Chapter 6. Measuring and Governing Architecture Characteristics

Architects must deal with an extraordinarily wide variety of architecture characteristics across all different aspects of software projects. Operational aspects like performance, elasticity, and scalability commingle with structural concerns, such as modularity and deployability. It benefits architects to understand how to measure and govern architectural characteristics, rather than drown in ambiguous terms and broad definitions. This chapter focuses on concretely defining some of the more common architecture characteristics and discusses how to build governance mechanisms for them.

Measuring Architecture Characteristics

Architects struggle to define architectural characteristics for a number of reasons:

They aren't physics

> Many architecture characteristics in common usage have vague meanings. For example, how does an architect design for *agility* or *deployability*? What about *wicked fast performance*? People around the industry have wildly differing perspectives on common terms—sometimes driven by legitimate differing contexts, sometimes accidental.

Wildly varying definitions

> Even within the same organization, different departments may disagree on the definitions of critical characteristics such as *performance*. Until developers, architects, operations, and others can unify on a common definition, how can they have a proper conversation?

Too composite

> Many desirable architecture characteristics are really collections of other characteristics at a smaller scale, as you might remember from our discussion of composite architectural characteristics in Chapter 5. For example, agility breaks down into characteristics such as modularity, deployability, and testability.

Decomposing composite architectural characteristics into their constituent parts is an important part of establishing objective definitions for architecture characteristics, which solves all three of these problems.

When an organization agrees that everyone will use standard, concrete definitions for architecture characteristics, they create a *ubiquitous language* around architecture. This standardization allows them to unpack composite characteristics to uncover *objectively measurable* features.

Operational Measures

Many architecture characteristics have obvious direct measurements, such as performance or scalability. Yet even these offer many nuanced interpretations, depending on the team's goals. For example, perhaps your team measures the average response time for certain requests—a good example of a measure for an operational architecture characteristic. But if your team only measures the average, what happens if some boundary condition causes 1% of requests to take 10 times longer than others? If the site has enough traffic, the outliers may not even show up. To catch outliers, you might also want to measure the maximum response times.

High-level teams don't just establish hard performance numbers; they base their definitions on statistical analysis. For example, say a video streaming service wants to monitor scalability. Rather than set an arbitrary number as the goal, the engineers measure the scale over time and build statistical models, then raise alarms if the real-time metrics fall outside the prediction models. If they do, the failure can mean two things: the model is incorrect (which teams like to know) or something is amiss (which teams also like to know).

The kinds of characteristics that teams measure evolve rapidly in conjunction with tools, targets, devices, and capabilities. For example, recently many teams have focused on performance budgets for metrics such as *first contentful paint* and *first CPU idle*, both of which speak volumes about performance issues for web page users on mobile devices. As these and myriad other things change, teams will find new things and ways to measure them.

The Many Flavors of Performance

Many of the architecture characteristics we describe have multiple definitions. Performance is a great example. Many projects look at general performance: for example, how long request and response cycles take for a web application. However, through tremendous work, architects and DevOps engineers in many organizations have established specific performance budgets for specific parts of an application. For example, many organizations have researched user behavior and determined that the optimum time for first-page render (the first visible sign of progress for a web page in a browser or mobile device) is some fraction of a second. Most applications fall in the double-digit range for this metric. But for modern sites attempting to capture as many users as possible, this is an important metric to track, and the organizations behind such sites have built extremely nuanced measures.

Some of these metrics have additional implications for application design. Many forward-thinking organizations place *K-weight budgets* for page downloads; that is, they allow a maximum number of bytes' worth of libraries and frameworks on a particular page. Their rationale derives from physics constraints: only so many bytes can travel over a network at a time, especially for mobile devices in low-bandwidth areas.

Structural Measures

Some objective measures are not so obvious as performance. What about internal structural characteristics, such as well-defined modularity? This is a great example of an implicit architectural characteristic—architects are responsible for defining components and interactions, and want to build a sustainable structure for overall

high quality. Unfortunately, there are not yet any comprehensive metrics to assess the quality of an architecture. However, there are metrics and common tools that allow architects to address some critical aspects of code structure, albeit along narrow dimensions.

One measurable aspect of code is complexity, defined by the *Cyclomatic Complexity* metric.

