FINDING STRUCTURAL DECAY IN ARCHITECTURES

Most things around us wear out and eventually experience decay; the we drive, household appliances and electronics, highway bridges, even ourselves. Software architecture is no different. One of the many responsibilities of a software architect is to continually assess the architectures supporting applications to determine whether they are still sound or in need of repair. But what does that mean and how do we do that? In this article I will discuss both architectural (macro) and source code analysis (micro) techniques for finding and fixing structural decay in your application architectures.

Architecture and Structural Decay

Software architecture can be thought of as the structural aspects of the application combined with architecture decisions and design principles (collectively known as architectural assertions). The structural aspects of an architecture form the overall shape of the application, describe how components interact, and describe how source code is written and organized. The shape of the application is represented by the type of architecture pattern you are using, whether it be microservices, service-based architecture, layered architecture, microkernel architecture, pipeline architecture, event-driven architecture, or any combination thereof. Structure also refers to the "-ilities" (or architectural characteristics as I like to call them) such as performance, reliability, availability, scalability, and so on. However, the structural aspects of your application are only half of what architecture is. The other half

include those architectural assertions you make such as "*only the business and services layer of the architecture can communicate with the persistence layer*" and "*leverage asynchronous messaging between service calls whenever possible to increase performance*". Analyzing your architecture and finding structural decay can tell you whether you are using the right architecture pattern and whether your application is healthy and can realize the architectural characteristics is was meant to address.

Macro Detection

The first step in finding structural decay in your architecture is to look at the macro view - in other words, the architecture itself. Analyzing the macro view can tell you a lot about how your application will perform, scale, and meet users needs and whether it is experiencing structural decay. Macro detection focuses more on the interaction and size of the components of the architecture rather than detailed source code metrics (we'll look at those in the next section under "Micro Detection"). For Java and C# a component is usually realized through package structures. For example, the package `myapp.business.services.refmanager` refers to the component *Reference Manager* in the subdomain *Services* of the business layer of the architecture. The source code written under this particular package structure would implement the *Reference Manager* component. Because of the fined-grained nature of services in a microservices architecture, a component and a microservice are generally the same thing.

There are three general structural decay indicators that you can look for, regardless of the type of architecture pattern you are using. The first of these is static coupling. There are two types of static coupling you need to be concerned about - afferent coupling and efferent coupling. Afferent coupling refers to the number of components or services that communicate with you (fan-in), whereas efferent coupling refers to the number of components you communicate with (fan-out). It is easy to remember: "a" comes before "e" - **a**fferent comes before, **e**fferent comes after. The level of afferent and efferent static coupling directly impacts modularity, maintainability, testability, availability, deployment, reliability, scalability, and the evolutionary aspects of your application. The more static coupling you have, the more these architecture characteristics are impacted. High levels of afferent and efferent coupling can

generally be resolved through splitting up components into smaller ones, therefore reducing the overall level of coupling in your system. Always remember it is better to have more components that are loosely coupled than fewer components that are more tightly coupled.

A second general type of structural decay is temporal coupling. Temporal coupling refers to components that are bound to some sort of non-static timing dependency, such as transactions or the order in which they need to be invoked. Temporal coupling is very difficult to find, and usually requires source code or repository forensics to detect. To illustrate temporal coupling, consider 5 components named C1, C2, C3, C4, and C5 respectively. The following analysis shows the components that changed for each feature and bug fix:

```
Feature 1: C1, C3, C4
Feature 2: C1, C2, C3
Feature 3: C4, C5
Bug Fix 1: C1, C3
Feature 4: C3, C5
Bug Fix 2: C4
Feature 5: C1, C2, C3
```

Notice that in all cases, every time component C1 changes, C3 also changes, even though component C4 changed as well for feature 1. This is a good indication that components C1 and C3 are temporally coupled. A word of caution through; don't jump to conclusions right away - wait for several iterations before making a judgement. While temporal coupling impacts some of the same characteristics as static coupling, specifically it is reliability and maintainability that are impacted the most.

