party components, which is both common and generally advisable, for the most part. In such cases, existing default implementations for those third-party components provide early support modules; and, many times, configuration is needed instead of coding in the early stages.

Iteration	1	2	3	5	10	15
UI layer	Home, with login	Custom areas	User contacts	…	…	…
Business layer	None, really	Layout validation	Social graph	…	…	…
Data layer	User name	Profile	Social data	…	…	…
Crosscutting concerns	Authentication (mocked)	Authentication (basic)	Logging (mocked)			

Table1. Example of how actual functionality and architecture grow iteratively on common three-tiered Web application. Note how the load time for the home page (a very important metric, in this case) is measured since the first iteration.

Table 1 shows an example of how this works in practice, as iterations go by. Note that at the end of the first iteration, the application goes throughout all of the proposed layers, and how the most important nonfunctional requirement (home-page response time) starts to be under control from then on, across the whole project.

Of course, this first test can be done with a single concurrent user, and it measures mainly static content; but the thresholds will be in place as back-end generation goes, and testing will involve many concurrent

connections in future iterations. However, no one can change functionality or infrastructure and affect response time without being noticed immediately, then reducing the fixing effort.

3.2 Architectural Patterns

Another common practice in the agile development of software architecture is the concentric approach, in which the starting point is a high-level technical vision of the solution, which the team can shape collaboratively, as previously described. This technical vision will provide the conceptual baseline that will serve as both a reference point to focus future work and a sanity check for refactoring (more on this later, when conceptual integrity is discussed).

The second level is the module decomposition, which consists of a set of modules with services that provide actual value to users or other modules and allow for a coherent separation of responsibility. These modules work as placeholders to which specific functionality can be added incrementally through the design and construction process. This decomposition provides a high-level grouping of components that make the design more manageable for both architects and other stakeholders, and the modules work sometimes as namespaces to help identify stakeholder concerns.

The third level is a decomposition that is usually described in terms of architectural styles or patterns—layers and tiers, in particular— for enterprise or business-information applications. At this level, the usually most significant definitions are thelayers, which are varying levels of abstraction, in terms of user-level value (in this case, the lower level of abstraction is what the end user knows the least)—in particular their API, as previously described—and tiers, which describe a structure for separating responsibilities according to their volatility and allowing for distribution. This level is the first that has well-defined interfaces and is usually considered good for work allocation among teams. That kind of allocation must be handled carefully to avoid architectural mismatch between the parts, as well as to keep from losing the advantages of collaboration to the hard separation of work pieces [3].

The fourth level is that of components, which are packaged pieces of software whose very specific responsibilities are defined by their interfaces and, possibly, with multiple implementations that can be selected dynamically. These are usually the highest-level pieces that software-development platforms recognize conceptually (in other words, those that are seen by the platform, which, in terms of syntax, means that the platform has the terms that correspond to that component or component type). At this point, our agile teams start to gain the capacity to use directly the language that they share with their users in the software that they produce.

The fifth level is the class level—finally, the object-oriented level of decomposition. At this level, programming languages are at their best, and developers can fully use the language that they share with the stakeholders in the software that they write (programming-language code and software configuration). Figure 1 illustrates a quick review of the concentric approach.

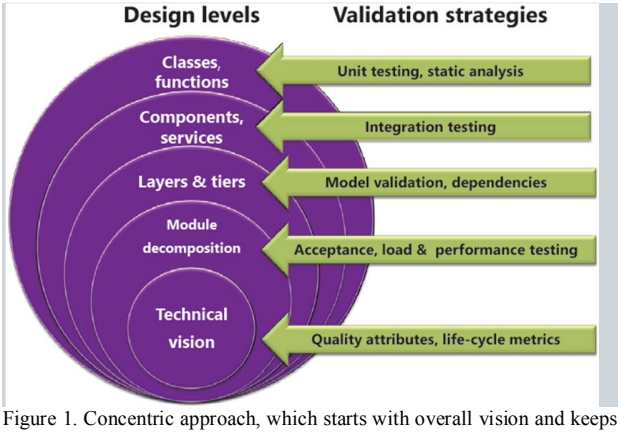

Figure 1. Concentric approach, which starts with overall vision and keeps growing as we get closer to final implementation. (All levels are refactored over time but kept in sync, although the inner levels usually stabilize faster.) (Click on the picture for a larger image)

Note also that we can use to our advantage domain-specific languages [4] providing a higher-level abstraction to how components orchestrate between them at the fourth level, or getting the domain closer to the object modeling at the fifth level. This latter approach can be leveraged by using an external DSL or an internal one, which often can be built by following domain-driven design [5].

