Chapter 8. Component-Based Thinking

In Chapter 3, we introduced the concept of a *module* as a collection of related code. In this chapter, we dive much deeper into this concept, focusing on the architectural aspect of modularity in terms of *logical components*—the building blocks of a system.

Identifying and managing logical components is a part of *architectural thinking* (see Chapter 2), so much so that we call this activity *component-based thinking*. Component-based thinking is seeing the structure of a system as a set of logical components, all interacting to perform certain business functions. It's at this level (not the class level) that an architect "sees" the system.

In this chapter, we define logical components within software architecture, how to identify them, and how to arrive at an appropriate level of granularity through analyzing what's known as *cohesion* (we define what that means a little later on in this chapter). We also discuss coupling between components and how and why to create loosely coupled systems.

Defining Logical Components

Think about the floor plan of a typical Western house, as illustrated in Figure 8-1. Notice the floor plan is made up of various rooms (such as a kitchen, bedrooms, bathrooms, a living room, an office, and so on), each serving a different purpose. These rooms represent the building blocks—the *components*—of the house.

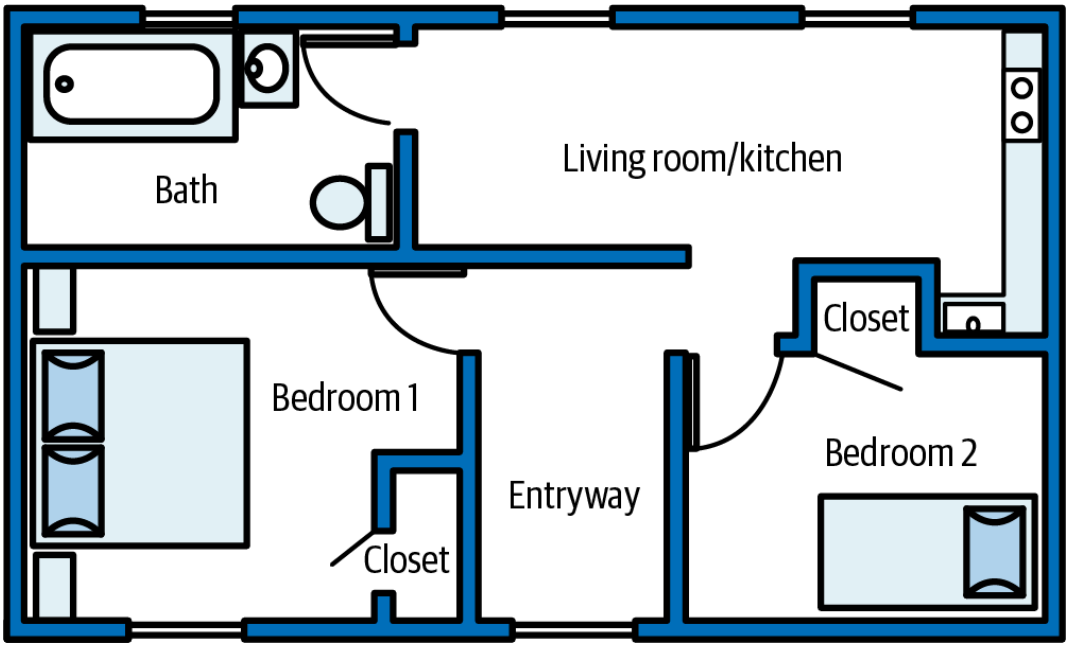

Figure 8-1. The various rooms represent the components of the house

In the same way, the major functions a system performs represent the components of that system, as shown in Figure 8-2. Like the rooms of a house, each component performs a specific function, such as managing inventory, shipping orders, or processing payments. Together they make up the system. Each component contains source code that implements that particular business function.

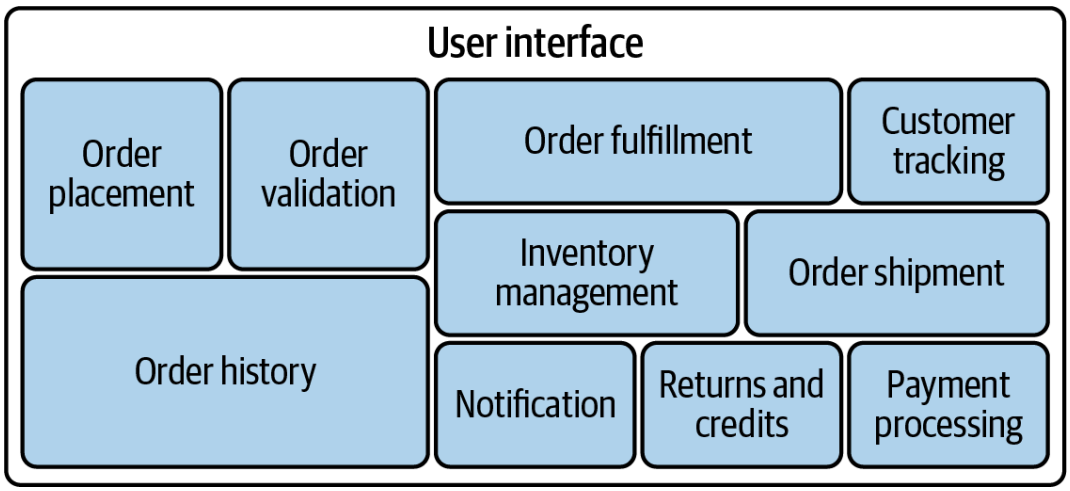

Figure 8-2. The various major functions represent the components of the system

Logical components in software architecture are usually manifested through a namespace or a directory structure containing the source code that implements that particular functionality. Typically, the *leaf nodes* of the directory structure or namespace containing source code represent the logical components in the architecture, and higher-level directories or namespace nodes represent the system's domains and subdomains. In the directory structure illustrated in Figure 8-3, the directory path *order_entry/ordering/payment* represents the `Payment Processing` component, and the path *order_entry/processing/fulfillment* represents the `Order Fulfillment` component. The source code underlying the directories implements these logical components.

