Software Architecture Analysis

Improve Software Longevity and Business Agility
Introduction

The consistent use of sound application architecture contributes to the success of any software development initiative. The pressures of delivering products on time and on budget conspire to compromise application architecture, and ultimately the quality and security of your code. Application architecture is not simply something that is designed at the beginning of a development cycle and then left alone. It is a critical, ongoing concern that extends across the lifecycle of any given application to ensure the overall integrity of a given code base, the reusability of application components and a consistent structure throughout large software systems. As your code evolves, so does your architecture.

Too often, the design of a system becomes lost during an application’s lifecycle as the code evolves over time, resulting in an excessively complex and brittle software system. Software designers and architects spend significant effort on the design of a software system before the system is implemented, but unfortunately, there is no way for that architecture to become a living part of the software system as the resulting code does. Therefore, software quickly becomes un-maintainable and brittle. Dedicated maintenance of the application architecture alongside the source code will lead to flexibility in code and the long term maintainability of your software, drastically increasing the long term ROI of development efforts.

To meet the demands of the business and the marketplace, development teams must create code that is reusable, secure, and defect-free. Teams also need the ability to regain control over code bases that have evolved over multiple iterations and diverged from their intended architecture.

This paper explains how better application architecture can help developers ensure the integrity of their applications in the face of growing development pressures. It explains the discipline of application architecture analysis, and how developers can identify and eliminate incorrect code dependencies through the use of automated architectural analysis and visualization tools.
While there have been many attempts to define application architecture, for the purposes of this paper, we consider the architecture of an application to be the blueprint for a piece of software. It describes the various parts of the software system and prescribes how those parts should interact. The architecture encompasses how pieces of code will depend on each other as well as how data will be stored and flow through the system. Architecture can also prescribe the manner in which the code should be implemented through one or many design patterns, as a software architect usually spends significant time white-boarding the architecture of a system or using UML to articulate it.

For a more tactical definition, application architecture can be thought of as the way in which the lines of code that comprise a code-base are organized into higher-level abstractions such as functions, classes, files, packages, and so on. When described this way, code can be visually represented, and the dependent relationships between discrete code components can be understood. By understanding the intended structure and design of an application, developers can ensure the original design continues to be enforced throughout the evolution of the code itself, hence avoiding unwanted dependencies or ‘spaghetti code.’

The Problem with Application Architecture

In the early stages of development, an application’s architecture will be in a pristine state. Once the coding begins, so does a slow degradation from the original architecture. While the general purpose of an architecture (or blueprint) is as a means to communicate plans to a distributed team or contributor, there are no tools to effectively communicate that architecture to all software developers such that every line of code that a developer writes is ensured to conform to the desired architecture.

As a result, the original architecture of an application becomes forgotten over time, across geography and among developers. When code evolves, its original design slowly degrades with each change or modification made that fails to reflect the intended architecture of the program. Worse still, the architecture of such an application becomes excessively complex. The result of this is an array of expensive, time consuming issues that can take development teams weeks, months and even years to fully remedy. The more immediate impact of architectural complexity is the increased difficulty of delivering new features and functionality when the underlying code is strewn with dependencies that can trigger unpredictable, crash causing behavior when violated. From a business standpoint, this lack of agility prevents companies from quickly adding features based on customer request or market opportunity. This slow calcification of code may also serve to shorten the lifespan of revenue generating applications as they become less flexible over time.
There are many pressures facing developers that drive them to de-prioritize application architecture (see Table 1 for details). Generally, these forces are related to an immediate ‘time-to-market’ need. For most development teams, application architecture is a long-term issue, because good application architecture will generate significant benefits later in the application lifecycle.

The single largest challenge to effective enforcement and analysis of application architecture has been the lack of visibility for development teams. Similar to architects designing a physical building, software architects need the ability to visualize their complex application at the code level if they are going to understand and optimize the dependencies between code blocks.

Senior developers and development managers know that code needs logical organization for it to be maintainable and reusable over time. This is because code needs the ability to evolve from its original structure to include new features or meet new business goals. Code must also be both flexible and scalable so that it can continue to be adapted to new market and customer requirements years after its release. Development teams pay a significant price for the failure to maintain the integrity of an application’s architecture in both the near and long term. The remainder of this paper will discuss the common application architecture problems and how they can be solved with advanced new software development tools.