A third general form of structural decay is component size. In almost all cases larger components are more difficult to test, more difficult to maintain, and directly impact modularity, availability, deployment, overall reliability, and scalability. You can determine component size by looking at the number of classes within the component (e.g., package) and also the total number of lines of code within the classes implementing that component. As you might guess, component size is directly related to component coupling; the larger the

component, the more likely the component has a higher degree of afferent, efferent, and temporal coupling.

Aside from the general structural decay indicators, there are also pattern-specific decay indicators you should look out for. Interestingly enough, microservices has the most pattern-specific structural decay indicators. The first one you'll want to watch out for is too many shared libraries (e.g., JAR files and DLL's) shared between microservices. Microservices achieves its high levels of agility, availability, testability, and deployability through the notion of a strong bounded context. The more libraries you share between services, the more coupled they are, thus breaking down that all-too-important bounded context. By the same token, too much inter-service communication also creates dependencies that break down the microservices architecture.

Another thing to keep track of in a microservices architecture are aggregation requests. Aggregation requests are those business requests that require the coordination of multiple microservices to fulfill the particular request (e.g., get all customer information). While you will likely have many aggregation requests (and hence many aggregation services), you should pay special attention to how many you have and in particular if they start increasing in number. If the number of aggregation services starts increasing to a high rate, chances are you have the wrong architecture pattern and should think about more of a service-based approach. Finally, you'll also want to keep an eye out for database sharing. Reporting services and eventual consistency patterns are the biggest culprit here. With microservices, always determine what is breaking that bounded context "share-nothing" notion and analyze the trends.

For the microkernel architecture (sometimes referred to as the plug-in architecture pattern) you'll want to keep an eye out for dependencies between plug-in components. Plug-in components within the microkernel architecture are meant to be standalone, independent components. Imagine how brittle your application would become if you unplug one component from your core system and everything else breaks. Another form of structural decay in a microkernel architecture is volatility and high complexity in the core system. One of the goals of a microkernel architecture is to minimize changes in the core system and move

volatility to smaller, easily tested plug-in modules. High rates of change and complexity in the core system means that you are likely not leveraging the power of the microkernel architecture and therefore are not likely to experience the key benefits of this architecture style (e.g., agility, testability, maintainability, deployability).

With the layered architecture (sometimes called the n-tiered architecture) you will want to look for things like static cross-domain dependencies and shared infrastructure coupling. Static cross-domain dependencies refers to components that are either statically or temporally coupled between layers of the architecture or between major sub-domains of a particular layer (like customer and order domains). Keep in mind that it is likely you will always have some level of coupling between sub-domains or layers of the architecture. For example, business delegate components in the presentation layer typically call business components in the business layer and so on. However, what you are specifically looking for here are high levels of coupling, particularly from lower levels up. The higher the level of cross-domain coupling you have, the harder it is to maintain, test, and deploy your application.

Shared infrastructure coupling occurs frequently with the layered architecture and refers to the high degree of coupling between business components and infrastructure components such as a persistence manager, security manager, reference manager, configuration manager, and so on. One way of addressing shared infrastructure coupling is to distribute these components across domains. For example, rather than having a central persistence manager that every component talks to you can create domain-specific persistence managers. Using this technique, you might therefore have a persistence manager for your customer domain, another one for your order domain, another for your shipping domain, and so on. This helps control change and reduce overall coupling in your monolithic layered architectures.

Micro Detection

Another effective way you can determine structural decay in your architecture is by analyzing the underlying source code. While there are dozens of code metrics you can gather that tell you a lot about the health of your source code, these are the ones I usually focus on

for detecting structural decay:

- Number of classes per package
- Number of lines of code per package
- Percent comments
- Average depth
- Average complexity
- Depth of inheritance (DIT)
- Efferent coupling count (CE)
- Afferent coupling count (CA)

As I mentioned earlier in the article, as an architect you should focus on the component level (e.g., package level), not each individual class within those packages. Classes within a given component (or package) will always have a certain level of coupling and contain metrics outliers. However, structurally what really matters are the relationships between the components and the size of the components themselves.

