Chapter 4. Architectural Coupling

Discussions about architecture frequently boil down to coupling: how the pieces of the architecture connect and rely on one another. Many architects decry coupling as a necessary evil, but it's difficult to build complex software without relying on (and coupling with) other components. Evolutionary architecture focuses on appropriate coupling—how to identify which dimensions of the architecture should be coupled to provide maximum benefit with minimal overhead and cost.

Modularity

First, let's untangle some of the common terms used and overused in discussions about architecture. Different platforms offer different reuse mechanisms for code, but all support some way of grouping related code together into *modules*. *Modularity* describes a logical grouping of related code. Modules in turn may be packaged in different physical ways. *Components* are the physical packaging of modules. Modules imply *logical* grouping, while components imply *physical* partitioning.

Developers find it useful to further subdivide *components* based on engineering practices, including build and deployment considerations. One kind of component is a *library*, which tends to run in the same memory address as the calling code and communicates via language function call mechanisms. Libraries are usually compile-time dependencies. Most concerns around libraries exist in application architecture, as most complex applications consist of a variety of components. The other type of component, called a *service*, tends to run in its own address space and communicates via low-level networking protocols like TCP/IP or higher-level formats like simple object access protocol (SOAP) or representational state transfer (REST). Concerns around services tend to come up most commonly in integration architecture, making these runtime dependencies.

All module mechanisms facilitate code reuse, and it is wise to try to reuse code at all levels, from individual functions all the way up to encapsulated business platforms.

Architectural Quanta and Granularity

Software systems are bound together in a variety of ways. As software architects, we analyze software using many different perspectives. But component-level coupling isn't the only thing that binds software together. Many business concepts semantically bind parts of the system together, creating *functional cohesion*. To successfully evolve software, developers must consider *all* the coupling points that could break.

As defined in physics, the *quantum* is the minimum amount of any physical entity involved in an interaction. An *architectural quantum* is an independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function properly. In a monolithic architecture, the quantum is the entire application; everything is highly coupled and therefore developers must deploy it en mass.

DOMAIN-DRIVEN DESIGN'S BOUNDED CONTEXT

Eric Evans's book Domain-Driven Design (http://www.domaindrivendesign.org/books/evans_2003) has deeply influenced modern architectural thinking. *Domain-driven design* (DDD) is a modeling technique that allows for organized decomposition of complex problem domains. DDD defines the *bounded context*, where everything related to the domain is visible internally but opaque to other bounded contexts. Before DDD, developers sought holistic reuse across common entities within the organization. Yet, creating common shared artifacts causes a host of problems, such as coupling, more difficult coordination, and increased complexity. The *bounded context* concept recognizes that each entity works best within a localized context. Thus, instead of creating a unified `Customer` class across the entire organization, each problem domain can create their own, and reconcile differences at integration points. DDD influenced several modern architectural styles, along with related factors like team organization (described in "Introducing PenultimateWidgets and Their Inverse Conway Moment" in Chapter 1).

In contrast, a microservices architecture defines physical bounded contexts between architectural elements, encapsulating all the parts that might change. This type of architecture is designed to allow incremental change. In a microservices architecture, the bounded context serves as the quantum boundary and includes dependent components such as database servers. It may also include architecture components such as search engines and reporting tools—anything that contributes to the delivered functionality of the service, as shown in Figure 4-1.

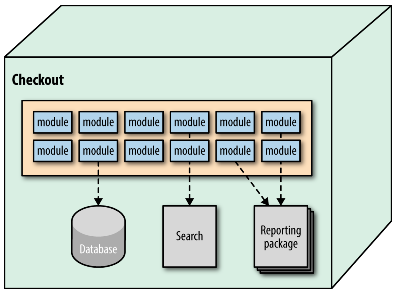

Figure 4-1. The architectural quantum in microservices encompasses the service and all its dependent parts

In Figure 4-1, the service includes code components, a database server, and a search engine component. Part of the bounded context philosophy of microservices operationalizes all the pieces of a service together, leaning heavily on modern DevOps practices. In the following section, we investigate some common architectural patterns and their typical quantum boundaries.

Traditionally isolated roles such as architect and operations must coordinate in an evolutionary architecture. Architecture is abstract until operationalized; developers must pay attention to how their components fit together in the real world. Regardless of which architecture pattern developers choose, architects should also explicitly define their quantum size. Small quanta implies faster change because of small scope. Generally, small parts are easier to work with than big ones. Quantum size determines the lower bound of the incremental change possible within an architecture.

As in physics, four fundamental interactions exist in nature: *gravitational*, *electromagnetic*, *strong*, and *weak*. The *strong* nuclear force, which holds atoms (and therefore ordinary matter) together, is notable for its strength. Breaking it unleashes much of the power of nuclear fission. Similarly, some architectural components are extremely difficult to break into smaller pieces. Metaphorically, they exhibit strong nuclear force. One of the keys to building evolutionary architectures lies in determining natural component granularity and coupling between components to fit the capabilities they want to support via the software architecture.

In evolutionary architecture, architects deal with *architectural quanta*, the parts of a system held together by hard-to-break forces. For example, transactions act like a strong nuclear force, binding together otherwise unrelated pieces. While it is possible for developers to break apart a transactional context, it is a complex process and often leads to incidental complications like distributed transactions. Similarly, parts of a business might be highly coupled, and breaking the application into smaller architectural components may not be desirable.

Figure 4-2 summarizes the relationship between these terms.

MONOLITHIC LISTING

We worked on a project for several years centered around automobile auctions. Not surprising, one of the large classes in the system was `Listing`, which grew into a monster. Developers undertook several technical refactoring exercises to find ways to break up the huge class because it was causing coordination problems. Finally, a scheme was hatched to break out one of the key parts, `Vendor`, into its own class. While the technical refactoring was a success, problems emerged between the interactions by developers and business analysts: developers kept talking about changes to `Vendor`, which wasn't a separate entity in their world. Developers violated what Eric Evans in *DDD* calls *ubiquitous language* on the project—make sure that all the terms on the team mean the same thing. While it made a few things more convenient for developers to split the functionality, the semantic coupling that defined the business process was violated, making our job more difficult.

Eventually, we unrefactored the `Listing` class back into a single large entity, because the software project revolved around it. We solved the coordination problem by treating `Listing` differently. Changes to `Listing` caused the continuous integration server to automatically generate a message to interested teams to encourage aggressive integration. Thus, we solved the coordination problem with an engineering practice rather than an architectural structure.

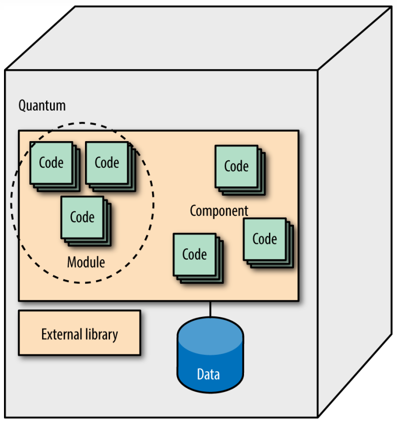

Figure 4-2. The relationship between modules, components, and quanta

As shown in Figure 4-2, the outermost container is the *quantum*: the deployable unit that includes all the facilities required for the system to function properly, including data. Within the quantum, several *components* exist, each consisting of code (classes, packages, namespaces, functions, and so on). An external component (from an open source project) also exists as a *library*, a component packaged for reuse within a given platform. Of course, developers can mix and match all possible combinations of these common building blocks.

