Chapter 11. The Modular Monolith Architecture Style

Thanks to the widespread adoption of domain-driven design (DDD), as well as increased focus on domain partitioning, the *modular monolith* architectural style has gained so much popularity since we wrote the first edition of this book in 2020 that we decided to add a chapter to the second edition describing (and rating) it.

Topology

As the name suggests, the modular monolith architecture style is a *monolithic* architecture. As such, it's deployed as a single unit of software: a web archive (WAR) file, a single assembly in .NET, an enterprise archive (EAR) file in the Java platform, and so on. Because modular monolith is considered a *domain-partitioned* architecture (one organized by business domains rather than technical capabilities), its isomorphic shape is defined as *a single deployment unit with functionality grouped by domain area*. Figure 11-1 illustrates the typical topology for modular monolith.

To get a sense of the nature of modular monolith's domain focus, consider the traditional layered architecture (described in Chapter 10). Its components are defined and organized by their *technical* capabilities: Presentation, Business, and Persistence layers, and so on. For example, the presentation logic for maintaining customer profile information might be represented by a component with the namespace *com.app.presentation.customer.profile*. The third node in the namespace represents the layer's *technical* concern (in this case, the Presentation layer).

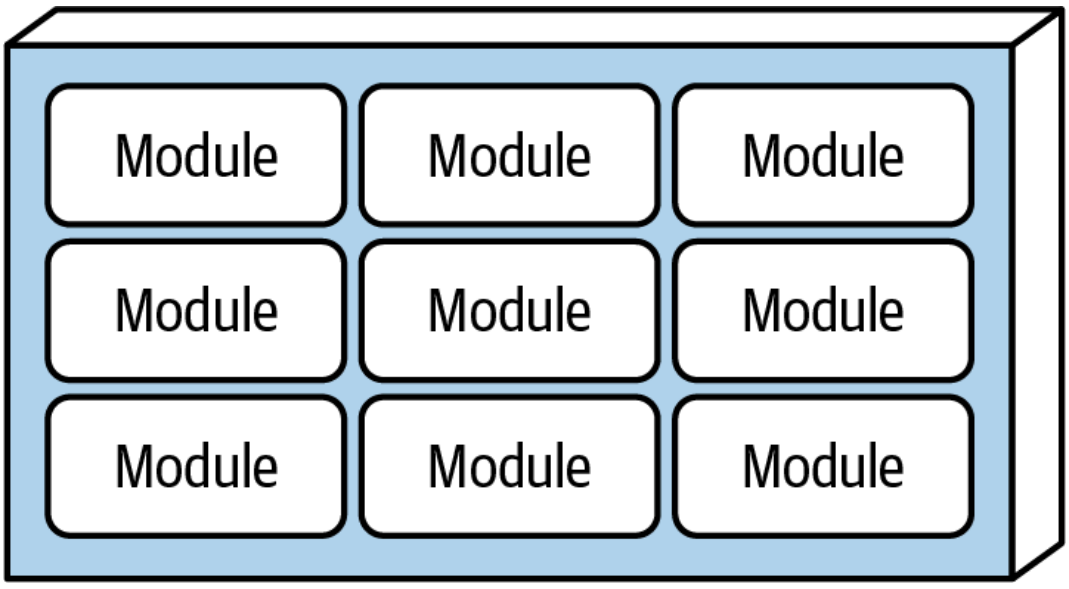

Figure 11-1. With the modular monolith architecture style, functionality is grouped by domain area

Conversely, modular monolith components are primarily organized by *domain*. As such, in a modular monolith architecture, the same customer profile maintenance component would be represented by the namespace *com.app.customer.profile*. Here, the third node of the namespace refers to a *domain* concern rather than a technical one. Depending on the complexity of the component, the namespace might further be divided by technical concern *after* the domain concern, such as *com.app.customer.profile.presentation* or *com.app.customer.profile.business*.

Style Specifics

Domains (or subdomains, in some cases) are called *modules* in this architectural style. Modules can be organized in one of two ways. The simplest architecture is a *monolithic structure*, where all of the modules and corresponding logical components are contained within the same code base, delineated by the namespace or directory structure they are contained in. A slightly more complex option is the *modular structure*, where each module is represented as an independent, self-contained artifact (such as a JAR or DLL file), and the modules are combined into one monolithic unit of software during deployment.

As with everything in software architecture, the choice between these two structural options depends on many factors and trade-offs. The following sections outline both and discuss their corresponding trade-offs.

Monolithic Structure

In the monolithic structure, all of the modules representing the system are contained in a single source-code repository. All the code associated with each module is deployed as a single unit when delivering or releasing the software. This structural option is illustrated in Figure 11-2. Each module is represented by a separate high-level directory containing the components and any subdomains that make up that module.