Who are Application Architects?

In any organization, team members wear many different hats. Architects within development teams can take many forms. In some groups, the architect is a senior developer or manager who is responsible for a team that maintains multiple code bases. While in others, the architect is a clearly-defined role with specific responsibilities. For more explanation about what application architects do, consider the following definition from Gartner:

“Application architects help realize a smooth transition from the current application portfolio to the future enterprise solution architecture. Through the effective development, management and sharing of high-quality application design artifacts, they promote the development and management of leverageable application and information services that can be assembled and reassembled (that is, orchestrated) to flexibly address changing business process workflows. Consider the discipline of application architecture a key part of enabling the next generation of IT solutions in your organization.”

(Source: Gartner, Defining the Discipline of Application Architecture, 2008)
<table>
<thead>
<tr>
<th>Table 1: Pressures That Attack Application Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
</tr>
<tr>
<td>Developers are under constant time pressure to deliver new features, meet release dates and innovate. It’s easy to understand how developers lack the time to consider the long-term architectural implications of a quick fix when business managers are demanding something as soon as possible.</td>
</tr>
<tr>
<td><strong>How architectural integrity can help</strong>: Architectural integrity helps developers make the right choice in less time when making modifications to existing code bases.</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>Developer resources are always at a premium, so architecture may be seen as less critical than another feature addition to an application about to ship. However, this short term gain can lead to long term pain, as poorly architected applications become expensive to maintain, accelerating the decline and eventual demise of the applications themselves.</td>
</tr>
<tr>
<td><strong>How architectural integrity can help</strong>: Architectural integrity offers development organizations a significant long term benefit not only because it saves developer time/cost, but also because it makes code bases easier to change, enabling development teams and lines of business to become far more agile.</td>
</tr>
<tr>
<td><strong>Developer Skill</strong></td>
</tr>
<tr>
<td>Developer skill varies across a wide array of aptitudes; some programmers are exceptional at creating abstractions, while some excel at navigating complex legacy code. Because of these different predilections, application architecture as a discipline will come more naturally to some than others. Due to this variance, unless centrally enforced, code interfaces can and will be implemented in inconsistent ways based on developer skills.</td>
</tr>
<tr>
<td><strong>How architectural integrity can help</strong>: An explicit, well understood architecture not only helps less skilled developers make good (consistent) decisions, but also facilitates coaching on sound disciplines between senior developers (architects) and developers.</td>
</tr>
<tr>
<td><strong>Churn</strong></td>
</tr>
<tr>
<td>Turnover in software development teams is a fact of life, but it can wreak havoc on the productivity of a software project given the difficulty in ramping a new developer on an overly complex software system.</td>
</tr>
<tr>
<td><strong>How architectural integrity can help</strong>: If code is developed within a consistent architecture, when developers inevitably exit a company, they will leave behind cleaner, architecturally-conformant, and well-organized code that will be more readily understood by incoming developers. By this means, architectural integrity can help teams maintain productivity in the face of developer turnover.</td>
</tr>
<tr>
<td><strong>Application Size</strong></td>
</tr>
<tr>
<td>The complexity of an application (and its resulting value to a development organization) increases non-linearly with the size of the code base under consideration. For larger code bases that include over 500,000 lines of code, managing architecture is significantly more challenging than managing a small code base with less than 50,000 lines of code.</td>
</tr>
<tr>
<td><strong>How architectural integrity can help</strong>: A good, consistently enforced architecture can significantly reduce the size of the code base by facilitating reuse and reducing duplication.</td>
</tr>
</tbody>
</table>
Assessing Architectural Complexity

Regardless of the application environment (such as embedded or web applications) or purpose (such as networking, military or gaming), experienced developers all know that complexity is an enemy to performance, security and stability. Architectural complexity can be as dangerous to the success of a software project as excessively complex and defect-prone code.

Understanding architectural complexity in the context of a given software system requires some means of quantification. Quantifying complexity is necessary because it can provide an objective measure of comparison between packages, folders, files and other code constructs. This comparison also allows senior developers and application architects to understand where repair, refactoring or re-architecting efforts are required to prevent quality defects and security vulnerabilities. There are two measures of evaluating the complexity of application architecture: ‘Tangle’ and ‘Fat’.