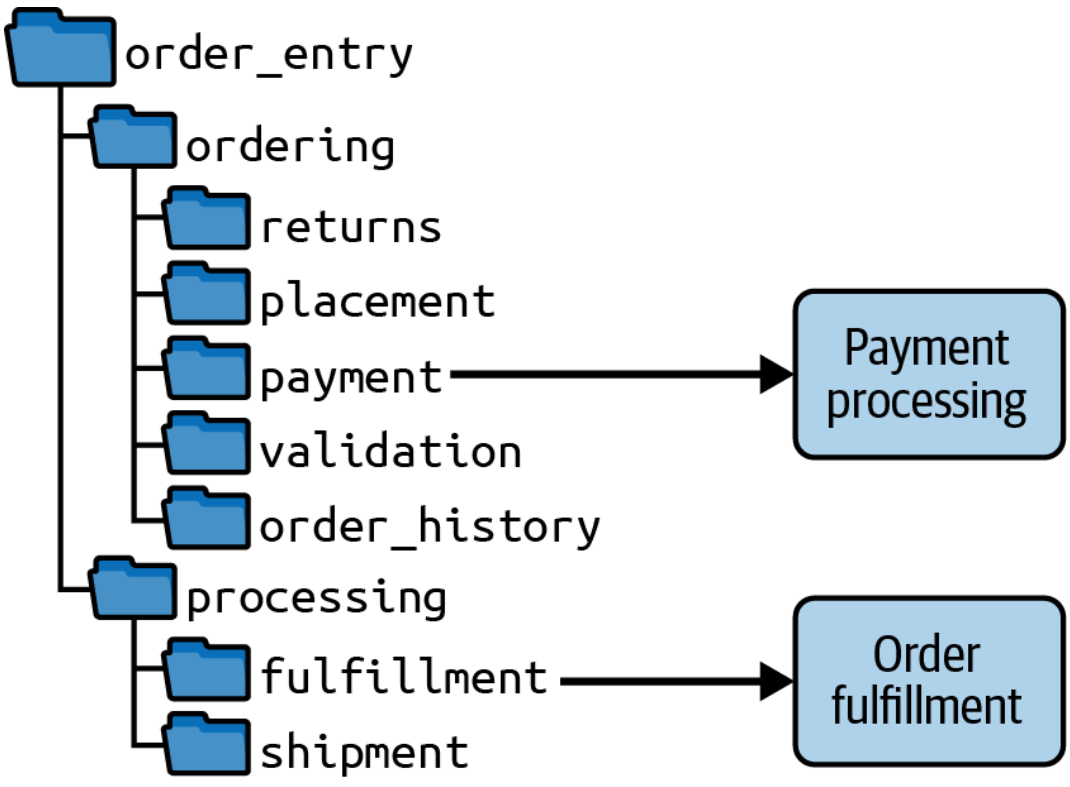

Figure 8-3. The leaf nodes of the directory represent the components of the system

An architect can analyze a software system's directory structures or namespaces to understand its internal structure—in other words, its *logical architecture*. We describe the differences between logical and physical architectures in the next section.

Logical Versus Physical Architecture

A *logical architecture* consists of a system's logical components (building blocks) and how they interact with one another. It also usually shows their interactions with various *actors* (users who interact with the system), and may also show a repository (not a database) to clarify where data is used or transferred between components. Figure 8-4 illustrates an example of a logical architecture.

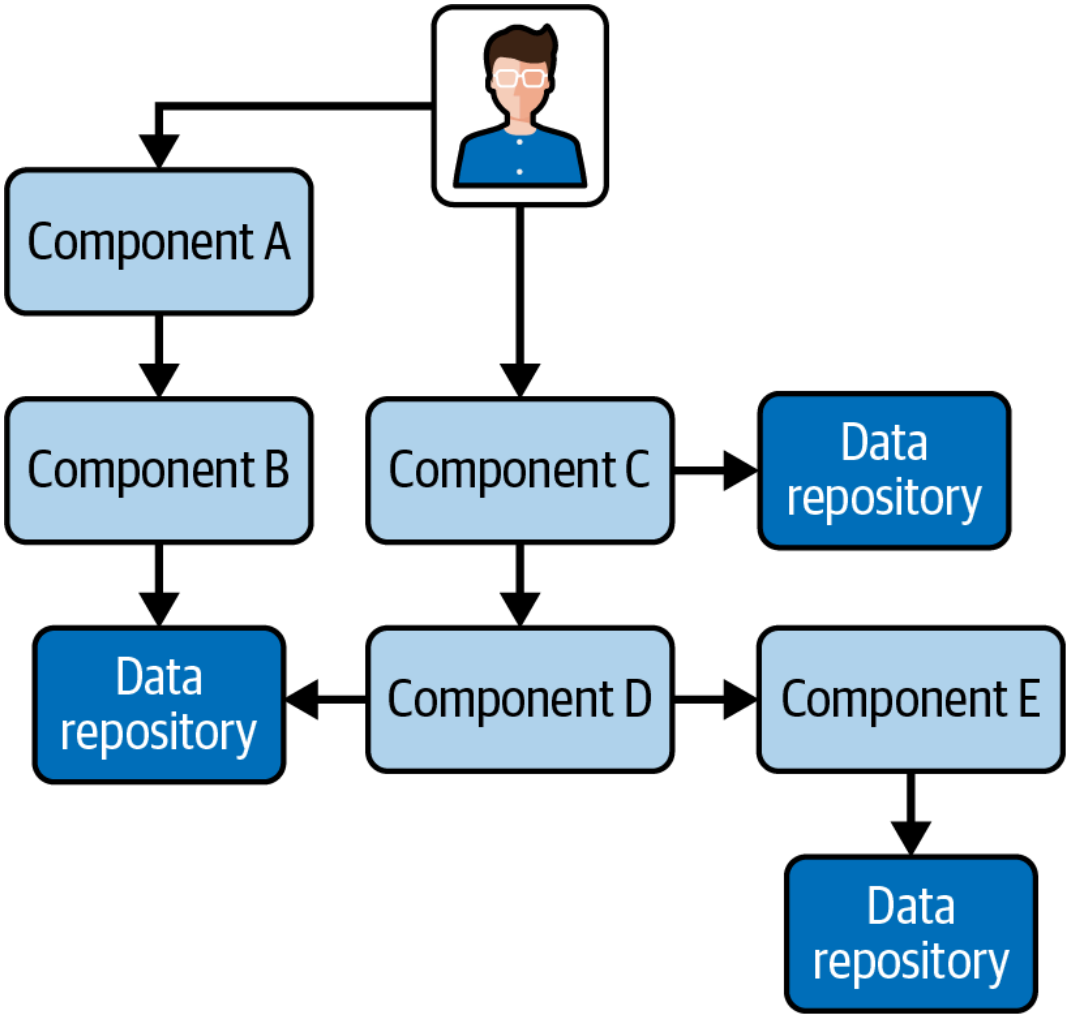

Figure 8-4. Diagram of a logical architecture

A logical architecture diagram doesn't typically show a user interface, databases, services, or other physical artifacts. Rather, it shows the logical components and how they interact, which should match the directory structures and namespaces that organize the code.

A *physical architecture*, on the other hand, includes such physical artifacts as services, user interfaces, databases, and so on. Figure 8-5 is an example of a physical architecture diagram.