Evolvability of Architectural Styles

Software architecture exists at least partially to enable certain types of evolution across specific dimensions—easier change is one of the reasons for architecture patterns. Different architectural patterns have different inherent quantum sizes, which impact their ability to evolve. In this section, we investigate several popular architecture patterns and evaluate their inherent quantum size, along with their impact on the architecture's natural ability to evolve based on our three evolutionary criteria: incremental change, fitness functions, and appropriate coupling.

Note that while the architectural pattern is critical for successful evolution, it isn't the only determining factor. The inherent characteristics of the pattern must be combined with the additional characteristics defined for the system to fully define the dimensions of evolvability.

Big Ball of Mud

First, consider the degenerate case of a chaotic system with no discernible architecture, colloquially known as the Big Ball of Mud antipattern. While typical architectural elements like frameworks and libraries may exist, developers haven't built structure on purpose. These systems are highly coupled, leading to rippling side effects when changes occur. Developers created highly coupled classes with poor modularity. Database schemas snaked into the UI and other parts of the system, effectively insulating them against change. DBAs spent the last decade avoiding refactoring by stitching together tightly bound join tables. Likely driven by draconian budget constraints, operations crams as many systems together as possible and deals with the operational coupling.

Figure 4-3 shows a class coupling diagram that exemplifies the Big Ball of Mud: each node represents a class, the lines represent coupling (either inward or outward) and the boldness of the line indicates the number of connections.

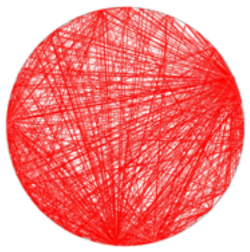

Figure 4-3. Afferent and efferent coupling for a dysfunctional architecture

Changing any part of the application depicted in Figure 4-3 (taken from a real project) presents intense challenges. Because so much exuberant coupling exists between classes, it is virtually impossible to modify one part of the application without impacting other parts. Thus, from an evolvability standpoint, this architecture scores extremely low. Developers who need to change data access throughout the application must hunt down all the places it exists and change them, risking missing some places.

From an evolution standpoint, this architecture fails each criteria drastically:

Incremental change

> Making any change in this architecture is difficult. Related code is scattered throughout the system, meaning changes to one component will cause unexpected breakages in other components. Fixing those breakages will generate more breakage, a rippling effect that never ends.

Guided change with fitness functions

> Building fitness functions for this architecture is difficult because no clearly defined partitioning exists. To build protective functions, developers must be able to identify parts to protect, and no structure exists in this architecture outside low-level functions or classes.

Appropriate coupling

> This architectural style is a good example of *inappropriate* coupling. No architectural advantages result from building software like this.

In this dire state, change is difficult and expensive. Essentially, because each part of the system is highly coupled to every other part, the quantum is the entire system—no part is easy to change because every part affects every other part.

Monoliths

Monolithic architectures often contain a large amount of highly coupled code. We investigate several variations of this architectural style, based on organization.

UNSTRUCTURED MONOLITHS

This architectural pattern includes several different variations, including systems with essentially independent classes coordinating as seen in Figure 4-4.

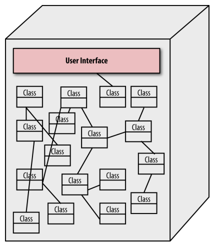

Figure 4-4. Monolith architectures sometimes contain a collection of loosely related classes

In Figure 4-4, different modules handle different tasks independently, utilizing shared classes for common functionality. A lack of coherent overarching structure hinders change in this architecture.

Incremental change

Large quantum size hinders incremental change because high coupling requires deploying large chunks of the application. Deploying a single component is difficult because each component is highly coupled to others, requiring change to those components as well.

Guided change with fitness functions

Building fitness functions for monoliths is difficult but not impossible. Because this architectural pattern has existed for a long time, many tools and testing practices have grown around it that can be used to create fitness functions. However, common guided change targets, such as performance and scalability, have traditionally been the Achilles' heel of monolithic architectures. While developers easily understand monoliths, building good scalability and performance is difficult, largely due to inherent coupling.

Appropriate coupling

A monolithic architecture, with little internal structure outside simple classes, exhibits coupling almost as bad as a Big Ball of Mud. Thus, changes in one portion of the code may have unanticipated side effects in sometimes far-reaching parts of the code base.

Though the evolvability of this architecture is a milder version of the "Big Ball of Mud". It is quite easy for this architecture to degenerate because there are few structural constraints to prevent it.

LAYERED ARCHITECTURE

Other monolith architectures utilize a more structured approach to creating a layered architecture, one variation of which appears in Figure 4-5.

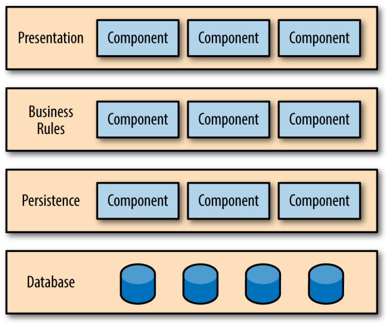

Figure 4-5. Typical layered monolith architecture

In Figure 4-5, each layer represents a technical capability, allowing developers to swap out technical architecture functionality easily. The primary design criteria for the layered architecture separates different technical capabilities into layers, each with a distinct responsibility. The primary advantages of this architecture are *isolation* and *separation of concerns*. Each layer is isolated from the others, accessible via a well-defined interface. This allows for implementation changes within the layer without affecting the other layers and grouping of similar code together, making space for specialization and separation within the layer. For example, a persistence layer typically encapsulates all implementation details of how data is saved, allowing other layers to ignore those details.

In all cases of monolith architecture, the quantum is the application, including dependent components like database servers. Evolving systems with large quantum size is difficult:

Incremental change

Developers find it easy to make some changes in this architecture, particularly if those changes are isolated to existing layers. Cross-layer changes can cause coordination challenges, especially if the organization's workforce resembles the layers of the architecture (a reflection of "Conway's Law"). For example, a team can swap one persistence framework for another with little disruption to other teams because they can perform that work behind the well-defined interface. If, on the other hand, the business is required to change something like `ShipToCustomer`, that change will affect all the layers, requiring coordination.

Guided change with fitness functions

Developers find it easier to write fitness functions in a more structured version of a monolith because the structure of the architecture is more apparent.

The separation of concerns in layers also allows developers to test more parts in isolation, making it easier to create fitness functions.

Appropriate coupling

One of the virtues of monolith architectures is easy understandability. Developers who understand concepts such as design patterns can easily apply that knowledge to layered architectures. A large portion of understandability is convenient access to all parts of the code. Layered architectures allow for easy evolution of the technical architecture partitions defined by the layers. For example, a well-designed (and implemented) layered architecture makes it easy to swap out the database, business rules, or any other layer with minimal side effects.