As with all code metrics, it is vitally important not to focus too much on the actual numbers, but rather the trends of the metrics. For example, if you have an inventory manager component that contains 7 classes with a total of 4200 lines of code, is that too big? Looking at trends will help answer that question, as well as looking at the relative size as compared to other components. If you find that the component grows significantly after each iteration, then you can take action to split the component.

The *number of classes per package* and the *number of lines of code per package* are good metrics for determining the overall size of the component. Usually, developers will already start to see this growth and start creating sub-packages under the primary package structure.

Sub-packages are a great indicator that the component is too big and should be split up. However, creating sub-packages is not the right solution for the problem of component size. Instead, refactor the component into multiple packages in the same hierarchy. For example, consider a component called *Order Placement* in a package called `business.orderplacement` containing 15 classes with 18,000 lines of code. Due to the number of classes, the development team decides to create a sub-package called `business.orderplacement.inventorymanagement` as part of the order placement process and places 5 classes in that package structure. The disadvantage of this approach is that you start to lose sight of what the *Order Placement* component really is, and it becomes hard to control the level of coupling and separation of concerns in the *Order Placement* component. A better architectural solution is to split the *Order Placement* component into two components; *Order Placement* and *Inventory Manager*. Now you can control the relationship between these two components and other components accessing these components in the architecture.

The *percent comments* metric is always an interesting one to analyze. While the industry accepted range is between 8% and 20%, some architects insist on comments approaching the 30% to 50% range (yikes). I use this metric to tell me how complex and well-structured the code is. In my opinion a "learnable architecture" is one that has self-documenting and clear code. A high percentage of comments is an indication that the code is overly complex or not readable. Method names like `process()` and `execute()` are examples of unreadable code that would require comments to describe what the method does. I usually strive for a range of 2% to 8% comments in my code bases. Code reviews are the best way to whittle down this percentage. If I can't read your code, then it likely needs to change so that it is readable. Remember this - the code always tells the truth, whereas comments sometimes lie (due to not updating comments when code is updated).

Average depth and *average complexity* are great metrics that help determine the level of maintainability, testability, and overall reliability in your architecture. I prefer the Steve McConnell complexity metric from his book *Code Complete* over the traditional cyclomatic complexity metric. The Steve McConnell complexity metric is calculated as 1 + the number of paths through a function or method. Values between 2 and 8 are considered normal range.

Cyclomatic complexity is more complicated and is calculated is the number of nodes minus the number of edges + 2, with edges being the paths to a piece of code and the nodes being the code itself. For example, look at the following code:

```
1   int max = a;
2   if (a<b)
3     max = b;
4   return max;
```

This code has 4 edges (path from line 1 to 2, 2 to 4, 2 to 3, and 3 to 4) and 4 nodes (each statement) for a cyclomatic complexity of 2 (4-4+2). A key point when using this metric when looking for structural decay is to ignore the individual values and study the trends instead. An average complexity of a given component (package or service) of 2.33 tells you absolutely nothing. However, the average complexity trend of 2.33, 2.67. 2.98, 3.12 tells you that the component is increasing in complexity after each iteration and is becoming less maintainable and less reliable.

The *Depth of inheritance (DIT)* is a good metric for detecting structural decay with regards to modularity, maintainability, testability, and reliability. I usually refer to this metric as the "Death of Inheritance", particularly when breaking apart a monolithic layered architecture into a microservices architecture. The higher the average DIT, the harder it will be to split apart functionality into microservices. For instance, where do all of those shared abstract classes and interfaces go in a microservices architecture? Usually in shared libraries, which is yet another aspect of structural decay I talked about earlier in the Macro Detection section.

Analysis Tools

Source code analysis tools are powerful in that in addition to providing lots of useful metrics, most provide visualizations as well. Being able to visualize your source code allows your brain to see patterns that you might not otherwise see by looking at rows and columns of numbers. However, while the nice eye-candy produced by most source code analysis tools is powerful, I always like to combine them with raw data metrics as well - that's how I usually

am able to identify trends. To that end, let me describe 3 of my favorite open-source code analysis tools to use for detecting structural decay.