Cyclomatic Complexity

Cyclomatic Complexity (CC) is a code-level metric designed by Thomas McCabe Sr., in 1976 to provide an objective measure for the complexity of code at the function/method, class, or application level. It is computed by applying graph theory to code—specifically, *decision points*, which cause different execution paths. For example, if a function has no decision statements (such as `if` statements), then `CC = 1`. If the function has a single conditional, then `CC = 2` because there are two possible execution paths.

The formula for calculating the CC for a single function or method is $CC = E - N + 2$CC = E - N + 2, where *N* represents *nodes* (lines of code), and *E* represents *edges* (possible decisions). Consider the C-like code shown in Example 6-1.

Example 6-1. Sample code for Cyclomatic Complexity evaluation

```
public void decision(int c1, int c2) {
    if (c1 < 100)
        return 0;
    else if (c1 + C2 > 500)
       return 1;
    else
      return -1;
}
```

The Cyclomatic Complexity for Example 6-1 is 3 (3 – 2 + 2), shown in Figure 6-1.

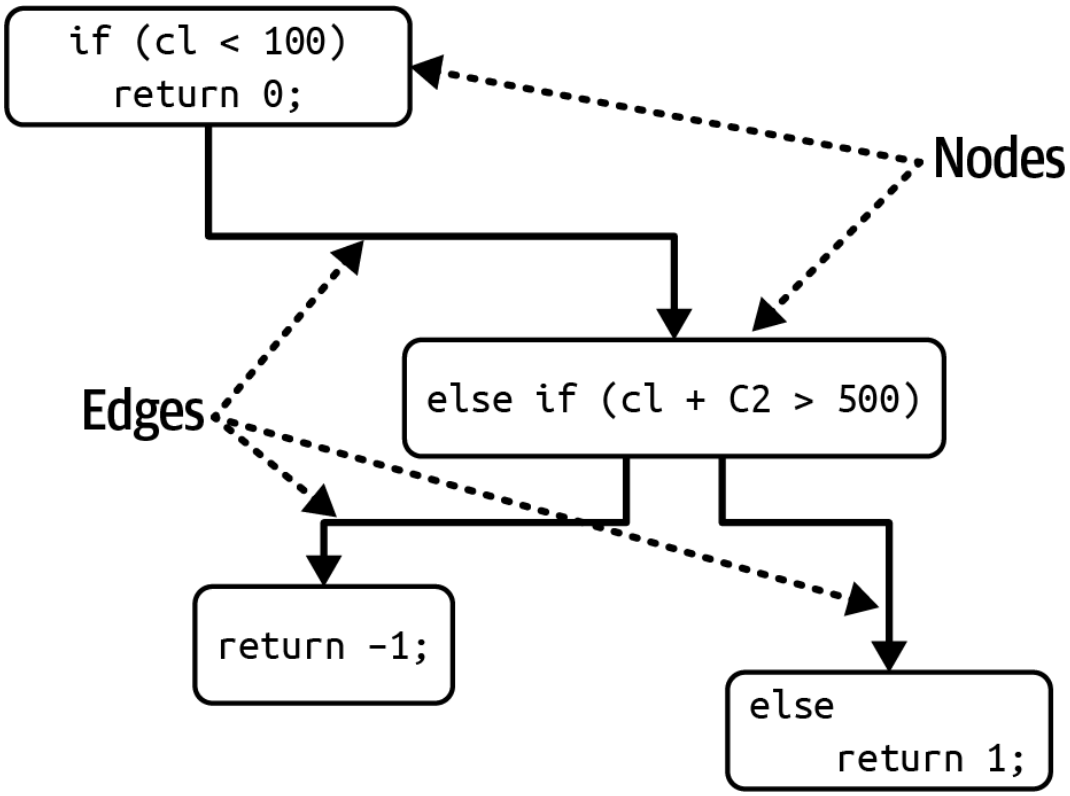

Figure 6-1. Cyclomatic Complexity graph for the decision function

The number 2 appearing in the Cyclomatic Complexity formula represents a simplification for a single function/method. For fan-out calls to other methods (known as *connected components* in graph theory), the more general formula is $CC = E - N + 2PCC = E - N + 2P$, where P represents the number of connected components.

Architects and developers universally agree that overly complex code represents a "code smell"—something that appears in code that is so bad that it has an imaginary odor. It harms virtually every one of the desirable characteristics of code bases: modularity, testability, deployability, and so on. If teams don't keep an eye on gradually growing complexity, it will dominate the code base.