Monolith architectures tend to have high coupling, both intentional and unintentional. When developers use layered architectures for separation of concerns (e.g., using a persistence layer to simplify data access), the layer typically exhibits high internal and low external coupling. Within the layer, each component is cooperating towards a single goal, so they tend toward high coupling. In contrast, developers typically define the interfaces between layers more carefully, creating lower coupling between layers.

MODULAR MONOLITHS

Many of the benefits architects tout about microservices—isolation, independence, small unit of change—can be achieved in monolithic architectures…if developers are extremely disciplined about coupling. Note that this discipline must extend beyond just technical architecture, encompassing other dimensions (notably data) equally. Modern tools make code reuse so convenient that developers struggle to achieve appropriate coupling in environments where coupling is easy. Fitness functions like the one in Example 4-1 allow architects to build safety nets into their deployment pipelines to keep monolith component dependencies clean.

Most modern languages allow building strict visibility and connection rules. If architects and developers build a modular monolith using those rules, they will have a much more malleable architecture, as demonstrated by a well modularized monolith depicted in Figure 4-6.

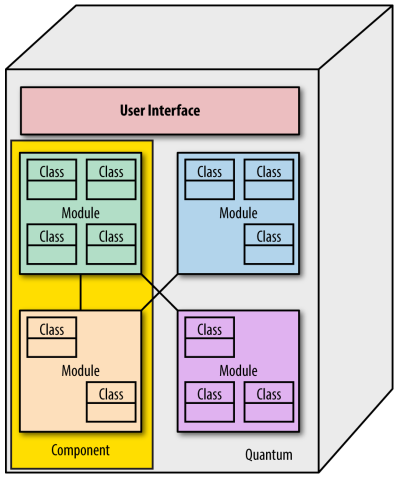

Figure 4-6. A modular monolith contains logical grouping of functionality with well-defined isolation between modules

Incremental change

Incremental change is easy in this type of architecture because developers can enforce modularity. However, despite the *logical* separation of functionality into modules, if the components containing the modules are difficult to individually deploy, the quantum size is still large. In a modular monolith, the degree of deployability of the components determines the rate of incremental change.

Guided change with fitness functions

Tests, metrics, and other fitness function mechanisms are easier to design and implement in this architecture because of good separation of components, allowing easier mocking and other testing techniques that rely on isolation layers.

Appropriate coupling

A well-designed modular monolith is a good example of appropriate coupling. Each component is functionally cohesive, with good interfaces between them and low coupling.

Monolithic architectures, particularly layered architectures, are a common choice when starting a project because developers understand the structure easily. However, many monoliths reach end of life and must be replaced because of decreasing performance, size of code base, and a host of other factors. A current common target for monolith migration is microservices-style architectures, which are more complex than monolithic architectures in areas like service and data granularity, operationalization, coordination, transactions, and so on. If a development team has a hard time building one of the simplest architectures, how will moving to a more complex architecture solve their problems?

If you can't build a monolith, what makes you think microservices are the answer?

—Simon Brown

Before embarking on an expensive architecture restructuring exercise, architects may benefit from improved modularization of what's already present. If nothing else, it's an excellent starting point for the more serious restructuring that follows.

MICROKERNEL

Consider another popular monolithic architectural style, the microkernal architecture, commonly found in browsers and integrated development environments (IDEs), as shown in Figure 4-7.

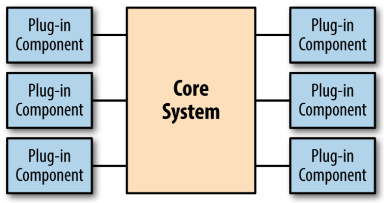

Figure 4-7. A microkernel architecture

The microkernel architecture shown in Figure 4-7 defines a core system with an API that allows plug-in enhancements. Two architectural quantum sizes exist for this architecture: one for the core system and another for the plug-ins. Architects typically design the core system as a monolith, creating hooks for well-known extension points for plug-ins. Plug-ins are usually designed to be self-contained and independently deployable. Thus, this architecture supports positive, incremental change, and developers can design for testability and facilitate fitness function definition. From a technical coupling standpoint, architects tend to design these systems with low coupling for practical reasons, which is related to keeping the plug-ins independent from one another to simplify them.

The primary challenge architects face in microkernel architectures revolves around contracts, a form of semantic coupling. To perform useful work, plug-ins must pass information in and out of the core system. As long as the plug-ins don't need to coordinate between each other, developers can focus on information and versioning with the core system. For example, most browser plug-ins interact only with the browser, not other plug-ins.

More complex microkernel systems, such as the Eclipse (http://eclipse.org) Java IDE, must support intraplug-in communication. The core of Eclipse offers no specific language support beyond interacting with text files. All complex behavior comes via plug-ins that pass information between each other. For example, the compiler and debugger must closely coordinate during a debugging session. Because the plug-ins shouldn't rely on other plug-ins to work, the core system must handle the communication, making contract coordination and common tasks like versioning complex. While this level of isolation is desirable because it makes the system less stateful, it is often not possible. For example, in Eclipse, plug-ins often require dependent plug-ins to function, creating another level of transitive dependency management around the architectural quantum of the plug-in.

Typically, microkernel architectures include a registry that tracks installed plug-ins and the contracts they support. Building explicit coupling between plug-ins increases semantic coupling between parts of the system and therefore increases the architectural quantum.

While the microkernel architecture is popular with tools such as IDEs, it is also applicable for a wide variety of business applications. For example, consider an insurance company. The standard business rules for handling claims exist companywide, yet each state may have special rules. Building this system as a microkernel allows developers to add support for new states as needed and to upgrade individual state behavior without affecting any other state because of the inherent isolation of the plug-ins.

Microkernel architectures offer reasonably good if limited opportunities to evolve the technical architecture via plug-ins. Systems with completely isolated plug-ins make evolution easier because no coupling exists between plug-ins; plug-ins that must collaborate increase coupling and therefore hinder evolution. If you design a system with interacting plug-ins, you should also build fitness functions to protect the integration points, modeled after consumer-driven contracts. The core system in microkernel architectures is typically large but stable—most changes in this architecture should occur in plug-ins (otherwise, the architect may have poorly partitioned the application). Thus, incremental change is straightforward: deployment pipelines trigger change to plug-ins to validate changes.

Architects don't traditionally include data dependencies within the technical architecture for microkernels, so developers and DBAs must consider data evolution independently. Treating each plug-in as a bounded context improves the evolvablilty of the architecture because it decreases external coupling. For example, if all the plug-ins use the same database as the core system, developers must worry about coupling occurring between plug-ins at the data level. If each plug-in is completely independent, this data coupling cannot occur.

From an evolutionary standpoint, microkernels have many desirable characteristics, including the following:

Incremental change

Once the core system is complete, most behavior should come from plug-ins, small units of deployment. If the plug-ins are independent, incremental change becomes even easier.

Guided change with fitness functions

Fitness functions are typically easy to create in this architecture because of the isolation between the core and plug-ins. Developers maintain two sets of fitness functions for systems like this: *core* and *plug-ins*. The core fitness functions guard against changes to the core, including deployment concerns like scalability. Generally, plug-in testing is simpler as the domain behavior is tested in isolation. Developers will want a good mock or stub version of the core to make testing plug-ins easier.