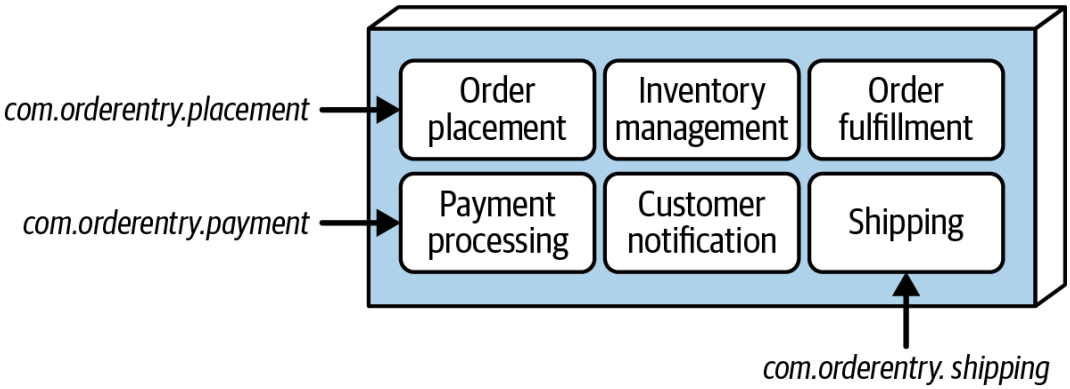

Figure 11-2. An example of the monolithic structure option

The following namespaces illustrate what a modular monolith might look like for the architecture shown in Figure 11-2:

```
com.orderentry.orderplacement
com.orderentry.inventorymanagement
com.orderentry.paymentprocessing
com.orderentry.notification
com.orderentry.fulfillment
com.orderentry.shipping
```

This is the simplest option for the modular monolith: all of the system's source code is located in one place and is thus more easily maintained, tested, and deployed. However, strict governance is needed (see "Governance") to maintain the boundaries of each module. Although this structural option is simple, developers have a tendency to reuse too much code across modules, as well as allowing too much communication between modules (see "Module Communication"). These practices can turn a well-architected modular monolith into an unstructured Big Ball of Mud.

Modular Structure

With the modular structure, modules are represented as self-contained artifacts (such as JAR and DLL files), then put together into a single deployment unit during deployment. Figure 11-3 illustrates this option using JAR files in the Java platform.

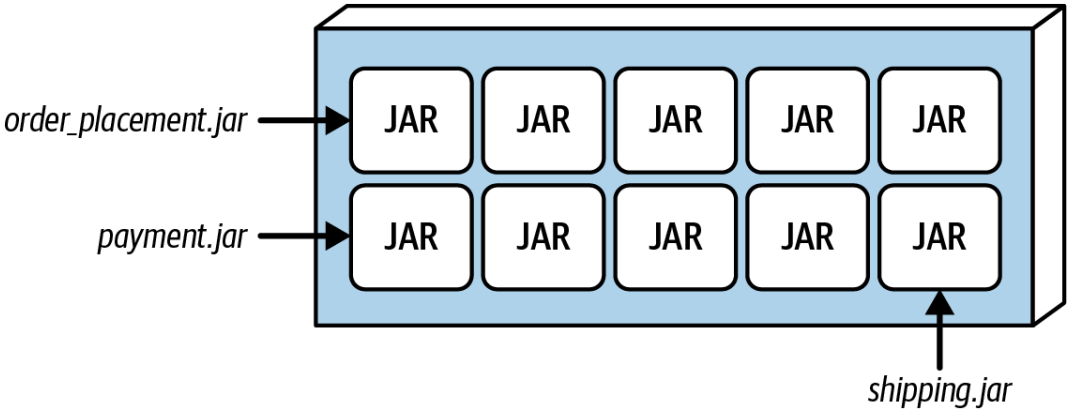

Figure 11-3. An example of the modular structure option using JAR files

The advantage of this structure is that each module is self-contained, allowing teams to work on separate modules (see "Team Topology Considerations"), many times even within the team's own dedicated source code repository for those modules. This option works well when the modules are largely independent of other modules. It also works well for larger, more complex systems where each module requires a different kind of expertise or business knowledge. With the modular structure option, developers are less apt to reuse code too much or to have modules communicate too much with one another (see "Module Communication"). This option also tends to produce cleaner boundaries between modules and better overall separation of concerns.

However, this structural option loses its effectiveness when modules that are dependent on one other need to communicate. Where this is the case, the monolithic structure approach is more effective.

Module Communication

Communication between modules is never a good thing in this architectural style, but we do acknowledge that in many cases it's necessary. For instance, in the architecture shown in Figure 11-2, the `OrderPlacement` module must communicate with the `InventoryManagement` module to have it adjust the inventory for the item ordered and perform any additional processing (for example, to order more stock if the inventory is too low). It also has to communicate with the `PaymentProcessing` module to apply payment for an order. Two primary options exist for communicating between modules, which we describe in the following sections.

Peer-to-peer approach

The most straightforward solution is simple peer-to-peer communication between modules. With this approach, a class file in one module instantiates a class in another module and invokes the necessary method(s) in that class to perform the operation (see Figure 11-4).