Cyclomatic Complexity is a great example of the bluntness of the metrics that architects have available. While it measures the complexity of code, it cannot determine whether that complexity is *essential* (because we're solving a complicated problem) or *accidental* (because we've implemented a poor design). Metrics like CC are extremely useful for assessing code, whether it's written by developers or generative AI. Generative AI tends to solve problems by brute force, often leading to accidental complexity.

What's a Good Value for Cyclomatic Complexity?

A common question the authors receive when talking about this subject is: what's a good threshold value for CC? Of course, like all questions in software architecture, the answer is: it depends! Specifically, it depends on the complexity of the problem domain. For example, if you have an algorithmically complex problem, the solution will yield complex functions. Some of the key aspects of CC for architects to monitor: are functions complex because of the problem domain or because of poor coding? Alternatively, is the code partitioned poorly? In other

words, could a large method be broken down into smaller, logical chunks, distributing the work (and complexity) into more well-factored methods?

In general, the industry thresholds for CC suggest that a value under 10 is acceptable, barring other considerations such as complex domains. We consider that threshold very high and would prefer code to fall under five, indicating cohesive, well-factored code. A metrics tool in the Java world, Crap4J, attempts to determine how poor (crappy) your code is by evaluating a combination of CC and code coverage; if CC grows to over 50, no amount of code coverage rescues that code from crappiness. The most terrifying professional artifact Neal ever encountered was a single C function that served as the heart of a commercial software package whose CC was over 800! It was a single function with over 4,000 lines of code, including liberal use of `GOTO` statements (to escape impossibly deeply nested loops).

Engineering practices like test-driven development (TDD) have the beneficial side effect of generating smaller, less complex methods on average for a given problem domain. When practicing TDD, developers try to write a simple test, then write the smallest amount of code to pass the test. This focus on discrete behavior and good test boundaries encourages well-factored, highly cohesive methods that exhibit low CC.

Process Measures

Some architecture characteristics intersect with software development processes. For example, agility often appears as a desirable feature. However, it is a composite architecture characteristic, made up of features such as testability and deployability.

Testability is measurable through code-coverage tools for virtually all platforms that report on what percentage of the code the tests execute. Like all software checks, though, these cannot replace thinking and intent. For example, a code base can have 100% code coverage, yet use poor assertions that don't actually provide confidence in the code's correctness.

Testability is an objectively measurable characteristic, as is deployability. Deployability metrics include percentage of successful deployments, how long deployments take, and issues/bugs raised by deployments. Every team must arrive at a good set of measurements that capture useful qualitative and quantitative data for their organization's and team's priorities and goals.

While agility and its related parts clearly relate to the software development process, that process may influence the structure of the architecture. For example, if ease of deployment and testability are high priorities, the architect would emphasize good modularity and isolation at the architecture level—an example of an architecture characteristic driving a structural decision. Virtually anything within the scope of a software project can rise to the level of an architecture characteristic if it manages to meet our three criteria, forcing an architect to make significant decisions to account for it.

Governance and Fitness Functions

Once architects establish and prioritize architecture characteristics, how can they make sure that developers will respect those priorities and implement their designs correctly and safely, regardless of schedule pressure? On many software projects, urgency dominates, yet architects still need tools and techniques to provide

architectural *governance*. Modularity is a great example of an aspect of architecture that is important but not urgent.

Governing Architecture Characteristics

Governance, derived from the Greek word *kubernan* (to steer) is an important responsibility of the architect role. As the name implies, its scope covers any aspect of the software development process that architects want to influence. For example, ensuring software quality falls under architectural governance because neglecting it can lead to disastrous quality problems.

Fortunately, architects have increasingly sophisticated solutions to this problem—a good example of incremental growth within the software development ecosystem's capabilities. The drive toward automation spawned by Extreme Programming created continuous integration (CI). CI led to further automation in operations, which we now call DevOps, and this chain continues through to architectural governance. The book *Building Evolutionary Architectures* (O'Reilly, 2022) by Neal Ford et al. describes a family of techniques called *fitness functions* used to automate many aspects of architecture governance. We'll spend the rest of this chapter looking at fitness functions.