Appropriate coupling

The coupling characteristics in this architecture are well defined by the microkernel pattern. Building independent plug-ins makes change trivial from a coupling standpoint. Dependent plug-ins make coordination more difficult. Developers should use fitness functions to ensure dependent components integrate properly.

These architectures should also include holistic fitness functions to ensure that developers maintain key architectural characteristics. For example, individual plug-ins might affect a systematic property like scalability. Thus, developers should plan to have a suite of integration tests to act as a holistic fitness function. In systems with dependent plug-ins, developers should also have a holistic fitness function to ensure contract and message consistency.

Event-Driven Architectures

Event-driven architectures (EDA) usually integrate several disparate systems together using message queues. There are two common implementations of this type of architecture: the *broker* and *mediator* patterns. Each pattern has different core capabilities, thus we discuss the pattern and evolution implications separately.

BROKERS

In a *broker* EDA, the architectural components consist of the following elements:

message queues

Message queues implemented via a wide variety of technologies such as JMS (Java Messaging Service).

initiating event

The event that starts the business process.

intra-process events

Events passed between event processors to fulfill a business process.

event processors

The active architecture components, which perform actual business processing. When two processors need to coordinate, they pass messages via queues.

A typical broker EDA workflow is illustrated in Figure 4-8, in which a customer of an insurance company changes their address.

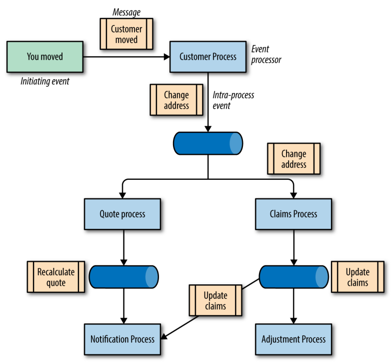

Figure 4-8. An asynchronous workflow to capture the "client moved" workflow

As seen in Figure 4-8, the initiating event is `client moved`. The first interested event processor is the `Customer process`, which updates the internal address record. Upon completion, it posts a message to a `address changed` message queue. Both the `Quote` and `Claims` processes respond to this event, updating their respective characteristics. Note that because the services need no coordination, these operations can occur in parallel, a key benefit of this architecture. Once completed, each processor posts to relevant queues, such as `Notification`.

Broker EDAs offer some design challenges when building robust asynchronous systems. For example, coordination and error handling are difficult because of a lack of a centralized mediator. Because the architectural parts are highly decoupled, developers must restore the functional cohesion of business processing to this architecture. Thus, behaviors such as transactions are more difficult.

Despite the implementation challenges, these are extremely evolvable architectures. Developers can add new behaviors to the system by adding new listeners on existing event queues, without affecting existing behavior. For example, let's say the insurance company wanted to add auditing to all claims updates. Developers can add an `Audit` listener on the `Claims` event queue without affecting the existing workflow.

Incremental change

Broker EDAs allow incremental change in multiple forms. Developers typically design services to be loosely coupled, making independent deployment easier. Decoupling in turn makes it easier for developers to make nonbreaking changes in the architecture. Building deployment pipelines for broker EDAs can be challenging because the essence of the architecture is asynchronous communication, which is notoriously difficult to test.

Guided change with fitness functions

Atomic fitness functions should be easy for developers to write in this architecture because the individual behaviors of event processors is simple. However, holistic fitness functions are both necessary and complex in this architecture. Much of the behavior of the overall system relies on the communication between loosely coupled services, making testing multifaceted workflows difficult. Consider the workflow in Figure 4-8. Developers can easily test the individual parts of the workflow by unit testing the event processors, but testing all processes is more challenging. There are a variety of ways to mitigate testing challenges in architectures like this. For example, correlation IDS, where each request is tagged with a unique identifier, helps track cross-service behavior. Similarly, synthetic transactions allow developers to test coordination logic without actually, for example, ordering washing machines.

Appropriate coupling

Broker EDAs exhibit a low degree of coupling, enhancing the ability to make evolutionary change. For example, to add new behavior to this architecture, new listeners are added to existing endpoints without affecting existing listeners. The coupling that does occur in this architecture is between services and the message contracts they maintain, a form of functional cohesion. Fitness functions using techniques like consumer-driven contracts help manage integration points and avoid breakages.

In business processes that lend themselves toward broker EDAs, the event processors are typically stateless, decoupled, and own their own data, making evolution easier because of fewer external coupling issues such as with databases, discussed in Chapter 5.

MEDIATORS

The other common EDA pattern is the *mediator*, where an additional component appears: a hub that acts as a coordinator, shown in Figure 4-9.

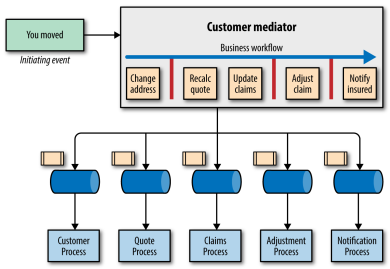

Figure 4-9. The "client moved" workflow in a mediator architecture

The mediator handles the initiating "client moved" event in Figure 4-9 and has the workflow defined within: `change address`, `recalc quotes`, `update claims`, `adjust claims`, and `notify insured`. The mediator posts messages to queues which in turn trigger the appropriate event processors. While the mediator handles coordination, this is still an EDA, enabling most of the processing to occur in parallel. For example, the `recalc quotes` and `update claims` processes run in parallel. Once all the tasks are complete, the mediator generates a message to the `notify insured` queue to generate a single status message. Any event process that needs to communicate with another processor does so via the mediator. Generally, event processors do not call one another in this type of architecture because the mediator defines important workflow information direct communication would bypass. Notice the vertical bars depicted in the mediator in Figure 4-9, indicating both parallel execution and coordination of service requests and responses.

This transactional coordination is the primary advantage of the mediator architecture. The mediator can ensure errors don't occur during the process, and generate a single status message to the insured. In a broker EDA, this type of coordination is more difficult. To generate a single notification message, for example, the coordination would occur at the `Notification` event processor or via an explicit message queue to handle this aggregation. While asynchronous architectures create challenges around coordination and transactional behavior they offer fantastic parallel scale.

Incremental change

Similar to broker EDAs, the services in a mediator EDA are typically small and self-contained. Thus, this architecture shares many of the operational advantages of the broker version.

Guided change with fitness functions

Developers find it easier to build fitness functions for the mediator than for the broker EDA. The tests for individual event processors don't differ much from the broker version. However, holistic fitness functions are easier to build because developers can rely on the mediator to handle coordination. For example, in the insurance workflow, a developer can write a test and easily tell if the entire process was successful because the mediator coordinates it.

Appropriate coupling

While many testing scenarios become easier with mediators, coupling increases, harming evolution. The mediator includes important domain logic, increasing the size of the architectural quantum to encompass it, which in turn couples each service to one another. In this architecture, when a developer makes a change, other developers must consider the side effects for the other services in the workflow, increasing coupling.