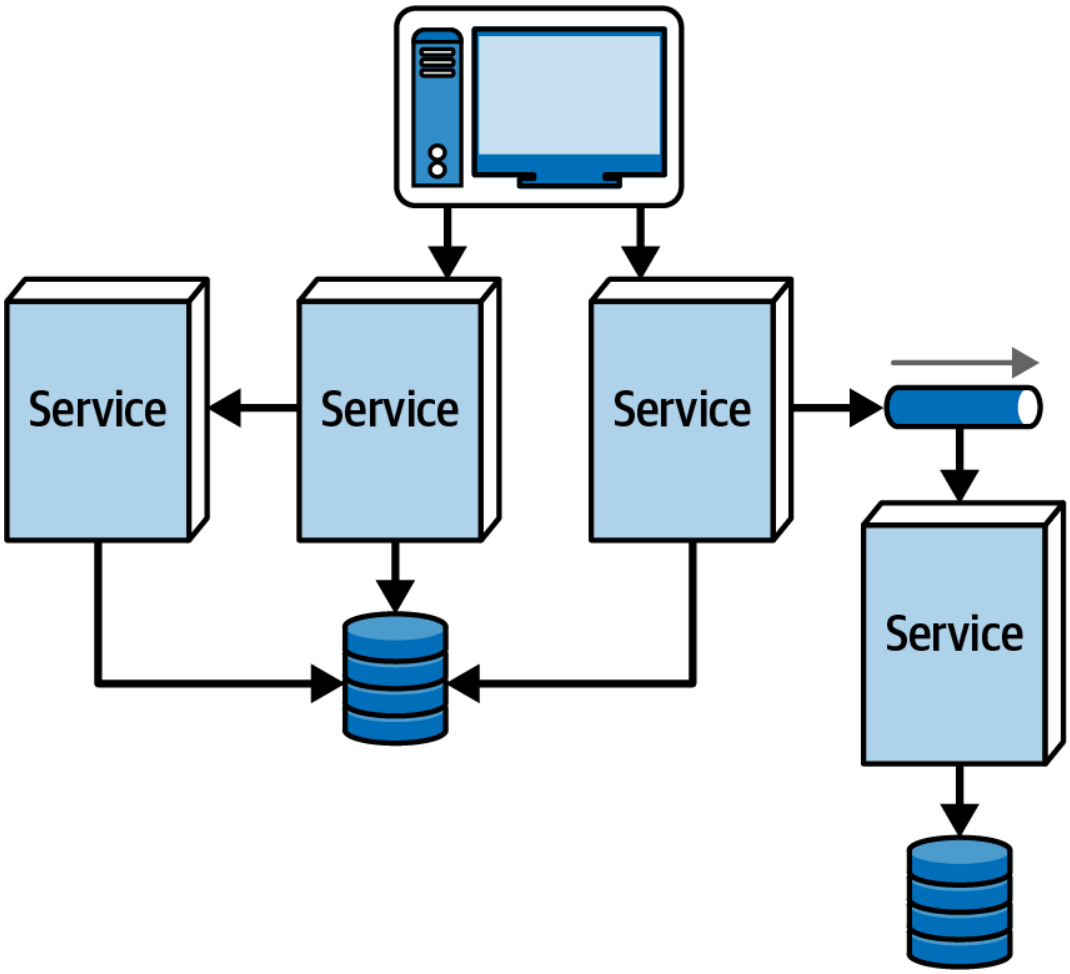

Figure 8-5. Diagram of a physical architecture

The system's physical architecture should closely represent one (or more) of the many architectural styles described in Part II of this book: microservices, layered architecture, event-driven architecture, and so on.

Many architects choose to bypass creating a logical architecture and start directly with a physical architecture. However, we advise against this practice, because a physical architecture doesn't always show where the system's functionality is and how it all fits together. For example, in a physical architecture, payment-processing functionality may be spread across multiple services, making it difficult to see how it interacts with other parts of the system. Furthermore, a physical architecture doesn't provide development teams with any guidance on how to build a monolithic or distributed system, or organize its code, resulting largely in unstructured architectures that are hard to maintain, test, and deploy.

Generally, a system's logical architecture is independent of its physical architecture. In other words, when creating a logical architecture, the focus is more on what the system does, how that functionality is demarcated, and how the functional parts of the system interact, rather than on its physical structure. For example, the architect creating a logical architecture like that shown in Figure 8-4 may not have determined yet whether those components will all go into a monolithic architecture (a single deployment unit) or will be deployed as services (separate deployment units), nor have they necessarily decided what kind of architecture style would be most appropriate.

Creating a Logical Architecture

Creating a logical architecture involves continuously identifying and restructuring logical components. *Component identification* works best as an iterative process. It involves producing candidate components, then refining them through a feedback loop, as illustrated in Figure 8-6.

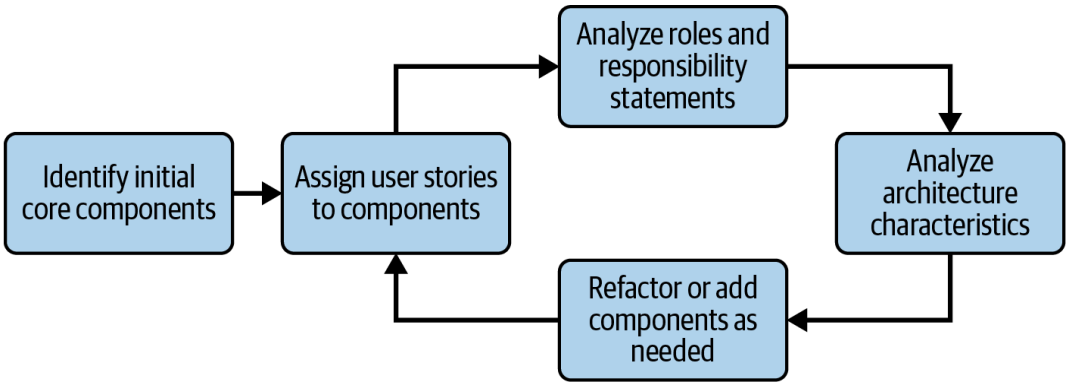

Figure 8-6. The component identification and refactoring cycle

The first activity an architect does in developing a logical architecture is to identify the initial core components, and then assign user stories or requirements to them. After this, they analyze the component's roles and responsibilities to make sure that the user stories or requirements assigned to it make sense. Then the architect looks at the architectural characteristics the system must support and determines whether any refactoring is needed to possibly break apart or combine the component based on those characteristics. Finally, the architect optionally refines the components based on this analysis. This process is a feedback loop that essentially never stops.

The workflow in Figure 8-6 can be used for greenfield (new) systems or anytime a feature is added to or changed in the system. For example, suppose you need to enable an order-entry system to allow users to pick items up at a store rather than having them shipped to their homes. This will involve adding scheduling code as well as changing the existing ordering process. This change may require new components, changes to existing ones, or both. As components change, their roles and responsibilities may change, prompting you to restructure your code or create new components for it.

We describe each of these steps in detail in the following sections.

Identifying Core Components

The challenge in starting out with a new logical architecture (or making major modifications to an existing one) is determining what the initial core components should be. One mistake many software architects make is spending too much effort trying to get the initial logical components perfect the first time. A better approach is to make a "best guess" as to what the initial core components might look like, based on the core functionalities of the system, and refine them through the workflow steps outlined in Figure 8-6. In other words, it's better to iterate over the logical components as you learn more about the system than to try to get it all perfect the first time, when you know least about the system and its specific requirements.

In most cases, the architect does not need to know all (or sometimes any) of the system's requirements and specifications to start creating logical components. Typically, the initial core components are based on major actions users might take or the system's major processing workflows.

We like to think of the initial core components as empty buckets. The bucket doesn't do anything until the architect starts "filling" it—that is, assigning user stories or requirements to that component. As illustrated in Figure 8-7, the component name should represent its proposed role and responsibility. However, until the architect assigns user stories to it (the next step in the workflow), it's essentially a placeholder—a best guess at a functional building block.

Figure 8-7. In the beginning, initial components are like empty buckets

The following are three common approaches to creating initial core components; the first two approaches (the Workflow and Actor/Action) are ones we find useful, and the third (the Entity Trap) is an antipattern we caution against.

The Workflow approach

A common approach architects use for identifying the initial core components of a logical architecture is the *Workflow* approach. As the name suggests, this approach leverages the major happy-path (nonerror) workflows a user might take through the system (or its main request-processing workflow). If the architect has some sense of the general flow, they can develop components based on those steps.

For example, assume you are building a new order-entry system for your company. You don't know the specific requirements and specifications yet, but you at least know the general workflow for processing a new order. You can thus assign a component to each step in the workflow:

1. User browses the catalog of items → `Item Browser`
2. User places an order → `Order Placement`
3. User pays for the order → `Order Payment`
4. Send user an email with order details → `Customer Notification`
5. Prepare the order → `Order Fulfillment`
6. Ship the order → `Order Shipment`
7. Email customer that order has shipped → `Customer Notification`

8. Track shipment → `Order Tracking`

In the Workflow approach, not every step in the major workflow yields a new component. In the workflow above, steps 4 and 7 both use the `Customer Notification` component. The architect models as many major workflows or user journeys they can for the system, identifying corresponding components from those steps.