Fitness Functions

The word *evolutionary* in the title *Building Evolutionary Architectures* comes more from evolutionary computing than from biology. One of the authors, Dr. Rebecca Parsons, spent some time in the evolutionary computing space, including working with tools like genetic algorithms. When a developer designs a genetic algorithm to produce some beneficial outcome, they often want to guide the algorithm by providing an objective measure of the quality of its outcome. That guidance mechanism is called a *fitness function*: an object function used to assess how close the output comes to achieving its aim.

For example, suppose you need to solve the traveling salesperson problem, a famous problem used as a basis for machine learning. Given a salesperson, a list of cities they must visit, and the distances between those cities, what is the optimum possible route—minimizing distance, time, and cost? If you design a genetic algorithm to solve this problem, you might use one fitness function to evaluate the length of the route and another to evaluate the overall cost associated with the route. Yet another might evaluate the time the traveling salesperson is away.

Practices in evolutionary architecture borrow this concept to create an *architectural fitness function*: any mechanism that provides an objective integrity assessment of some architecture characteristic or combination of architecture characteristics.

Fitness functions are not some new framework for architects to download. Rather, they offer a new perspective on many existing tools. Notice in the definition the phrase *any mechanism*—the verification techniques for architecture characteristics are as varied as the characteristics are. Fitness functions overlap many existing verification mechanisms, depending on the way they are used: in chaos engineering or as metrics, monitors, or unit testing libraries (see Figure 6-2).

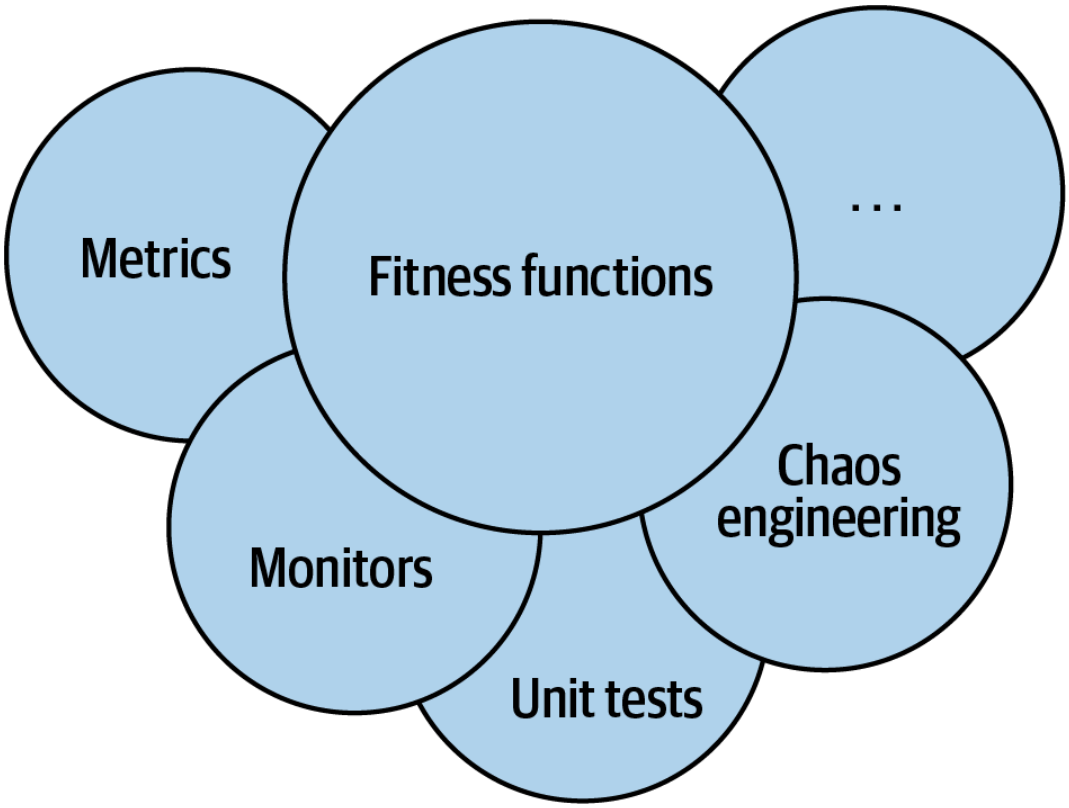

Figure 6-2. The mechanisms of fitness functions

Many different tools may be used to implement fitness functions, depending on the architecture characteristics. Let's look at a couple of examples of fitness functions that test various aspects of modularity.