From an evolutionary standpoint, the broker architecture has clear advantages because of reduced coupling. In the mediator pattern, the coordinator acts as a coupling point, binding all the affected services together. In a broker topology, behavior can evolve by adding new processors to existing message queues without affecting the others (except in cases of overburdening the queue with traffic, which is solvable by a variety of architectural patterns and/or fitness functions). Because broker topologies are inherently decoupled, evolution is easier.

This is a classic example of an architectural tradeoff. Broker EDAs offer many advantages in terms of evolvability, asynchronicity, scale, and a host of other desirable characteristics. However, common tasks like transactional coordination become more difficult.

Service-Oriented Architectures

There are a variety of service-oriented architectures (SOAs) in existence, including many hybrids. Here are some common architectural patterns.

ESB-DRIVEN SOA

A particular manner of creating SOAs became popular several years ago, building an architecture based around services and coordination via a *service bus*—typically called an *Enterprise Service Bus* (ESB). The service bus acts as a mediator for complex event interactions and handles various other typical integration architecture chores such as message transformation, choreography, and so on.

While ESB architectures typically use the same building blocks as EDAs, the organization of services differs, and is based on a strictly defined service taxonomy. The ESB style differs from organization to organization, but all are based on segregating services based on reusability, shared concepts, and scope. A representative ESB SOA is shown in Figure 4-10.

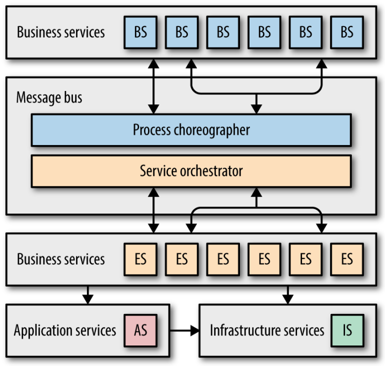

Figure 4-10. Typical service taxonomy for an ESB SOA

In Figure 4-10, each layer of the architecture has specific responsibilities. The business services define the coarse-grained functionality of the business as abstract definitions, sometimes defined by business users using standards like BPEL (Business Processing Execution Language). The goal of business services is to capture what the company does in an abstract way. To determine whether you have defined these services at the correct level of abstraction, ask yourself, “Are we in the business of…” affirmatively for each of the service names (like `CreateQuote` or `ExecuteTrade`). While a developer might invoke a `CreateCustomer` service in order to create a quote, that isn’t the main thrust of the business but rather a necessary intermediate step.

The abstract business services must call code to implement their behavior, which are *enterprise services*: concrete implementations meant for sharing, owned by a distinct service teams. The goal of this team is to create reusable services integration architects can “stitch together” using choreography to form business implementations. Developers aim for high reuse and design enterprise services accordingly (to understand why this often fails, see “Antipattern: Code Reuse Abuse”).

Some services don’t need a high degree of reuse. For example, one part of the system may need geolocation, but it isn’t important enough to devote resources to make it a full blown enterprise service. The *Application Services* at the bottom left in Figure 4-10 handle these cases. These are bound to a specific application context and not meant for reuse, and are typically owned by a specific application team.

Infrastructure services are shared services owned by an infrastructure team to handle nonfunctional requirements such as monitoring, logging, authentication/authorization, and so on.

The defining characteristic of an ESB-driven SOA is the message bus architectural component, responsible for a wide variety of tasks as follows:

Mediation and routing

> The message bus knows how to locate and communicate with services. Typically, it maintains a registry of physical location, protocols, and other information needed to invoke services.

Process choreography and orchestration

> The message bus composes enterprise services together and manages tasks like invocation order.

Message enhancement and transformation

> One of the benefits of an integration hub is its ability to handle protocol and other transformations on behalf of applications. For example, `ServiceA` may “speak” `HTTP` and needs to call `ServiceB`, which only “speaks” `RMI/IIOP`. Developers can configure the message bus to handle this transformation invisibly anytime this conversion is needed.

The architectural quantum for ESB-driven SOA is massive! It basically encompasses the entire system, much like a monolith, but is much more complex because it is a distributed architecture. Making singular evolutionary change is extraordinarily difficult in ESB-driven SOA because the taxonomy, while assisting reuse, harms common change. For example, consider the `CatalogCheckout` domain concept within an SOA—it is smeared throughout the technical architecture. Making a change to only `CatalogCheckout` requires coordination between the parts of the architecture, commonly owned by different teams, generating a tremendous amount of coordination friction.

Contrast this representation of `CatalogCheckout` with the bounded context partitioning of microservices. In a microservices architecture, each bounded context represents a business process or workflow. Thus, developers would build a bounded context around something like `CatalogCheckout`. It is likely that `CatalogCheckout` will need details about `Customer`, but each bounded context "owns" their own entities. If other bounded contexts also have the notion of `Customer`, developers make no attempt to unify around a single, shared `Customer` class, which would be the preferred approach in an ESB-driven SOA. If the `CatalogCheckout` and `ShipOrder` bounded contexts need to share information about their customers, they do so via messaging rather than trying to unify around a single representation.

ESB-driven SOA was never designed to exhibit evolutionary properties, so it's no surprise that none of the evolutionary facets score well here:

Incremental change

While having a well-established technical service taxonomy allows for reuse and segregation of resources, it greatly hampers making the most common types of change to business domains. Most SOA teams are as partitioned as the architecture, requiring herculean amounts of coordination for common changes. ESB-driven SOA is notoriously difficult to operationalize as well. It typically consists of multiple physical deployment units, making coordination and automation challenging. No one chooses ESBs for agility and operational ease of use.

Guided change with fitness functions

Testing in general is difficult within ESB-driven SOA. No one piece is complete—every piece is part of a larger workflow and isn't typically designed for isolated testing. For example, an enterprise service is designed for reuse, but testing its core behavior is challenging because it is only a portion of potentially a variety of workflows. Building atomic fitness functions is virtually impossible, leaving most verification chores to large-scale holistic fitness functions that do end-to-end testing.

Appropriate coupling

From a potential enterprise reuse standpoint, extravagant taxonomy makes sense. If developers can manage to capture the reusable essence of each workflow, they will eventually write all the company's behavior once and for all, and future application development consists of connecting existing services. However, in the real world this isn't always possible. ESB-driven SOA isn't built to allow independent evolvable parts, so it has extremely poor support for it. Designing for categorical reuse harms the ability to make evolutionary change at the architectural level.

Software architectures aren't created in a vacuum—they always reflect the ecosystem in which they were defined. For example, when SOA was a popular architectural style, companies didn't use tools like open-source operating systems—all infrastructure was commercial, licensed, and expensive. A decade ago, a developer proposing a microservices architecture, where every service runs on its own instance of an operating system and machine, would be laughed out of the operations center because the architecture would have been ludicrously expensive. Because of the dynamic equilibrium of the software development ecosystem, new architectures arise because of a literal new environment.

While architects may still choose ESB-driven SOA for integration heavy environments, scale, taxonomy, or other legitimate reasons, they choose it for those features rather than evolvability, for which it is spectacularly unsuited.

MICROSERVICES

Combining the engineering practices of Continuous Delivery with the logical partitioning of bounded context forms the philosophical basis for the microservice style of architecture, along with our architectural quantum concept.