This is, again, only a "best guess" as to what the logical architecture might look like. These components represent empty buckets and have no responsibilities yet, so they will likely change as they evolve (as illustrated in Figure 8-6). This is perfectly normal and part of the iterative nature of software architecture. Don't worry about trying to model every workflow in your system. Instead, focus on the major workflows. The rest will evolve as you learn more about the system and start gathering user stories and requirements.

The Actor/Action approach

Another way architects identify initial core components is the *Actor/Action* approach. This approach is particularly useful when a system has multiple actors. With this approach, the architect identifies the major actions a user can perform in the system (such as placing an order). The system itself is always an actor, too, performing automated functions such as billing and replenishing stock.

In our order-entry system example, an architect might identify three actors: the *customer*, the *order packer* (the person who packs the box and sends it off for shipping), and the *system*. The architect then identifies the major actions each of these actors takes and assigns components to those actions:

Customer actor

- Search for items → `Item Search`
- View the details about an item → `Item Details`
- Place an order → `Order Placement`
- Cancel an order → `Order Cancel`
- Register as a new customer → `Customer Registration`
- Update customer information → `Customer Profile`

Order packer actor

- Select the box size → `Order Fulfillment`
- Mark the order as ready for shipment → `Order Fulfillment`
- Ship the order to the customer → `Order Shipment`

System actor

- Adjust inventory → `Inventory Management`
- Order more stock from the supplier → `Supplier Ordering`
- Apply payment → `Order Payment`

Like with the Workflow approach, not every action necessarily has its own component. For example, selecting the box size for the order and marking it ready for shipping are both done by the `Order Fulfillment` component.

Generally, the Actor/Action approach generates more components than the Workflow approach, depending on how many major workflows the architect chooses to model. However, in both approaches, the architect can identify the initial core components and how they communicate even before receiving detailed requirements or specifications.

The Entity Trap

It's all too tempting for an architect to start identifying components by focusing on the entities involved with the system, then deriving components from those entities. For example, in a typical order entry system, you may identify `Customer`, `Item`, and `Order` as the primary entities in the system, and correspondingly create a `Customer Manager` component, an `Item Manager` component, and an `Order Manager` component. We strongly advise against this approach for the following reasons.

First, the names of the logical components are ambiguous and don't describe the role of the component. For example, asking what the `Order Manager` component does just by looking at the name yields the useless answer "it manages orders," indicating nothing about its specific role or responsibility in the system. Compare that to a component name like `Validate Order`, and the importance of a good, descriptive component name becomes clear. If a component name includes a suffix like `Manager`, `Supervisor`, `Controller`, `Handler`, `Engine`, or `Processor`, that's a good indicator that the architect might be caught in the Entity Trap antipattern.

Second, components become dumping grounds for domain-related functionalities. Consider an entity-based component name like `Order Manager`, as shown in Figure 8-8. Every bit of order functionality would go into that single component: order validation, order placement, order history, fulfilling the order, shipping the order, tracking the order, and so on. Essentially, it becomes like those "kitchen sink" utility classes every developer has written at least once in their career, with dozens of methods that perform string manipulation, data manipulation, time calculations, and whatever else the developer can put in there.

Third, when components become too coarse-grained, they do too much and lose their purpose. Rather than being fine-grained and single purpose (like the `Validate Order` component example), components can become too big. Such components are hard to maintain, test, and deploy, and hence not very reliable.

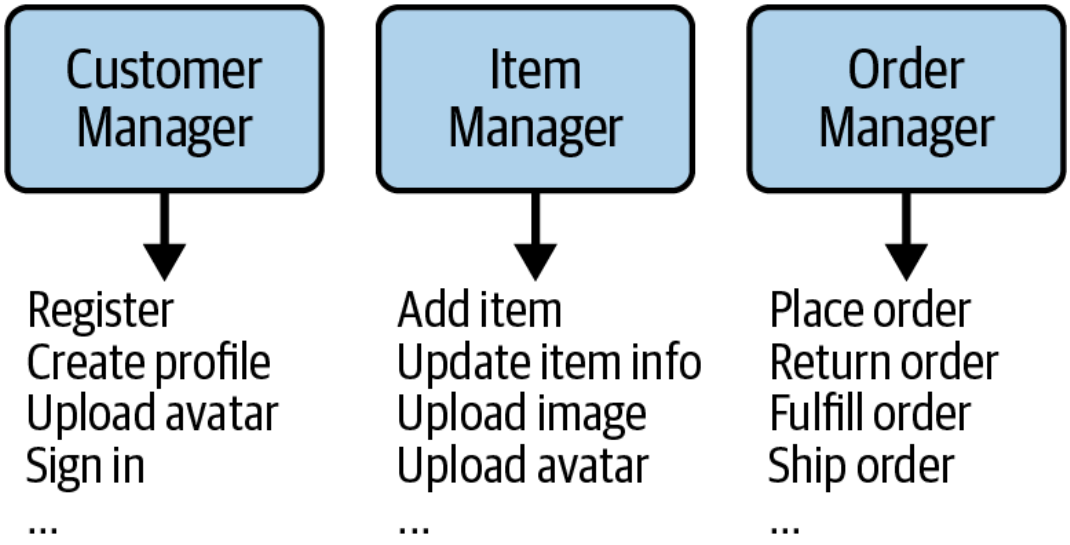

Figure 8-8. Using the Entity Trap antipattern produces components with too much responsibility

If an architect is building a system that truly is entity based and simply performs CRUD-based operations (create, read, update, and delete) against those entities, then the system doesn't need an architecture, but rather a CRUD-based framework, tool, or no-code/low-code environment that allows the developers to generate most of the source code that acts on those entities.

Assigning User Stories to Components

The next step in creating a logical architecture is to assign user stories or requirements to the logical components. This is an iterative process, since most user stories or requirements are not completely known up front; they evolve as the system evolves. This step is meant to start filling those empty buckets, giving the components specific roles and responsibilities, as illustrated in Figure 8-9.

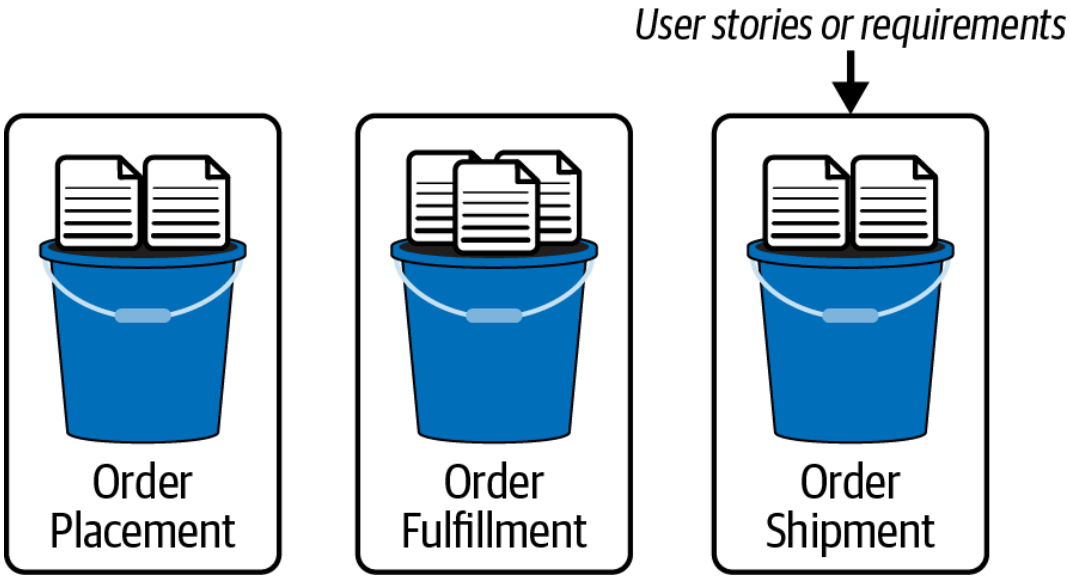

Figure 8-9. Filling up the empty buckets (components) with user stories or requirements

To see how logical components evolve, consider the following user stories:

Customer #1

> As a customer, I would like to have my order validated to make sure I have entered everything completely and correctly.