Cyclic dependencies

Modularity is an implicit architecture characteristic that most architects care about. Because poorly maintained modularity harms the structure of a code base, architects generally place a high priority on maintaining good modularity. However, on many platforms, there are forces working against those good intentions. For example, when coding in any popular Java or .NET development environment, as soon as a developer references a class not already imported, the IDE helpfully presents a dialog asking if they would like to auto-import the reference. This occurs so often that most programmers reflexively swat the auto-import dialog away. But arbitrarily importing classes between components can spell disaster for modularity. For example, Figure 6-3 illustrates *Cyclic Dependencies*, a particularly damaging antipattern that architects aspire to avoid.

In Figure 6-3, each component references something in the other components. Having a network like this damages modularity, because it's impossible to reuse a single component without bringing the others along. And if those other components are coupled to other components? The architecture will tend more and more toward the Big Ball of Mud antipattern. How can architects govern this behavior without constantly looking over the developers' shoulders? Code reviews help, but happen too late in the development cycle to be fully effective. If the development team imports rampantly across the code base for a week until the code review, they've already done serious damage to the code base.

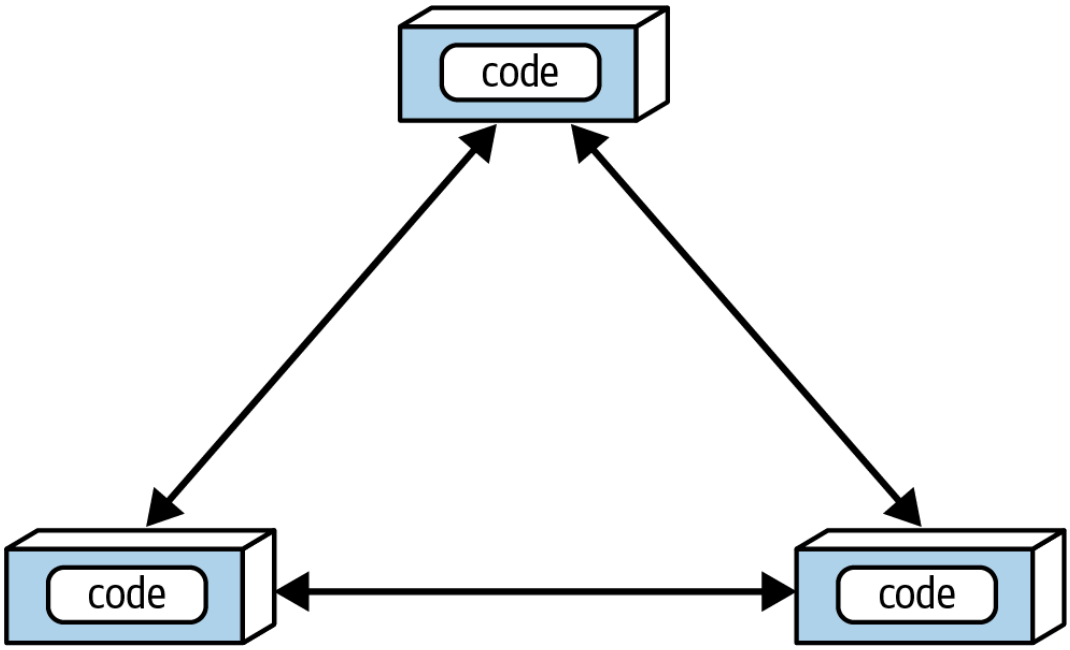

Figure 6-3. Cyclic dependencies between components

The solution to this problem is to write a fitness function to look after cycles, as shown in Example 6-2.

Example 6-2. Fitness function to detect component cycles

```
public class CycleTest {
    private JDepend jdepend;

    @BeforeEach
    void init() {
      jdepend = new JDepend();
      jdepend.addDirectory("/path/to/project/persistence/classes");
      jdepend.addDirectory("/path/to/project/web/classes");
      jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
      Collection packages = jdepend.analyze();
      assertEquals("Cycles exist", false, jdepend.containsCycles());
    }
}
```

This code uses the metrics tool JDepend to check the dependencies between packages. The tool understands the structure of Java packages and fails the test if it finds any cycles. An architect can wire this test into a project's continuous build on and stop worrying about trigger-happy developers accidentally introducing cycles. This is a great example of a fitness function guarding the *important rather than urgent* practices of software development: it's an important concern for architects, yet has little impact on day-to-day coding.