‘Tangle’ and Application Architecture

A software system is tangled when it contains dependencies that may cause unintended consequences for one code component when a second component is modified. In the dependency graphs below, code is represented visually to illustrate the dependencies that exist between different code components before they are assembled into an application. In Figure 1, all of the dependencies are downward facing, which ensures there are no dependencies that can compromise the quality or security of the application. By contrast, in Figure 2, a tangle of four packages is visually presented.

Good vs. Bad Complexity

As you tackle software complexity, you must distinguish between essential and accidental complexity. These ideas have circulated around software development for decades, yet they are often overlooked. In software, there are often essential complexities – complexities that can’t be avoided. For example, building a rocket ship is tremendously complicated, the software required to run it would need massive sophistication which would introduce complexity.

However, there are other complexities that are not necessary to the construction of your software – accidental complexities. For example, building a rocket ship is tremendously complicated, the software required to run it would need massive sophistication which would introduce complexity.

For example, if unnecessary and unwanted dependencies are introduced because a developer doesn’t understand the intended architecture of an application, you have accidental complexity. Accidental complexity leads to fragile code, so as you work to improve the structure of your application, be sure to invest your time reducing the bad complexity of your software, not the good.
Figure 1: Example of dependency map with no bad dependencies

Figure 2: Example of dependency map with dangerous dependencies
The presence of circular or upwards dependencies in Figure 2 introduces unnecessary challenges to the developers responsible for maintaining this code base. Too much Tangle in an application’s architecture causes the following problems:

- **More complex testing** – When there are excessive dependencies, more code components are required to be linked for effective testing. For example, if a code base is layered, the QA team can test each component with its dependencies from the lower to the higher, so that they are always testing against tested components. When cyclic dependencies are present, developers are required to test and reuse all components together.

- **Excessive integration** – When components do not have clean, well defined interfaces, developers can be tempted to copy and paste the functionality they need, instead of referencing the existing copy. This leads to duplication of effort in maintaining that code, and the risk of bug fixes not being propagated to all of the copies of the code.

- **Inflexible release dates** – Each code component in a tangle must be released at the same time because of the dependencies they share. This limits the flexibility of developers to release individual code components and slows overall development efforts.

- **Version control issues** – Developers working on code within a tangle will clash over version issues because their files are essentially tethered to the files of other developers.

### ‘Fat’ and Application Architecture

At any point in software design, it’s possible to introduce too much complexity. As a general rule, you should attempt to limit complexity to “mind-sized chunks” so each component can be individually viewed and understood. Theoretically, there is never a reason that complexity should grow beyond a defined level. ‘Fat’ is a representation of complexity because it measures the number of dependencies within a code base. Many developers are familiar with Cyclomatic Complexity, a well established measure of complexity at the function level. To provide context for the definition of Fat in application architecture, Fat can be thought of as Cyclomatic complexity applied to all levels of a software system, not just the function level.
A software system may be said to ‘Fat’ when its structure contains too much code at a given level of the code-base hierarchy. In the graphics below, code components are presented visually to illustrate how an individual level of a code-base hierarchy can become Fat. An example of lean hierarchical structure can be seen in Figure 3. For comparison, an example of a Fat hierarchical structure is presented in Figure 4.

**Figure 3: Example of dependency map with lean hierarchical structure**

![Figure 3: Example of dependency map with lean hierarchical structure](image1)

**Figure 4: Example of dependency map with fat hierarchical structure**

![Figure 4: Example of dependency map with fat hierarchical structure](image2)
If a dependency graph isn’t readable (such as the one in Figure 4), it’s probably not fair to expect a developer to understand the architecture of the code base it represents. Another way to think of ‘fat’ in application architecture is that it represents a lack of structure in the underlying code. Coverity’s Architecture Analysis capability (discussed in more detail later) has auto-partitioning that is designed specifically to help find the inherent structure of a given code base. Too much ‘Fat’ in an application’s architecture causes the following problems:

• Fat code is difficult to understand due to the challenge of understanding the functionality of each component. A developer that wants to understand fat code must gradually build an understanding of all components before truly understanding any single piece. This ‘chicken-and-egg’ problem is often solved through multiple, expensive iterations.