All of these levels (which, in architecture literature, are also called structures [6]) can also be considered independently, according to the specific needs and scope of each project.

3.3 Quality Attributes and Architecture

One of the most common discussions about architecture is about what aspects of a system's design are architectural in nature. In particular, quality-attribute-related requirements are most often determined by the architecture. From an agile perspective, it is very important to keep in mind that quality-attribute requirements must be managed as part of the product backlog and implemented incrementally. Specifically, that means managing the prioritization of a heterogeneous mix of requirements, both features and quality-attribute requirements. Another aspect of interest is the fact that multiple quality attributes tend to require trade-off analysis and decisions, where standard prioritization might not be enough.

To manage quality-attribute requirements effectively, the authors recommend considering the quality attribute as a user goal, with specific requirements built into user stories that support that goal. The stories must have measurable acceptance criteria defined clearly, so that tests can be written for the components that are implemented. Examples of these requirements (with metrics in parentheses) areflexibility(complexity, dependencies, coupling, layering), performance (response time, resource usage), and scalability (load and response time). These metrics should be integrated with the continuous build process, as the next section will show.

3.4 Architecture Validation

To finish this section, the authors present the key practices for testing and validation that are related to architecture. From our perspective, these are test-driven development, automated integration testing, automated quality-attribute requirements testing, automated deployment, environment-configuration management, and application-configuration management.

As described in the first part of this section, the authors believe in the early definition of interfaces. These definitions, wherever possible, must be created in terms of executable unit tests (or supported in some other way by the language or testing harness, such as language-syntax pre- and post-condition specifications [7] .Not only will these specifications be the safeguards in place for local and multi component refactoring, but they will also provide the entry point for finding defects when an incident is reported. The idea is that any incident that is reported will require finding the applicable test, so that if it is not there, it can be created; otherwise, it must be modified to catch the defect, and then the implementation can be corrected.

It must be kept in mind that many architecturally significant changes will escape notice by unit tests.

To manage changes that exceed the unit-test contracts, automated tests are required for integration and quality attributes. The latter tend to be harder to create, but they pay off when quality-attribute requirements that are hard to implement are affected. These tests usually need to be scheduled with lower frequency than unit tests, depending on their resource usage.

Examples of these are:

Scalability. Acceptable response times when system load is increased to a certain level. Implementation of such tests requires not only tool support, but also careful capacity planning for the testing environment—both client side and server side, when applicable—and automated deployment to the testing environment.

Flexibility. Instantiation of the layers pattern. Implementation of supporting tests includes dependency metrics matching the structure of the pattern implementation. As described in the following section on model base evaluation, it requires the configuration of tests to accept upper-layer to adjacent lower-layer dependency, and not the reverse.

For all of this to be possible, it is necessary to manage configuration in two levels: environment-dependent and environment-independent. Managing environment-dependent configuration will enable automated deployment, and will focus on physical and logical resource configuration. For the rest of the configuration, the issue will be defining variability of available functionality (usually, dependent on the customer).

The next section discusses the use of available technologies for the implementation of these practices.

4. Specific Techniques and Technologies

To implement reasonably the techniques that the previous section described, it is necessary to use appropriate tools and technologies, not only because of the expense that is incurred, but also to provide the necessary discipline through automation.

As the agile mindset stated in its manifesto, individuals and interactions are more important than processes and tools; from there, however, the agile world has derived a helpful set of tools that take tedious manual tasks away from people and make them easy to execute fast and frequently—providing a lot of feedback upon which individuals can act. For our architectural quest, the authors follow the same principles and basic ideas and extend them to cover the concepts that have been discussed.

The first level of technologies that are used can comprise regular testing tools and frameworks, such as unit-testing tools—from the traditional xUnit (such as jUnit, NUnit, cppUnit, and MS Test) to the ones that come from behavior-driven development [8] (such as RSpec, xUnit.net, JBehave, and Cucumber, among others). Included also are user-acceptance or functional testing tools (such as Fit/Fitnesse, Selenium, and Watir, among others) and a host of technologies that are needed for performance and stress testing. All of these, of course, run at an individual level, as well as on the build server, and with different frequencies (unit tests in every check-in, functional a few times a day, load and stress usually over the night, and so on).