In a layered architecture, the focus is on the *technical* dimension, or how the mechanics of the application work: persistence, UI, business rules, etc. Most software architectures focus primarily on these technical dimensions. However, an additional perspective exists. Suppose that one of the key bounded contexts in an application is *Checkout*. Where does it live in the layered architecture? Domain concepts like *Checkout* smear across the layers in this architecture. Because the architecture is segregated via technical layers, there is no clear concept of the *domain* dimension in this architecture, as can be seen in Figure 4-11.

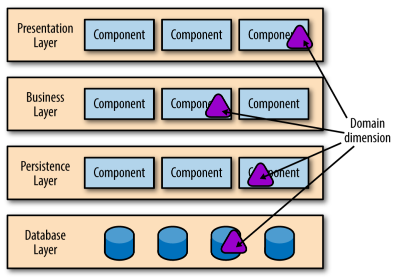

Figure 4-11. The domain dimension is embedded within technical architecture

In Figure 4-11, some portion of *Checkout* exists in the UI, another portion lives in the business rules, and persistence is handled by the bottom layers. Because layered architecture isn't designed to accomodate domain concepts, developers must modify each layer to make changes to domains. From a domain perspective, a layered architecture has zero evolvability. In highly coupled architectures, change is difficult because coupling between the parts developers want to change is high. Yet, in most projects, the common unit of change revolves around domain concepts. If a software development team is organized into silos resembling their role in the layered architecture, then changes to *Checkout* require coordination across many teams.

In contrast, consider an architecture where the *domain* dimension is the primary segregation of the architecture, as shown in Figure 4-12.

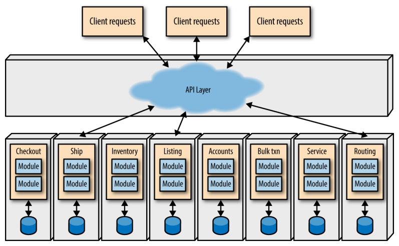

Figure 4-12. Microservices architectures partition across domain lines, embedding the technical architecture

As shown in Figure 4-12, each service is defined around DDD domain concept, encapsulating the technical architecture and all other dependent components (like databases) into a bounded context creating a highly decoupled architecture. Each service "owns" all parts of its bounded context, and communicates with other bounded contexts via messaging (such as REST or message queues). Thus, no service is allowed to know the implementation details of another service (such as database schemas), preventing inappropriate coupling. The operational goal of this architecture is to replace one service with another without disrupting other services.

Microservices architectures generally follow seven principles, as discussed in Building Microservices Architectures:

Modeled around the business domain

The emphasis in microservices design is on the business domain, not technical architecture. Thus, the quantum reflects the bounded context. Some developers make the mistaken association that a bounded context represents a single entity such as `Customer`; instead, it represents a business context and/or workflow such as CatalogCheckout. The goal in microservices isn't to see how small developers can make each service but rather to create a useful bounded context.

Hide implementation details

The technical architecture in microservices is encapsulated within the service boundary, which is based on the business domain. Each domain forms a physical bounded context. Services integrate with each other by passing messages or resources, not by exposing details like database schemas.

Culture of automation

Microservices architectures embrace Continuous Delivery, by using deployment pipelines to rigorously test code and automate tasks like machine provisioning and deployment. Automated testing in particular is extremely useful in fast-changing environments.

Highly decentralized

Microservices form a *shared nothing* architecture—the goal is to decrease coupling as much as possible. Generally, duplication is preferable to coupling. For example, both the `CatalogCheckout` and `ShipToCustomer` services have a concept called `Item`. Because both teams have the same name and similar properties, developers try to reuse it across both services, thinking it will save time and effort. Instead, it increases effort because changes must now propagate between all the teams that share the component. And, whenever a service changes, developers must worry about changes to the shared component. If, on the other hand, each service has their own `Item` and passes information they need from `CatalogCheckout` to `ShipToCustomer` without coupling to the component, they can each change independently.

Deployed independently

Developers and operations expect that each service component will be deployed independently from other services (and other infrastructure), reflecting the physical manifestation of the bounded context. The ability for developers to deploy one service without affecting any other service is one of the defining benefits of this architectural style. Moreover, developers typically automate all deployment and operations tasks, including parallel testing and Continuous Delivery.

Isolate failure

Developers isolate failure both within the context of a microservices and in the coordination of services. Each service is expected to handle reasonable error scenarios and recover if possible. Many DevOps best practices (such as the circuit breaker pattern, bulkheads, and so on) commonly appear in these architectures. Many microservices architectures adhere to the Reactive Manifesto (http://www.reactivemanifesto.org/), a list of operational and coordination principles that lead to more robust systems.

Highly observable

Developers cannot hope to manually monitor hundreds or thousands of services (how many multicast SSH terminal sessions can one developer observe?). Thus, monitoring and logging become first-class concerns in this architecture. If operations cannot monitor one of these services, it might as well not exist.

The main goals of microservices are isolation of domains via physical bounded context and emphasis on understanding the problem domain. Therefore, the architectural quantum is the service, making this an excellent example of an evolutionary architecture. If one service needs to evolve to change its database, no other service is affected because no other service is allowed to know implementation details like schemas. Of course, the developers of the changing service will have to deliver the same information via the integration point between the services (hopefully protected by a fitness function like *consumer-driven contracts*), allowing the calling service developers the bliss of never knowing the change occurred.

Given that microservices is our exemplar for an evolutionary architecture, it is unsurprising that it scores well from an evolutionary standpoint.

Incremental change

> Both aspects of incremental change are easy in microservices architectures. Each service forms a bounded context around a domain concept, making it easy to make changes that only affect that context. Microservices architectures rely heavily on automation practices from *Continuous Delivery*, utilizing deployment pipelines and modern DevOps practices.

Guided change with fitness functions

> Developers can easily build both atomic and holistic fitness functions for microservices architectures. Each service has a well-defined boundary, allowing a variety of levels of testing within the service components. Services must coordinate via integration, which also requires testing. Fortunately, sophisticated testing techniques grew alongside the development of microservices.

Appropriate coupling

> Microservices architectures typically have two kinds of coupling: *integration* and *service template*. Integration coupling is obvious—services need to call each other to pass information. The other type of coupling, service templates, prevents harmful duplication. Developers and operations benefit if a variety of facilities are consistent and managed within microservices. For example, each service needs to include monitoring, logging, authentication/authorization, and other "plumbing" capabilities. If left to the responsibility of each service team, ensuring compliance and lifecycle management like upgrades will likely suffer. By defining the appropriate technical architecture coupling points in service templates, an infrastructure team can manage that coupling while freeing individual service teams from worrying about it. Domain teams merely extend the template and write their behavior. When upgrades to infrastructure changes, the template picks it up automatically during the next deployment pipeline execution.

The physical bounded context in microservices correlates exactly to our concept of architectural quantum—it is a physically decoupled deployable component with high functional cohesion.

One of the key principles of the microservices style of architecture is strict partitioning across domain-bounded contexts. The technical architecture is embedded within the domain parts, honoring DDD's bounded context principle by making each service physically separate, which leads to a *share nothing* architecture from the technical perspective. Physical separation is expected for each service, allowing easy replacement and evolution. Because each microservice embeds the technical architecture within the bounded context, any service may evolve in any way necessary. Thus, the dimensions of evolvability for microservices corresponds to the number of services, each of which developers can treat independently because each service is highly decoupled.