Order preparer

> As the person preparing the order, I would like to know what size box I should use for the order, so I can pack it in the most efficient way possible.

Customer #2

> As a customer, I would like to receive an email each time the order status changes, so I always know the state of my order.

Assume that the architect has identified the following logical components so far:

- `Order Placement`
- `Order Fulfillment`
- `Order Shipment`
- `Inventory Management`

It makes sense to assign the first user story to the `Order Placement` component, since that's the component the user is interacting with to place the order:

> Validate the order (Customer #1 user story) → `Order Placement`

Determining the box size should probably be handled by the `Order Fulfillment` component, since it's responsible for all the system logic needed to prepare and pack the order in a box:

> Determine box size (Order preparer user story) → `Order Fulfillment`

But what about the third user story? Which of the four components listed should send emails to the customer when the order has been placed, is ready for shipment, and has shipped? The answer might be the `Order Placement`, `Order Fulfillment`, and `Order Shipment` components. But keep in mind that the user story is implemented through source code, which needs to reside in a specific directory or namespace. Since it wouldn't be a good idea to replicate the code across three components, the architect needs to define a new component to handle this user story:

> Email customer (Customer #2 user story) → `Customer Notification` (new)

The `Order Placement`, `Order Fulfillment`, and `Order Shipment` components need to communicate with the new component to let it know to send an email. With this addition, the logical architecture now looks like Figure 8-10.

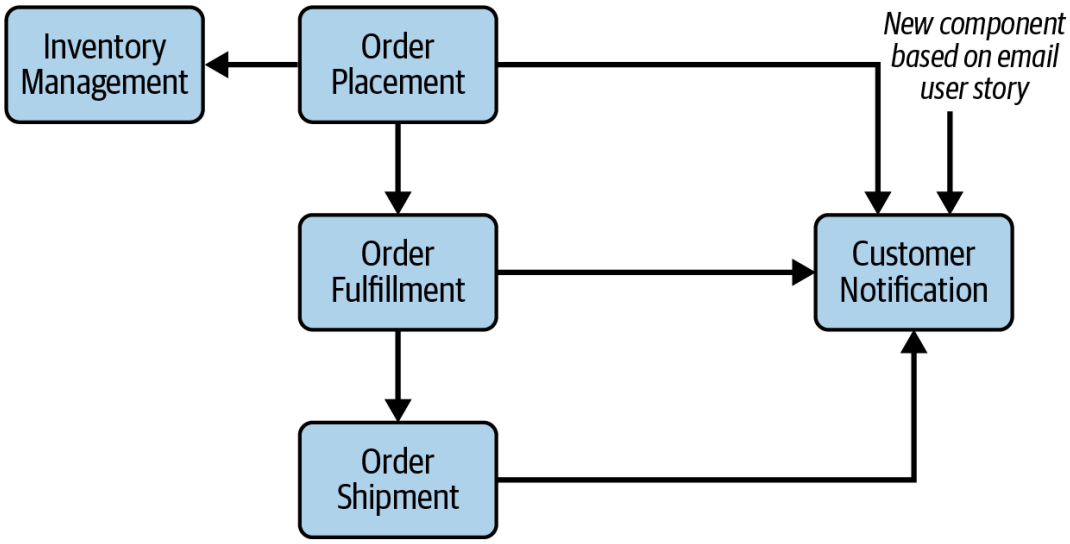

Figure 8-10. Evolving components based on new user stories

Analyzing Roles and Responsibilities

The next step in refining logical components is to analyze the roles and responsibilities of each component. This is how the architect ensures that the requirements or user stories assigned to those components belong there and that the components are not doing too much. What the architect is concerned about in this step is *cohesion*: how, and how much, a component's operations interrelate. Over time, components can get too big, even though their operations all interrelate.

To illustrate how this step works, assume the architect has assigned the following requirements to the `Order Placement` component:

- Validate the order to ensure all fields have been entered and are correct.
- Display the shopping cart with the item descriptions, quantities, and prices.
- Determine the correct shipping address.
- Collect payment information.
- Generate a unique order ID.
- Apply payment for the order.
- Adjust inventory counts for the items ordered.
- Email the customer an order summary.

If the architect were to write a role and responsibility statement for this component, it would read as follows:

> This component is responsible for validating the order and displaying the valid shopping cart, complete with the item picture, description, quantity, and price. This component is also responsible for determining the correct shipping address for the order, as well as collecting all the payment information from the customer. In addition, it's also responsible for applying the payment, adjusting inventory, and emailing the customer the order summary.

While all of these operations have to do with placing an order, the `Order Placement` component is clearly taking on too much responsibility, particularly with regard to applying the payment, adjusting the inventory, and emailing the customer. One way to see if a component is taking on too much responsibility is to look for conjunctive phrases such as *and*, *also*, *in addition*, or *as well as*, and excessive use of commas.

Recall from earlier in the chapter that a logical component is represented by a namespace or directory in the code repository. In the case of the `Order Placement` component, *all* the source code representing this component would be in the same directory or namespace, such as *com/app/order/placement* or *com.app.order.placement*. This is a lot of functionality—likely too much code for a single directory. Therefore, it would make sense to separate the class files for payment processing, inventory management, and email communications into separate directories representing those functionalities. That's exactly what separating logical components is all about.

If the architect moves the responsibilities of applying the payment, adjusting the inventory, and sending the customer an email to separate components, they can reduce the responsibility of the single `Order Placement` component, making it easier to maintain, test, and deploy. The resulting components would look like this:

`Order Placement`

- Validate the order to ensure all fields have been entered and are correct.
- Display the shopping cart with the item descriptions, quantities, and prices.
- Determine the correct shipping address.
- Collect payment information.
- Generate a unique order ID.

`Payment Processing`

- Apply payment.

`Inventory Management`

- Adjust the inventory counts for the items ordered.

`Customer Notification`

- Email the customer an order summary.

Each component now has a clearer, more distinct role and responsibility.

Analyzing Architectural Characteristics

The final analysis step is to consider the architectural characteristics the system will require. Some architectural characteristics, such as scalability, reliability, availability, fault tolerance, elasticity, and agility (the ability to respond quickly to change), may influence the size of a logical component.

For example, breaking up a larger component (one with lots of responsibility) into smaller components makes each of them easier to maintain and test (that's agility), and provides better scalability, elasticity, and fault tolerance. Another good example is where two parts of a system deal with user input: if one part deals with hundreds of concurrent users and the other needs to support only a few at a time, they will need different architecture characteristics. Thus, while a purely functional view of component design might lead an architect to assign a single component to handle user interaction, analyzing the components in terms of architecture characteristics might lead to a subdivision.

Because the architect must know the architectural characteristics before building a logical architecture, it is usually done *after* determining which architectural characteristics are most important to the system.