The first tool is an Eclipse plug-in called X-Ray (xray.inf.usi.ch/xray.php). X-Ray gives you two powerful visualizations; the first is a downward flow view which clearly illustrates the DIT metric (Depth of Inheritance) as well as the overall structure of your code (lots of smaller methods vs. fewer large methods). I use this tool when analyzing the overall level of effort to split up a monolith into microservices because it clearly illustrates the inheritance structures. The higher the DIT, the more difficult it will be to break apart your monolith. It also helps me find seams in the code. The other view in X-Ray is a coupling dependency view that allows you to zoom in and out and adjust the level of afferent and efferent coupling to visualize the dependencies in your code. JDepend (github.com/clarkware/jdepend) and NDepend (www.ndepend.com) are two other analysis tools that also do a good job at detecting coupling between components and go a little deeper than X-Ray does for dependency management.

Perhaps one of the best eye-candy source code visualization tools is Code City (wettel.github.io/codecity.html). Code City provides you with a visualization of your code as a city landscape. Not only is it fun to use, but also provides you some really useful information. Each city block represents a component (package), and each building in the city block represents a class. The height of each building represents the number of methods in that class, and the width of each building represents the number of attributes of that class. Visually it is relatively easy to identify hotspots through the number of skyscrapers that appear vs. big flat 2-story warehouses. Code City also is able to visually replay a time-sequence of the city blocks so you can see which iterations went awry and where problems started occurring.

While X-Ray and Code City are good visualization tools, my favorite tool to use to detect structural decay is Source Monitor (www.campwoodsw.com/sourcemonitor.html). Source monitor can analyze C++, C, C#, Java, HTML, Delphi, VB.NET, and Visual Basic, and is simple and easy to use. Using the simple wizard screen flow, just specify the source directory you want to analyze (the entire source base or a single package structure) and source monitor will run analysis on that source base and produce loads of metrics. While it is great for identifying

trends, the thing I like best about Source Monitor (aside from it being free) is the fact that you can export method-level metrics to a comma-delimited file, and run your own analytics on each iteration to determine when components are added or removed, when components get to big, and best of all, temporal coupling candidates.

Conclusion

As architects we don't always have the time to sit down with multiple development teams and continually analyze the architectures. These basic macro and micro techniques can help you determine warning signs (structural decay) and address problem areas before things get bad. Think about it as a toothache. If you go to the dentist right away, it's a matter of a small cavity to fill which isn't that painful. However, wait too long and you are looking at a root canal (yuck) or possible tooth extraction (double-yuck). It's the same with software; the quicker you address decay, the less painful it will be to fix it.

I give a 90 minute session at NFJS (www.nofluffjuststuff.com) titled *Analyzing Software Architecture* where I cover this information in much more detail, show the visualization tools, and then use the tools an on actual source code base to demonstrate how to detect structural decay. So look for me at the next local NFJS symposium in your area to learn more about detecting structural decay.

In the meantime, Adam Tornhill wrote a fun book called *Your Code as a Crime Scene* in which he takes you through a great journey of code forensics to detect issues in your code (and whether you even have the right team in place!). Another more academic resource is a book titled *The Art and Science of Analyzing Software Data* by Christian Bird, Tim Menzies, and Thomas Zimmerman.

About the Author

Mark Richards (www.wmrichards.com) is an experienced, hands-on software architect involved in the architecture, design, and implementation of microservices architectures, service-oriented architectures, and distributed systems. He has been in the software industry since 1983 and has significant experience and expertise in application, integration, and enterprise architecture. Mark founded the New England Java Users Group and served as president from 1999 through 2003. He is the author of numerous technical books and videos, including several books on Microservices (O'Reilly), the Software Architecture Fundamentals video series (O'Reilly), Enterprise Messaging video series (O'Reilly), Java Message Service, 2nd Edition (O'Reilly), and a contributing author to 97 Things Every Software Architect Should Know (O'Reilly). Mark has a master's degree in computer science and numerous architect and developer certifications from IBM, Sun, The Open Group, and Oracle. He is a regular conference speaker at the No Fluff Just Stuff (NFJS) Symposium Series and has spoken at hundreds of conferences and user groups around the world on a variety of enterprise-related technical topics.