"SHARE NOTHING" AND APPROPRIATE COUPLING

Architects often call microservices a "share nothing" architecture. The primary advantage of this architecture style is no coupling at the technical architecture layer. But people who decry coupling are usually talking about "inappropriate coupling." After all, a software system with no coupling isn't very capable. "Share nothing" really means "no entangling coupling points." Even in microservices, some things need to be shared and coordinated, such as tools, frameworks, libraries, and so on. For instance, logging, monitoring, service discovery, etc. A service team forgetting to add monitoring capabilities to their service is a disaster at deployment time. In a microservices architecture, if a service can't be monitored, it disappears into a black hole.

Service templates (such as DropWizard (http://www.dropwizard.io/) and Spring Boot (http://projects.spring.io/spring-boot/)) are common solutions to this problem in microservices. These frameworks allow a DevOps team to build consistent tools, frameworks, versions, etc., into the service template. Service teams use the template to "snap in" their business behavior. When the monitoring tool updates, the service team can coordinate the update to the service template without bothering other teams.

If there are clear benefits, then why haven't developers embraced this style before? A decade ago, automatic provisioning of machines wasn't possible. Operating systems were commercial and licensed, with little support for automation. Real-world constraints like budgets impact architectures, which is one of the reasons developers build more and more elaborate shared resources architectures, segregated at the technical layers. If operations is expensive and cumbersome, architects build around it, as they did in ESB-SOAs.

The Continuous Delivery and DevOps movements added a new factor into the dynamic equilibrium. Now, machine definitions live in version control and support extreme automation. Deployment pipelines spin up multiple test environments in parallel to support safe continuous deployment. Because much of the software stack is open source, licensing and other concerns no longer impact architectures. The community reacted to the new capabilities emergent in the software development ecosystem to build more domain-centric architectural styles.

In microservices architecture, the domain encapsulates technical and other architectures, making evolution across domain dimensions easy. No one perspective on architecture is "correct," but rather a reflection on the goals developers build into their projects. If the focus is entirely on technical architecture, then making changes across that dimension is easier. However, if the domain perspective is ignored, then evolving across that dimension is no better than the Big Ball of Mud.

One of the major factors that impacts the ability to evolve an application at the architectural level is how *unintentionally coupled* each part of the system is. For example, in a layered architecture, architects specifically couple layers together in an intentional way. However, the domain dimension is unintentionally coupled, making evolution in that dimension difficult, because the architecture is designed around technical architecture layers, not the domain. Thus, one of the important aspects of an evolvable architecture is *appropriate coupling* across dimensions. We discuss how to identify and utilize quantum boundaries for practical purposes in Chapter 8.

SERVICE-BASED ARCHITECTURES

A more commonly used architectural style for migration is a *service-based architecture*, which is similar to but differs from microservices in three important ways: service granularity, database scope, and integration middleware. Service-based architectures are still domain-centric but address some challenges developers face when restructuring existing applications toward more evolutionary architectures.

Larger service granularity

> The services in this architecture tend to be larger, more "portion of a monolith" granularity than purely around domain concepts. While they are still domain-centric, the larger size makes the unit of change (development, deployment, coupling, and a host of other factors) larger, diminishing the ability to make change easily. When architects evaluate a monolithic application, they often see coarse-grained divisions around common domain concepts such as CatalogCheckout or Shippping, which form a good first-pass at partitioning the architecture. The goals of operational isolation are the same in service-based architectures as in microservices but are more difficult to achieve. Because the service size is bigger, developers must consider more coupling points and the complications inherent in larger chunks of code. Ideally, the architecture should support the same kind of deployment pipeline and small unit of change as microservices: when a developer changes a service, it should trigger the deployment pipeline to rebuild the dependent services, including the application.

Database scope

> Service-based architectures tend towards a monolithic database, regardless of how well-factored the services are. In many applications, it isn't feasible or possible to restructure years (or decades) of intractable database schemas into atomic-sized chunks for microservices. While the inability to atomize the data may be inconvenient in some situations, it is impossible in some problem domains. Heavily transactional systems are a poor match for microservices because coordination between services, transactional behavior is too costly. Systems with complex transactional requirements map more cleanly to service-based architectures because of less stringent database requirements.

While the database remains unpartitioned, the components that rely on the database will likely change, becoming more granular. Thus, while the mapping between the services and the underlying data may change, it requires less restructuring. We cover evolutionary database design in Chapter 5.

Integration middleware

> The third difference between microservices and service-based architectures concerns externalized coordination via a mediator like a service bus. Building greenfield microservices applications allows developers to not worry about old integration points, but those horrors describe many environments rife with legacy systems that still perform useful work. Integration hubs, like enterprise service buses, excel at forming glue between disparate services with different protocols and message formats. If architects find themselves in environments where integration architecture is the top priority, using an integration hub makes adding and changing dependent services easier.

Using an integration hub is a classic architectural tradeoff: by using a hub, developers need to write less code to glue applications together, and may use it to mimic transaction coordination between services. However, using a hub increases the architectural coupling between components—developers can no longer make changes independently without coordinating with other teams. Fitness functions can mitigate some of this coordination cost, but the more developers increase coupling, the harder the system is to evolve.

Here is how a service-based architecture measures against our evolutionary architecture evaluation:

Incremental change

> Incremental change is relatively functional in this architecture because each service is domain centric. Most changes in software projects occur around domains, providing alignment between unit of change and quantum of deployment. While not as agile as microservices because the service size tends to be larger, many of the advantages of microservices is preserved.

Guided change with fitness functions

> Developers typically find it more difficult to write fitness functions in service-based architectures than in microservices because of increased coupling (typically at the database) and a larger bounded context. Increased code coupling often makes writing tests more difficult, and increased data coupling creates its own host of problems. The larger bounded context of service-based architectures creates more opportunities for developers to create internal coupling points, complicating testing and other diagnostics.

Appropriate coupling

> Coupling is often the reason developers pursue a service-based architecture rather than microservices: difficulties deconstructing database schemas, high degree of coupling within a monolith targeted for restructuring, and so on. Creating domain-centric services helps ensure appropriate coupling, and ser-

vice templates help create the appropriate level of technical architecture coupling.

Server-based BaaS architectures allow limited evolution but attractive operational characteristics. Service-based architectures are certainly more inherently evolvable than ESB SOA architectures. The degree to which developers have deviated from bounded context largely determines the quantum size and how much damaging coupling appears.

Service-based architectures are a good compromise between the philosophical purity of microservices and the pragmatic realities of many projects. By loosening the strictures on service size, database independence, and incidental but useful coupling, this architecture solves the most painful aspects of microservices while preserving many of the benefits.

“Serverless” Architectures

“Serverless” architectures are a recent shift in the software development equilibrium, with two broad meanings, both applicable to evolutionary architecture.

Applications that significantly or primarily depend on third-party applications and/or services in “the cloud” are called BaaS (Backend as a Service). For example, consider the simplified example shown in Figure 4-13.