Restructuring Components

Feedback is critical in software design. Architects must continually iterate on their component designs in collaboration with developers. Designing software provides all kinds of unexpected difficulties. Thus, an iterative approach to component design is key. First, it's virtually impossible to account for all the different discoveries and edge cases that will arise, any of which could encourage redesign. Second, as the architecture and developers delve more deeply into building the application, they gain a more nuanced understanding of where its behaviors and roles should lie.

Architects should expect to restructure components frequently throughout the lifecycle of a system or product—not only in greenfield systems, but in any that undergo frequent maintenance.

Component Coupling

When components communicate with each other, or when a change to one component might impact other components, the components are said to be *coupled* together. The more coupled a system's components are, the harder it is to maintain and test the system (see Figure 8-11). Therefore, it's important to pay close attention to coupling.

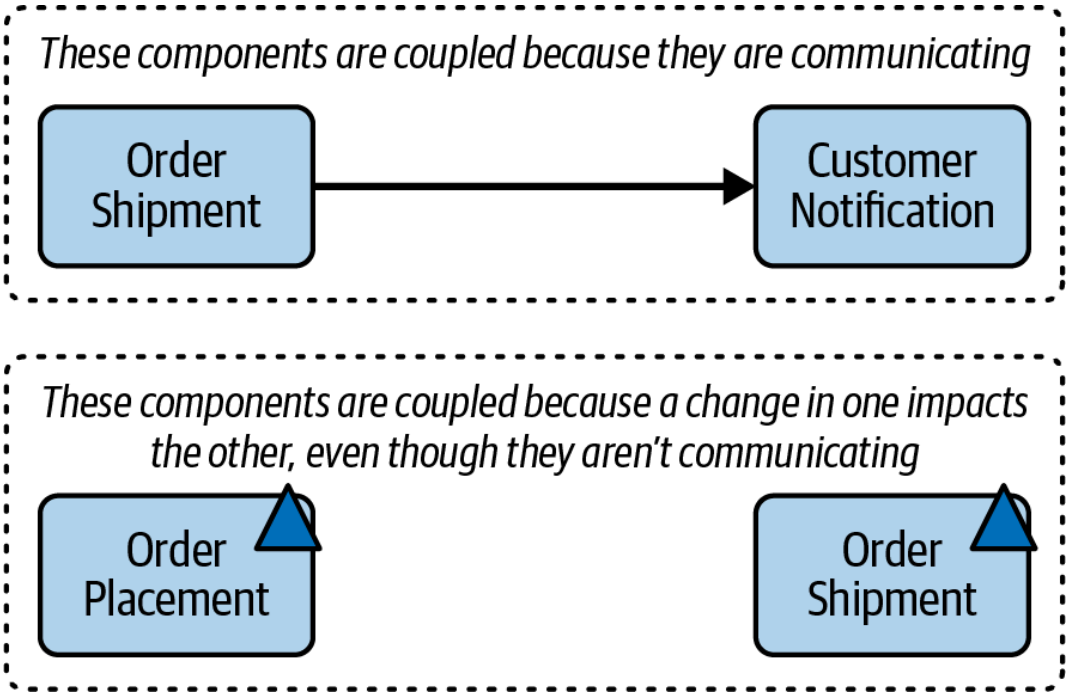

Figure 8-11. Components can be coupled even if they don't communicate directly

Static Coupling

Static coupling occurs when components communicate synchronously with each other. Architects need to be concerned about two types of coupling: afferent and efferent.

Afferent coupling (also known as *incoming* or *fan-in* coupling) is the degree to which other components depend on a target component. For instance, consider the `Customer Notification` component from our order-entry example in Figure 8-12. To email the customer, both the `Order Placement` and `Order Shipment` components need to communicate with the `Customer Notification` component. This means that the `Customer Notification` component is said to be *afferently coupled* to the `Order Placement` and `Order Shipment` components and would have an afferent coupling level of 2 (the number of incoming dependencies). Afferent coupling is usually denoted as *CA*.

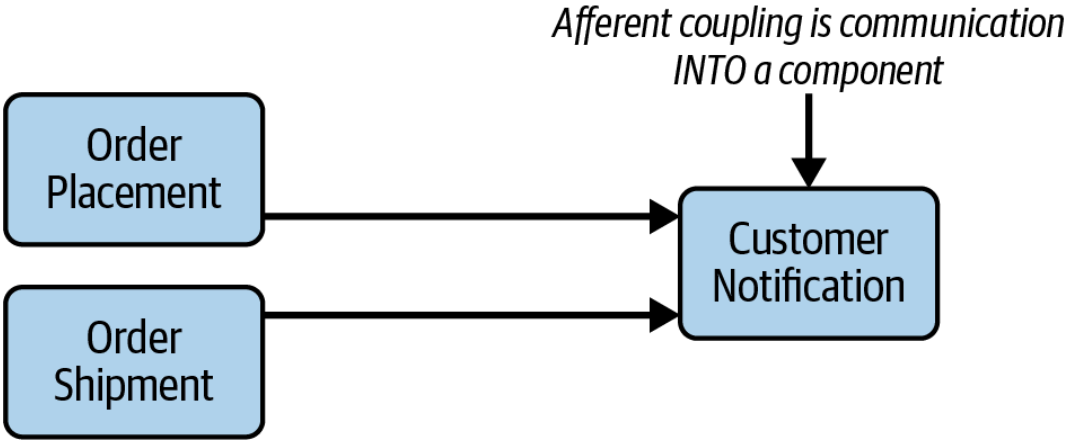

Figure 8-12. The `Customer Notification` component is afferently coupled to the other components

Efferent coupling (also known as *outgoing* or *fan-out* coupling) is the degree to which a target component depends on other components. For example, as illustrated in Figure 8-13, the `Order Placement` component is dependent on the

`Order Fulfillment` component and, as such, is efferently coupled to it and would have an efferent coupling level of 1 (the number of outgoing dependencies). Efferent coupling is usually denoted as *CE*.

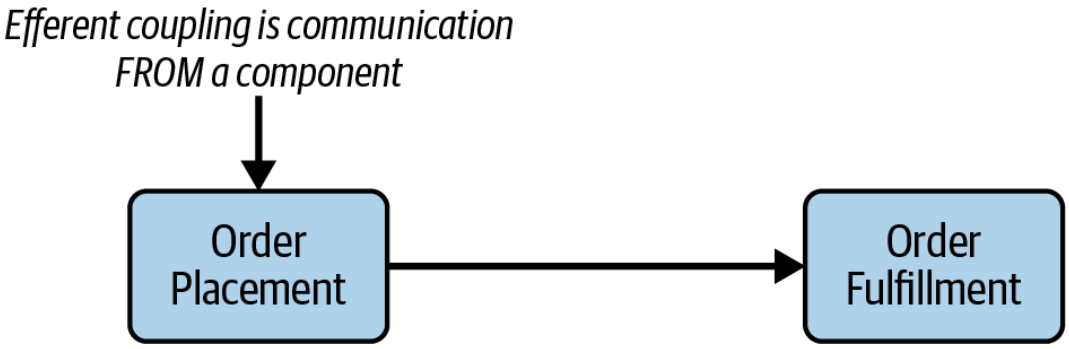

Figure 8-13. The `Order Placement` component is efferently coupled to the `Order Fulfillment` component

Temporal Coupling

Temporal coupling describes nonstatic dependencies, usually those based on timing or *transactions* (single units of work). For example, when the system is processing an order, the functionality in the `Order Placement` component must be invoked before the functionality in the `Order Shipment` component. Those components are thus said to be temporally coupled.

The problem with temporal coupling is that it's hard to detect using the tooling currently available on the market. In most cases, this kind of coupling is instead identified through design documents or detected through error conditions.