Distance from the Main Sequence fitness function

In "Coupling", we introduced the more esoteric metric of *Distance from the Main Sequence*, which can also be verified using fitness functions, as shown in Example 6-3.

Example 6-3. Distance from the Main Sequence fitness function

```
@Test
void AllPackages() {
    double ideal = 0.0;
    double tolerance = 0.5; // project-dependent
    Collection packages = jdepend.analyze();
    Iterator iter = packages.iterator();
    while (iter.hasNext()) {
      JavaPackage p = (JavaPackage)iter.next();
      assertEquals("Distance exceeded: " + p.getName(),
        ideal, p.distance(), tolerance);
    }
}
```

This code uses JDepend to establish a threshold for acceptable values, failing the test if a class falls outside the range. (The tool ArchUnit, highlighted in the next section, allows architects to create similar fitness functions.) This example of an objective measure for an architecture characteristic illustrates the importance of collaboration between developers and architects when designing and implementing fitness functions. The intent is not for a group of architects to ascend to an ivory tower and develop esoteric fitness functions that developers cannot understand—it's to implement automated governance rules that ensure the quality of the code base.

Tip

Architects must ensure that developers understand the purpose of a fitness function before imposing it on them.

The sophistication of fitness function tools has increased over the last few years, especially as some special-purpose tools have emerged. One such tool is ArchUnit, a Java testing framework inspired by—and using—several parts of the JUnit ecosystem. ArchUnit provides a variety of predefined governance rules, codified as unit tests, and allows architects to write specific tests that address modularity. Consider the layered architecture illustrated in Figure 6-4.