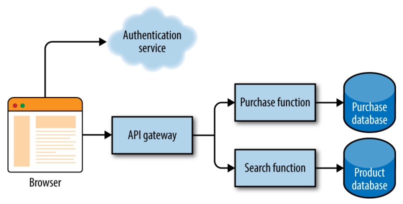

Figure 4-13. Serverless BaaS

In Figure 4-13, the developer writes little or no code. Instead, the architecture consists of wiring together services, including things like authentication, data transfer, and other integration architecture pieces. This type of architecture is appealing because, the fewer lines of code an organization writes, the fewer it must maintain. However, integration-heavy architectures come with their own challenges.

The other type of serverless architectures is FaaS (Function as a Service), which eschews infrastructure entirely (at least from the developer’s standpoint), provisioning infrastructure per request, automatically handling scaling, provisioning, and a host of other management duties. Functions in FaaS are triggered by event types defined by the service provider. For example, Amazon Web Services (AWS) is a common FaaS provider, supplying events triggered by file updates (on S3), time (scheduled tasks), and messages added to a message bus (e.g., Kinesis). Providers often limit the amount of time a request may take along with other restrictions, primarily around state. The general assumption is that FaaS functions are stateless, placing the burden of managing state on the caller.

Incremental change

Incremental change in serverless architectures should consist of redeploying code—all the infrastructure concerns exist behind the abstraction of “serverless.” These types of architecture are natural fits for deployment pipelines, to handle testing and incremental deployment as developers make changes.

Guided change via fitness functions

Fitness functions are critical in this type of architecture to ensure integration points stay consistent. Because coordination between services is key, developers can expect to write a larger percentage of holistic fitness functions, which must run in the context of several integration points, to ensure third-party APIs haven’t drifted. Architects frequently build anticorruption layers between integration points to avoid the Vendor King antipattern, discussed in “Antipattern: Vendor King”.

Appropriate coupling

From an evolutionary architecture standpoint, FaaS is attractive because it eliminates several different dimensions from consideration: technical architecture, operational concerns, and security issues, among others. While this architecture may be easy to evolve, it suffers from serious constraints around practical considerations, offloading much of the complexity to the invoker. For example, while FaaS will handle elastic scalability, the caller must handle any transactional behavior and other complex coordination. In a traditional application, transactional coordination is typically handled by the back-end. However, if the BaaS doesn’t support that behavior, coordination must move to the user interface (the invoker of the service).

Architects shouldn’t choose an architecture without evaluating it against the real problems they must solve.

TIP

Make sure your architecture matches the problem domain. Don’t try to force fit an unsuitable architecture.

While serverless architectures have many appealing features, they also have limitations. In particular, all-encompassing solutions often suffer from the “Antipattern: Last 10% Trap” as discussed on . Most of what a team needs to build is quick and easy, but other times, building a complete solution can be frustrating.

Controlling Quantum Size

The quantum size of an architecture largely determines how easy it will be for developers to make evolutionary changes. Large quanta like monoliths and ESB SOA are difficult to evolve because of the coordination required for each change. More decoupled architectures like broker event-driven and microservices offer many more avenues for easy evolution.

The structural constraints on evolving architecture depend on how well developers have handled coupling and functional cohesion. Evolution is easier if developers have created a modular component system with well-defined integration points. For example, if developers build a monolith, but are diligent about good modularity and component isolation, that architecture will offer more opportunities to evolve because the size of the architectural quantum is smaller due to decoupling.

TIP

The smaller your architectural quanta, the more evolvable your architecture will be.

Case Study: Guarding Against Component Cycles

PenultimateWidgets has several monolithic applications under active development. When designing components, one of the architect's goals is to create self-contained components—the more isolated the code, the easier it is to make changes. A common problem in many languages with powerful IDEs is the *package dependency cycle*, which describes the common scenario illustrated in Figure 4-14.

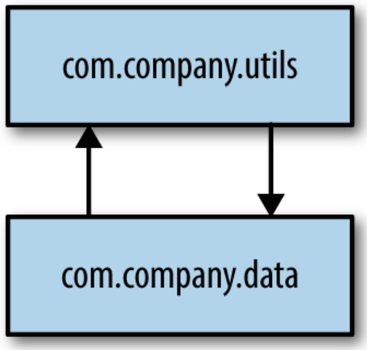

Figure 4-14. Package dependency cycle

In Figure 4-14, the package `com.company.data` imports from `com.company.utils`, and `com.company.utils` imports from `com.company.data`—neither component can be used without dragging the other along, creating a component cycle. Obviously, cycles hurt changeability because an elaborate network of cycles makes incremental change difficult. Languages like Java or C# have development environments that assist developers (via code insight helpers built into the IDE) by suggesting missing imports. Because developers implicitly import so many things in the course of daily coding with the help of the IDE, preventing package dependency cycles is difficult because the tooling fights against this effort.

The PenultimateWidgets architects on these systems worry about developers accidentally introducing cycles between components. Fortunately, they have a mechanism to help guard against factors that harm the evolvability of applications—fitness functions. Rather than abandon the benefits of IDEs because they encourage bad habits, an engineering safety net via fitness functions can be built instead. Both commercial and open source tools exist for many popular platforms to help untangle cycles. Many take the form of a static code analysis tool that looks for cycles, while others provide "to-do" lists of refactorings to assist developers in fixing them.

After the cycles have been removed, how can you prevent a developers's idle habits from introducing new ones? Coding standards don't help for this type of problem because developers have a hard time remembering bureaucratic policies in the heat of coding. Instead, they prefer to establish tests and other verification mechanisms to guard against too-helpful tools.

The PenultimateWidgets developers use a popular open source tool for the Java platform called JDepend (http://clarkware.com/software/JDepend.html), which includes both textual and graphical interfaces to help analyze dependencies. Because JDepend is written in Java, developers can utilize its API to write structural tests of their own. Consider the test case in Example 4-1.

Example 4-1. Using JDepend to identify cycles programmatically

```java
import java.io.*;
import java.util.*;
import junit.framework.*;

public class CycleTest extends TestCase {
    private JDepend jdepend;

    protected void setUp() throws IOException {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/util/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    /**
     * Tests that a single package does not contain
     * any package dependency cycles.
     */
    public void testOnePackage() {
        jdepend.analyze();
        JavaPackage p = jdepend.getPackage("com.xyz.thirdpartyjar
        assertEquals("Cycle exists: " + p.getName(),
               false, p.containsCycle());
        }

        /**
         * Tests that a package dependency cycle does not
         * exist for any of the analyzed packages.
         */
        public void testAllPackages() {
            Collection packages = jdepend.analyze();
            assertEquals("Cycles exist",
                         false, jdepend.containsCycles());
        }
    }
```

In Example 4-1, the developer adds the directories containing packages to `jdepend`. Then, the developer can test either a single package for cycles or the entire codebase, as shown in the unit test `testAllPackages()`. Once the project has gone through the laborious task of identifying and removing cycles, put the `testAllPackages()` unit test in place as an application architecture fitness function to guard against future cycle occurrence.

PREV
3. Engineering Incremental Change

NEXT
5. Evolutionary Data