The Law of Demeter

Most architects are told to strive for loose coupling in system design. Less coupling between components or services makes a system more maintainable, easier to test, and less risky to deploy. It also increases the system's overall reliability, because changes impact fewer components, allowing fewer opportunities for error.

One technique for creating loosely coupled systems is called the *Law of Demeter*, otherwise known as the *Principle of Least Knowledge*. In Greek mythology, the goddess Demeter produced all the grain for the entire world, but she had no idea what people did with it (feeding livestock, making bread, and so on). Demeter was effectively decoupled from the rest of the world.

The Law of Demeter states: *a component or service should have limited knowledge of other components or services.* While this may sound simplistic and obvious, it's not. Let us show you what we mean.

Consider the components and their corresponding communications illustrated in Figure 8-14. Upon accepting a customer's order, the `Order Placement` component must tell the `Inventory Management` component to decrement the inventory. If the stock goes down too low, the `Order Placement` component must do two things: tell the `Supplier Ordering` component to order more stock from the supplier, and tell the `Item Pricing` component to adjust the price based on the limited supply available. Finally, once all this is done, the `Order Placement` component tells the `Email Notification` component to email the customer with the order details.

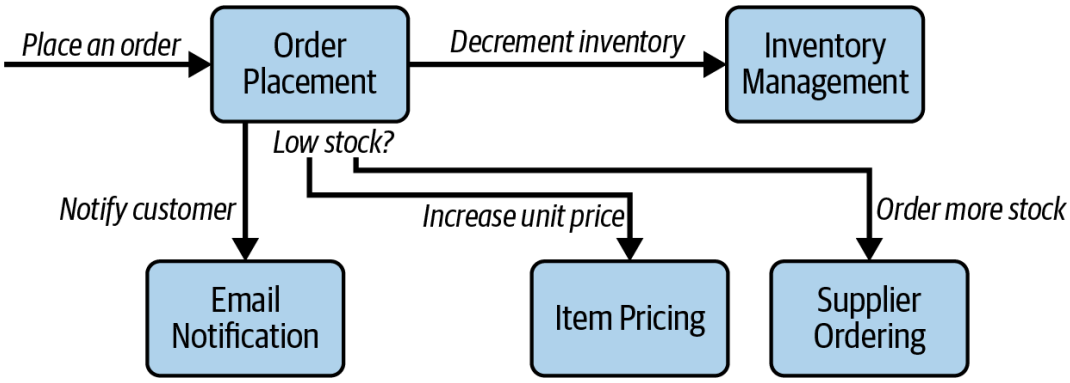

Figure 8-14. The `Order Placement` component is highly coupled to the rest of the system because it has too much knowledge

Notice that, in this architecture, the `Order Placement` component is highly coupled to the other components. While it doesn't have the *responsibility* to perform those actions, it does have the *knowledge* that these actions need to occur—and more knowledge means tighter coupling.

The idea behind the Law of Demeter is to limit each component's *knowledge* about the rest of the system. In Figure 8-14, the `Order Placement` component knows that the inventory must be decremented, more stock may need to be ordered, the item price may need to be adjusted, and an email must be sent to the customer. (That's a lot of knowledge!) But what if that knowledge could be distributed to other components? If so, then the architect can effectively decouple that component from the rest of the system.

To see how the Law of Demeter can be applied to reduce component coupling, refer to the system in Figure 8-14. If the architect added another component between `Order Placement` and `Inventory Management` to defer that *knowledge* that inventory must be decremented, the `Order Placement` component would still have the same efferent (outgoing) coupling level. Therefore, that coupling point needs to remain as is.

However, what about the knowledge that, if the supply goes down too low, the system must order more stock and adjust the item price? Both of those pieces of knowledge *can* be deferred to the `Inventory Management` component, thereby reducing the coupling level of the `Order Placement` component. Figure 8-15 illustrates the resulting architecture after applying the Law of Demeter. Removing the *knowledge* that certain functions need to happen makes the `Order Placement` component less coupled to the rest of the system.

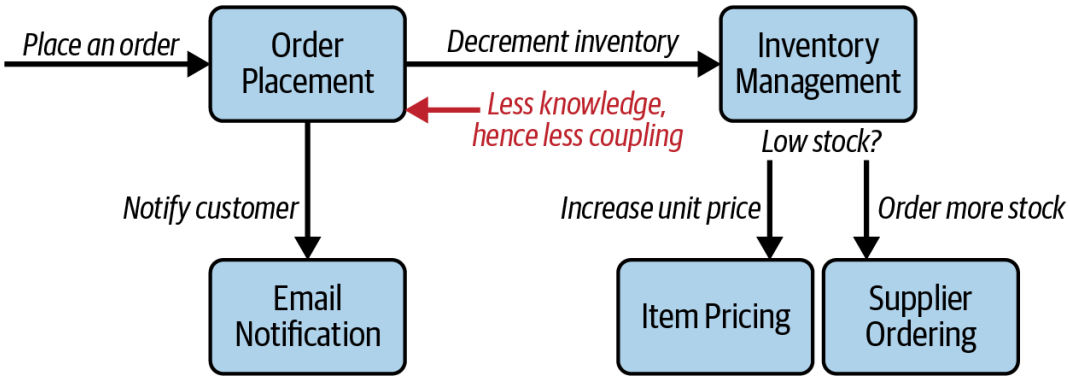

Figure 8-15. The `Order Placement` component is less coupled to the system when it has less knowledge

The astute reader will observe that, while applying the Law of Demeter reduced the coupling level of the `Order Placement` component, it also *increased* the

coupling level of the `Inventory Management` component. Applying the Law of Demeter does not necessarily reduce the system-wide level of coupling; rather, it usually redistributes that coupling to different parts of the system.

Case Study: Going, Going, Gone—Discovering Components

If a team has no special constraints and is looking for a good general-purpose component decomposition, the Actor/Actions approach works well as a generic solution.

If the architect applies the Actor/Action approach to Going, Going, Gone (GGG), they'll find that this system has three main roles: the *bidder*, the *auctioneer*, and the *system*—a frequent participant in this modeling technique for internal actions. The roles interact with the system using the following main actions:

`Bidder`

- View live video stream.
- View live bid stream.
- Place a bid.

`Auctioneer`

- Enter live bids into system.
- Receive online bids.
- Mark item as sold.

`System`

- Start auction.
- Make payment.
- Track bidder activity.

Given these actions, the architect can build a set of starter components for GGG and then iterate on it. One such solution appears in Figure 8-16.