• Fat code is hard to visualize, and therefore hard to explain to other developers.

• Fat code is hard to modify because it lacks clean abstractions, this means changes can have unexpected effects on a large portion of the system.

Fat vs. Tangle – Balancing Application Architecture

One fundamental challenge every developer and architect faces regarding application architecture is how to balance between fat and tangle. Generally, an application should allow a degree of cyclic dependency (e.g., between classes and functions), but these tangles should be of a manageable size. Tangles should not be allowed to grow to the point where they span higher level components (e.g., packages). Controlling Fat directly addresses the first of these (e.g., between classes and functions) by limiting size. However, if the Fat of a code base is not monitored, developers risk introducing package level tangles by placing multiple tangled classes in one large, fat package (or hiding the complexity in large classes or large functions).

It is important to have a single view into the combined weighting of Fat and Tangle in a measurement framework across a code base so you can objectively assess the balance of both variables. An example of this view as delivered by Coverity Architecture Analysis can be seen in Figure 5.
In the design of architecture, overcompensating to control either Fat or Tangle will increase the likelihood of problems arising because of an excess of the other variable. For example, to control tangle in a software system, a developer may decide to continually create new code components that have downward dependencies to those above. If this occurs too often, development teams will lose the ability to reuse code, because doing so introduces too many difficult to track dependencies.

Similarly, when considering the problems of Fat in application architecture, some developers may think combining multiple code components at the same hierarchical level into a single larger component will help eliminate potential problems. Coverity recommends against overreliance on this approach because over time, the temptation for other developers to reuse a single component from within this larger component will grow. When this reuse occurs, the connected code from within this larger component becomes unnecessarily enmeshed, bloating the application with code that is never used, making it more difficult to maintain.
Because application architecture plays a central role in the development of high quality, secure and maintainable code, Coverity developed a capability to automate the analysis of application architecture. Coverity Architecture Analysis enables the automatic analysis and visualization of architectural structure and dependencies in large, C/C++ and Java code bases. By making software architecture easy to define, communicate and control, it allows developers to assign limits to the structural complexity of their code, to measure it at every level and highlight code blocks that are excessively complex. By defining application architecture and how code should be structured, teams will more easily understand the intended architecture of a given application, how to build towards it, and how to efficiently maintain it.

Coverity Architecture Analysis enables teams to bridge the gap between how a code-base is structured and how it should be structured, allowing development teams to evolve code in accordance with its original, intended architecture. By graphically presenting an arrangement of components within the software system, Coverity Architecture Analysis allows developers to convey a number of architectural rules in a highly condensed format. An example of this type of diagram is presented in Figure 6 below.

**Figure 6: Cell Diagram of Code Components**
Coverity Architecture Analysis leverages Coverity’s patented Software DNA Map analysis system, the same technology that has allowed Coverity’s static analysis product to consistently deliver the most accurate results in the industry. The Software DNA Map captures a high fidelity representation of a software system to enable deep analysis of a given software system prior to run time. Because of this, the capabilities of Coverity Architecture Analysis consistently deliver exceptionally accurate results that developers and architects can trust to help them make the right decisions regarding the evolution of their applications. Also, there is no additional implementation overhead for applying Coverity Architecture Analysis for those who already leverage Coverity’s static analysis solutions.

Coverity Architecture Analysis promotes shared higher-level abstractions of any given code base, allowing senior developers and managers to tackle the architectural complexity challenges of managing and evolving the large code bases. Key capabilities include:

**Automatic Visibility into Application Architecture and Dependencies**

- View dependency graphs as diagrams or matrices at the function, method, type, and variable level
- Discover and analyze the source of any dependency, tangle or unused code cluster at any application level such as: method to method, function to type, class to class, package to class, and more
- View structural differences over time and analyze dependencies at specific code-base levels such as class, folder or design
- Automatically find hidden structure in large graphs and isolate tangled items and natural clusters by auto-partitioning
- Discover everything an item uses (or is used by) at any level

---

**Dangerous Software Dependencies**

The following dependencies are potentially dangerous to your software:

**Cyclical dependencies**

- When two or more components of a software system are interdependent
- Developers cannot extract components for reuse in other parts of the software system