In short, we build up from the basics of the appropriate development practices—adding some specific test at the unit, acceptance, or stress level to validate some architectural concerns. To this standard tooling, a second level is added—with more specific checks over quality attributes, such as lines of code per class/module, code-coverage statistics, static analysis, style analysis, cyclomatic complexity, afferent and efferent coupling, dependencies, and more. Some of the tools that are used in this space are (for .NET) FXCop, StyleCop, NDepend, and

built-in tools in Microsoft Visual Studio Team System; and (for Java) FindBugs, JDepend, Checkstyle, Lattix, and built-in features on IntelliJ IDEA. Within the realm of dynamic languages such as Ruby, JavaScript, and Python, this is a less developed area, because of the inherent difficulty of performing static analysis on them. However, there is strong evidence that shows that as the runtime engines are going increasingly the way of just-in-time compilers, this gap will be filled soon.

Then, there is a third level of metrics about flexibility and maintainability that has to do with the project life cycle itself— metrics such as code-churn, volatility, correlations, and adherence to the architectural models. In this space, Visual Studio Team System is making great strides, while there are many people who implement part of this by using build-tool plug-ins or custom scripts that crunch data and produce reports or alarms, based on data that comes from the source repository, build server, issue tracker, testing environments, and modeling tools.

Indeed, to be able to perform validation against an architectural model, such a model has to be in place. To do so, we can pick among myriad tools—from Enterprise Architect (or some of the Rational suite of tools) to Visual Studio Team System. What is important here is to take the time to automate the process to extract the relevant metadata that is needed to validate the code, references between packages or services, or module composition.

Additionally, it is very important to distinguish the code or module view of the system from the runtime view of the system during evaluation. Runtime characteristics are usually harder to perceive, but their high implementation costs make early analysis and testing worthwhile. Finally, it is very useful to learn also how to perform some level of reverse-engineering—allowing to grab some information from the actual implementation into the model, and automating part of the documentation chores.

The final step of this methodology involves the deployment and configuration of the different staging environments, in which virtualization becomes an incredible enabler—allowing for quick turn-on and turn-off of all the needed environments (with baseline configuration), where we can use remote scripting to perform the deployment of the latest build and configuration to any of these environments, and then perform all sorts of testing. The current power of virtualization platforms such as VMWare, Hyper-V, and others makes it really easy to manage multiple basic images—taking and reverting to snapshots, even across distributed physical machines. Of course, all of this is not something that the authors encourage anyone to try setting up from day one. Instead, you should increasingly add over each iteration, but have all of the appropriate (and project-relevant) techniques folded into the main plan, to ensure that these controls are getting into place as the project goes on.

5. Conclusions

The authors of this article believe that architectural considerations are fundamental for delivering value in most software projects—also, that agile teams have much to offer in terms of mechanics, techniques, and tools for the software-architecture community. These contributions are best considered in terms of the development of a language that is shared by all stakeholders and spans the spectrum from the user's view of the system to the actual code. This language consists of the set of both user requirements and design decisions that are made during the life of the product. Its final purpose is to allow users and teams to create excellent results that will provide value, according to the expectations of stakeholders, throughout the lifetime of the product.

References

[1] Fowler, Martin, et al. Manifesto for Agile Software Development Web site, 2001.

[2] Cohn, Mike. " Advice on Conducting the Scrum of Scrums Meetings." Mountain Goat Software Web site, May 2007. (Originally published in Scrum Alliance Web site.)

[3] Austin, Robert D., and Lee Devin. Artful Making: What ManagersNeed to Know About How Artists Work. New York: Prentice Hall, 2003. (Page 144.)

[4] Martin Fowler is currently writing a whole book on DSL, but the work in progress is available athttp://martinfowler.com/dslwip/.

[5] Evans, Eric. Domain-Driven Design: Tackling Complexity in theHeart of Software. Boston: MA, Addison-Wesley, 2004.

[6] Bass, Len, Paul Clements, and Rick Kazman. Software Architecturein Practice. Second ed. Boston, MA: Addison-Wesley, 2003.

[7] Mandrioli, Dino, and Bertrand Meyer. Advances in Object-OrientedSoftware Engineering. New York: Prentice Hall, 1991. (Chapter 1, "Design by Contract.")

[8] North, Dan. "Introducing BDD. " http://dannorth.net/introducing -bdd. DanNorth.net Web site, September 20, 2006. (Originally published in Better Software Magazine Web site.)

Diego Fontdevila, Professional Services Director at Grupo Esfera, specializes in software architecture and agile methodologies. He has 13 years of experience as both software developer and university teacher. Currently, Diego is a Master of Software Engineering Management student at Carnegie Mellon University.

Martín Salías, Senior Architect at Southworks, has more than 25 years in the software industry, working on different industries for customers around the world, and covering many platforms and languages. He is a member of the Agile Alliance and has been awarded as a Microsoft MVP since 2002.