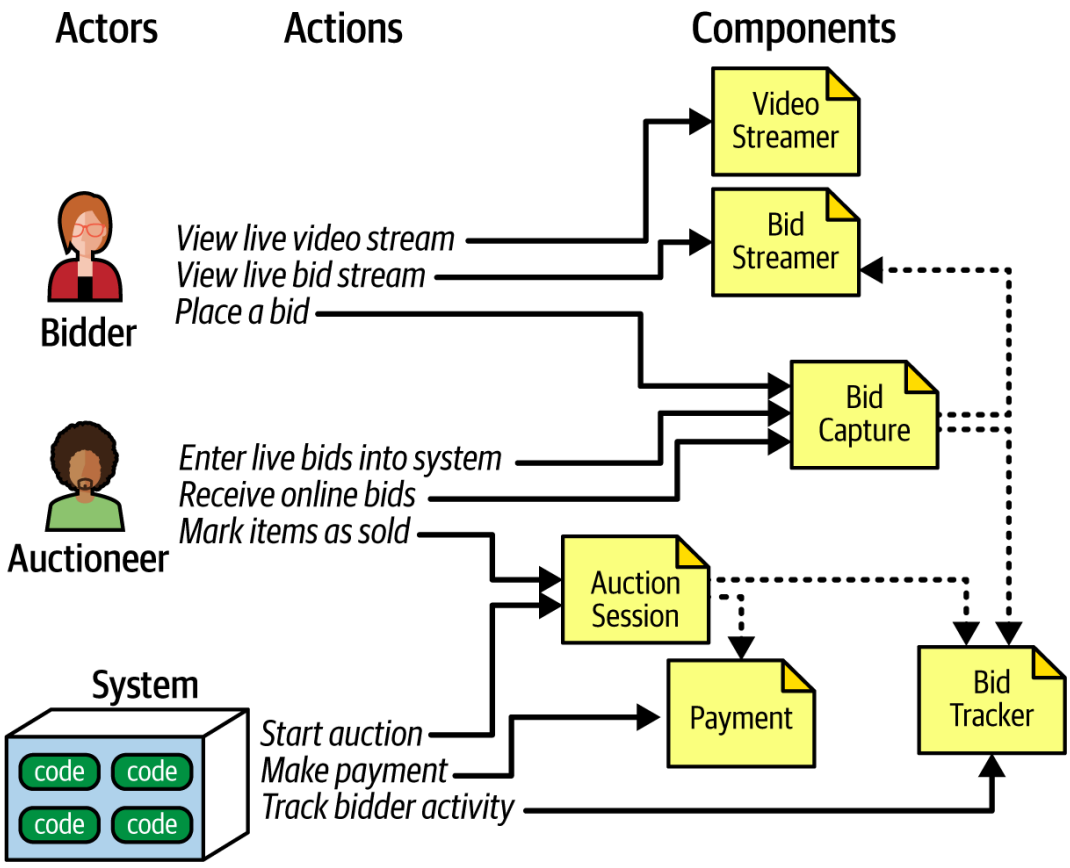

Figure 8-16. Initial set of components for Going, Going, Gone

In Figure 8-16, each role and action maps to a component. The components may need to collaborate to share information. These are the components we identified for this solution:

`Video Streamer`

: Streams a live auction to users.

`Bid Streamer`

: Streams bids to users as they occur. Both `Video Streamer` and `Bid Streamer` offer bidders a read-only view of the auction.

`Bid Capture`

: This component captures bids from the auctioneer and bidders.

`Bid Tracker`

: Tracks bids and acts as the system of record.

`Auction Session`

: Starts an auction. When the bidder wins and the auction for that item ends, this component triggers the payment and resolution steps, including notifying bidders of the next item up for bid.

`Payment`

: A third-party payment processor for credit card payments.

After the initial round of component identification (see Figure 8-6), the architect analyzes the previously identified architecture characteristics to determine if any

of them will change the design of the component. For example, the current design features a `Bid Capture` component to capture bids from both bidders and the auctioneer. This makes sense functionally: bids from anyone can be captured and handled the same way. But what previously identified architecture characteristics does bid capture call for? The auctioneer doesn't need the same level of scalability or elasticity as the bidders, who could potentially number in the thousands.

By the same token, the auctioneer might need certain previously identified architecture characteristics more than other parts of the system do—like reliability (ensuring connections don't drop) and availability (ensuring the system stays up). For example, while it would be bad for business if a bidder can't log in to the site or suffers from a dropped connection, it would be disastrous if either of those things were to happen to the auctioneer.

To support the bidders' and auctioneer's different levels of need for the same architecture characteristics, the architect decides to split the `Bid Capture` component into two components: `Bid Capture` and `Auctioneer Capture`. The updated design appears in Figure 8-17.

The architect creates a new component for `Auctioneer Capture`. They also update the information links from the `Auctioneer Capture` to both `Bid Streamer` (to show the online bidders the live bids) and `Bid Tracker` (to manage the bid streams). `Bid Tracker` is now the component that will unify two very different information streams: the auctioneer's single stream of information and the multiple streams from bidders.

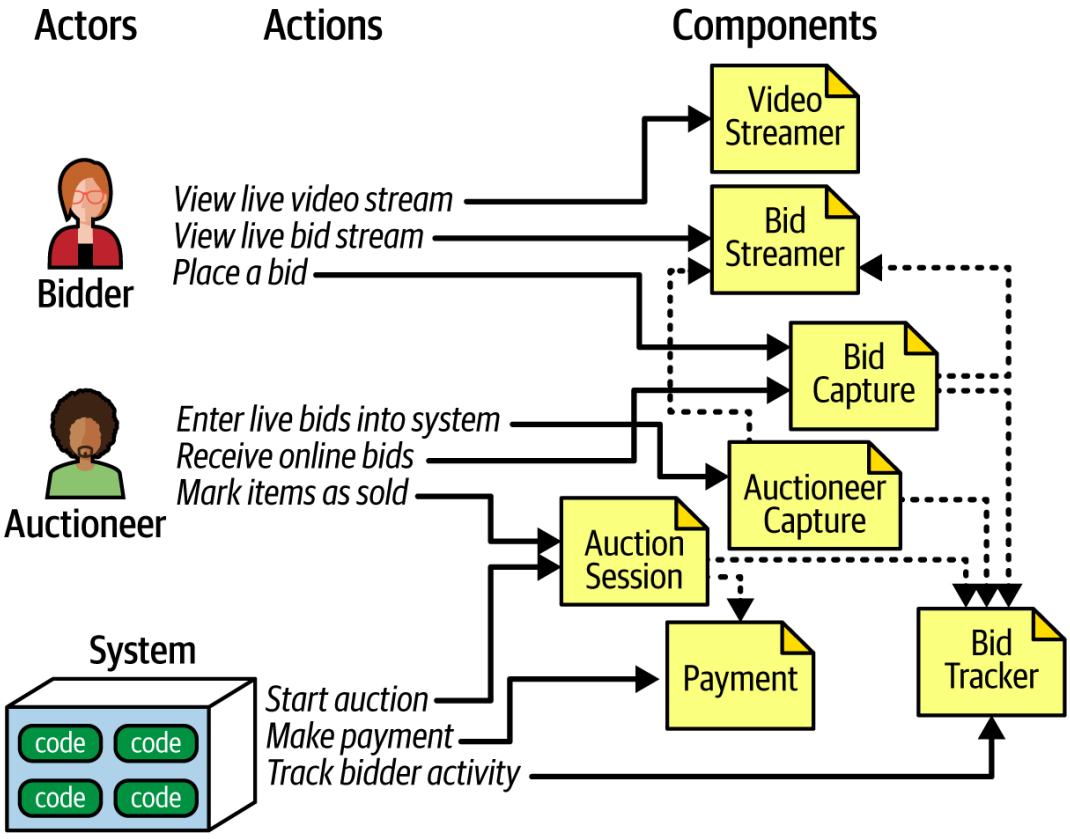

Figure 8-17. Revised components of the GGG case study

The design shown in Figure 8-17 is unlikely to be the final design. More requirements must be uncovered: how people will register new accounts, how payment functions will be administered, and so on. However, this design provides a good starting point for iterating further.

This is one possible set of components to solve the GGG problem—but it's not necessarily the best nor the only one. There are few software systems that can be implemented in only one way. Every design has different sets of trade-offs. As an architect, don't obsess over finding the "one true design," because there are many that will suffice. Try to assess the trade-offs between different design decisions as objectively as possible, and choose the one that has the "least worst" set of trade-offs.