**Upward dependencies**

- When a software component that is lower-level in the system depends on a higher level component
- When a low level component cannot be reused without bringing in the higher-level component, which may provide functionality that is not needed in other applications

**Abstraction violation dependencies**

- When a dependency skips a level in the hierarchy of the software system, bypassing desired functionality
- Security vulnerabilities may be introduced if an authentication layer is bypassed
Define and Automatically Enforce Application Architecture

- Design and publish architecture block diagrams to shared repository for distribution via web and IDE plug-ins
- Apply structural transformations using simple mapping expressions to transform the model hierarchy such as isolating test code or combining APIs
- Set limits for structural complexity at any application level and monitor excessive complexity across projects and over time
- Accurately predict the impact of code changes such as replacing a library before a single line of code is modified
- Send IDE-based warnings directly to programmers if code changes violate intended architecture
- Notify developers via RSS when new violations occur that enter into a software build

Coverity Architecture Analysis is part of the Coverity Integrity Center, and is a natural compliment to Coverity’s other precision software analysis capabilities which include Coverity Build Analysis, Coverity Dynamic Analysis, and Coverity Prevent, the industry-leading static analysis product. The combination of Coverity’s precision software analysis offerings enables customers to identify quality, performance and security flaws at each step in the software development process.

Who Benefits from Architectural Integrity?

Depending on your role in a software organization, ensuring the integrity of your application will have different benefits:

- **Development managers** - Can follow a code structure that supports their development processes (e.g., team to component, release, test and so on) rather than undermining it.
- **Architects** - Can plan and enforce the architecture of new applications, or discover the current architecture of legacy applications. Regardless of whether the application is new or legacy, visibility allows an architect to communicate the architectural direction to the development team so they know when the code base deviates from its intended design.
- **Developers** - Can know when they violate the application’s architecture prior to code check in, not after. Additionally, Developers will know more quickly where exactly they should add or modify code.
- **QA teams** - Can assure aspects of an application they previously lacked visibility into.
- **Senior executives or line of business managers** - Gain improved developer efficiency and productivity. Gain business agility, because an application with a consistently used architecture is easier to change or modify based on market demand.
Development organizations face many challenges that impair their ability to monitor and ensure the integrity of architecture as code bases are created and evolve over time. While always a challenge for waterfall environments, agile development introduces yet another force of complexity that can warp the intended architecture of an application over time to the frequency of releases and incremental, yet constant, change.

To protect both developer productivity and an organization’s investment in creating code, development teams must have the ability to visualize and enforce application architecture. As a discipline, application architecture should be a priority for senior developers and application architects alike, as it provides a common direction that delivers both long and short term benefits to any development team. Application architecture provides these senior team members with a consistent direction to share with more junior developers. Additionally, it can also serve as a valuable touch-point to share with enterprise architects within a company, allowing senior IT and development leaders to ensure alignment of their efforts.

Ensuring the consistency of application architecture over time offers significant benefits to development organizations. It will identify issues and provide a framework that ensures future development will be of high quality. Coverity Architecture Analysis can be a powerful tool to help senior developers ensure the architectural integrity of their software and improve the overall productivity of their teams.

Coverity Architecture Analysis is available as part of the Coverity Integrity Center, which includes Coverity Prevent, the industry-leading static analysis product, in addition to Architecture Analysis, Build Analysis and Dynamic Analysis capabilities. Together, these advanced analysis products help software organizations achieve business goals by lowering development costs, reducing customer found defects and accelerating time to market of new applications.

Request a Trial

Request a free Coverity trial and see first hand how to rapidly detect and remediate serious defects and vulnerabilities. No changes to your code are necessary. There are no limitations on code size, and you will receive a complimentary report detailing actionable analysis results. Register for the on-site evaluation at www.coverity.com or call us at (800) 873-8193.
About Coverity

Coverity (www.coverity.com), the software integrity company, is the trusted standard for companies that have a zero tolerance policy for software failures, problems, and security breaches. Coverity’s award winning portfolio of software integrity products helps customers prevent software problems throughout the application lifecycle. Over 100,000 developers and 500 companies including ARM, Phillips, RIM, Rockwell Collins, Samsung and UBS rely on Coverity to help them ensure the delivery of superior software. Coverity is a privately held.