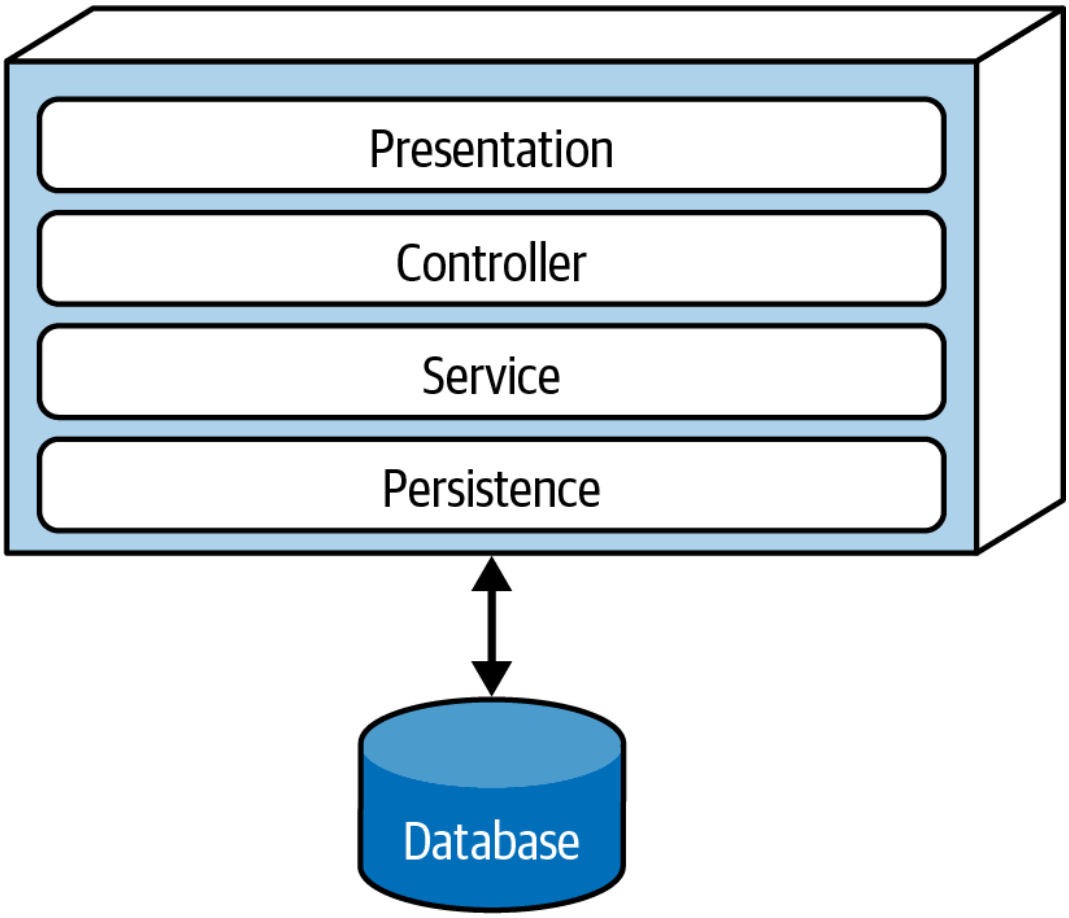

Figure 6-4. Layered architecture

When designing a layered monolith such as the one in Figure 6-4, the architect defines the layers for good reasons (we describe these motivations, trade-offs, and other aspects in Chapter 10). However, some developers may not understand the importance of these patterns, while others may adopt a "better to ask forgiveness than permission" attitude because of some overriding local concern, such as performance. But allowing them to erode the reasons for the architecture will hurt the long-term health of the architecture.

ArchUnit allows architects to address this problem via a fitness function, shown in Example 6-4.

Example 6-4. ArchUnit fitness function to govern layers

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

In Example 6-4, the architect defines the desirable relationship between layers and writes a verification fitness function to govern it.

There is a similar tool in the .NET space, NetArchTest. Example 6-5 shows a layer verification in C#.

Example 6-5. NetArchTest for layer dependencies

```
// Classes in the presentation should not directly reference repositories
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

Our discussion of testability highlights a problem with any metric or measurement: the possibility that developers will try to game the system. Once they learn how architects are measuring compliance, they may code to the metric rather than build the correct thing. For example, a common failing of testability occurs when developers who are taking shortcuts write unit tests, but without any assertions. “Touching” the code but not verifying that it works amounts to cheating on the code-coverage metric. Writing governance code using tools like ArchUnit helps architects prevent this behavior by ensuring that every unit test includes at least one assertion. Of course, dedicated rule-breakers will always find a way, but fitness functions like this prevent accidental lapses.

Another example of fitness functions is Netflix’s “Chaos Monkey” and the attendant Simian Army. When Netflix decided to move its operations to Amazon’s cloud, the architects no longer had control over operations, which worried them: what would happen if a defect appeared operationally? To solve this problem, they spawned the discipline of *chaos engineering*. The Chaos Monkey essentially acts as a production fitness function, simulating general chaos to see how well the system can endure it. Latency was a problem with some AWS instances, so the Chaos Monkey would simulate high latency. (In fact, this was such a problem, they eventually created a specialized Latency Monkey). This led them to develop additional tools: for instance, the Chaos Kong, which simulates an entire Amazon datacenter failure, has helped Netflix avoid such outages when they occur for real.

In particular, the *Conformity*, *Security*, and *Janitor* “Monkeys” (fitness functions) exemplify the automated-governance approach. The Conformity Monkey allows Netflix architects to define governance rules enforced by the monkey in production. For example, if they decide that each service should respond without errors for all requests, they build that check into the Conformity Monkey. The Security Monkey checks each service for well-known security defects, like ports that shouldn’t be active and configuration errors.

Finally, the Janitor Monkey looks for instances to which no other services route anymore. Netflix has an evolutionary architecture, so developers routinely migrate to newer services, leaving old services running with no collaborators. Because running services on the cloud consumes money, the Janitor Monkey looks for orphan services and disintegrates them out of production.

Chaos engineering offers an interesting new perspective on architecture: it’s not a question of *if* something will eventually break, but *when*. Anticipating those breakages and testing to prevent them makes systems much more robust. A book by some of the Netflix innovators, Casey Rosenthal and Nora Jones, called *Chaos Engineering* (O’Reilly, 2020) highlights this approach.

Atul Gawande’s influential book *The Checklist Manifesto* (Metropolitan, 2009) describes how professionals such as airline pilots and surgeons use checklists. (Sometimes they’re even legally mandated to do so.) It’s not because those professionals don’t know their jobs or are forgetful. Rather, when professionals do a highly detailed job over and over, it becomes easy for details to slip by them; a succinct checklist forms an effective reminder.

This is the correct perspective on fitness functions—they are not a heavyweight governance mechanism, but a mechanism for architects to express and automatically verify important architectural principles. Developers know that they shouldn’t release insecure code, but that competes with dozens or hundreds of other priorities. Tools like the Security Monkey, and fitness functions generally, allow architects to build important governance checks into the substrate of the architecture.