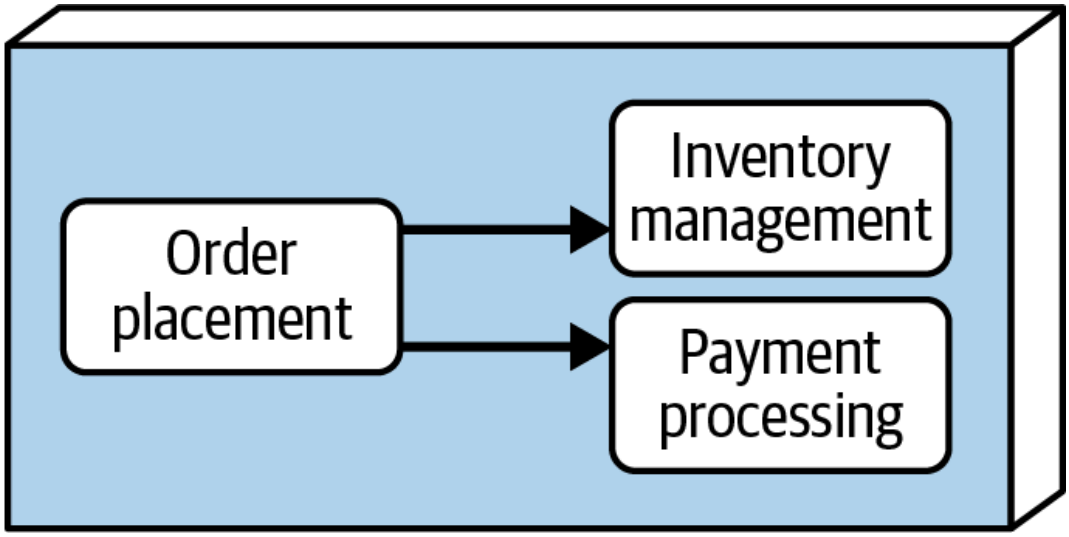

Figure 11-4. Peer-to-peer communication between modules

One issue with the monolithic approach is that it's *too* convenient for developers to instantiate any class contained within another module. This makes it easy to go from a well-structured architecture to the Big Ball of Mud antipattern (see Figure 9-1).

With the modular structure, however, the classes contained in another module may be located in separate, external artifacts (JARs or DLL files) rather than a separate directory in the source code repository. A module that communicates with other modules won't compile unless it has the class references, which means the developer has to form a *compile-time* dependency between those modules. The usual response to this issue is to create a shared interface class between those modules (in a separately shared JAR or DLL file) so that each module can compile independently of other modules. Either way, too much communication between modules using the modular structure approach results in the *DLL Hell* antipattern (or, in the Java platform, the *JAR Hell* antipattern).

Mediator approach

The *mediator* approach decouples modules by using a mediator component to form an abstraction layer between modules. The mediator acts as an orchestrator, accepting requests and delivering them to the appropriate modules. Figure 11-5 illustrates this approach.

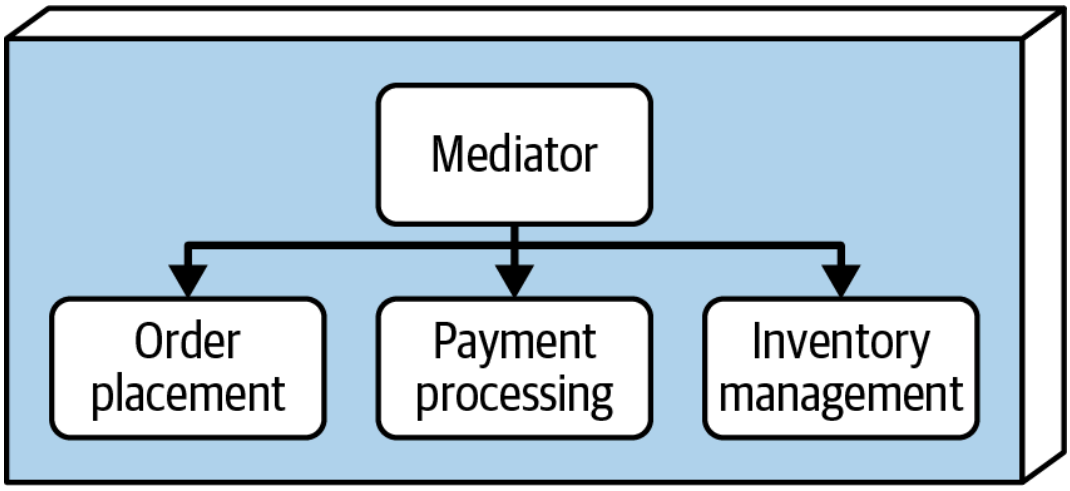

Figure 11-5. The mediator decouples modules, so they don't have to communicate with each other

The astute reader will observe that, while the mediator approach decouples modules, each module is effectively coupled to the mediator. This approach doesn't remove *all* coupling and dependencies, but it does simplify the architecture and keep the modules independent from each other. Note that it is the *mediator*, not the dependent modules, that needs some sort of API or interface to invoke the functionality in other modules.

Data Topologies

Because the modular monolith architecture is usually deployed as a single unit of software, it typically relies on a monolithic database topology. Using a single database helps reduce communication between modules, since the data is shared. However, if the modules are independent from each other and perform specific functions, they can also have their own databases containing specific contextual data, even though the architecture itself is monolithic. Figure 11-6 illustrates these two database topology options.

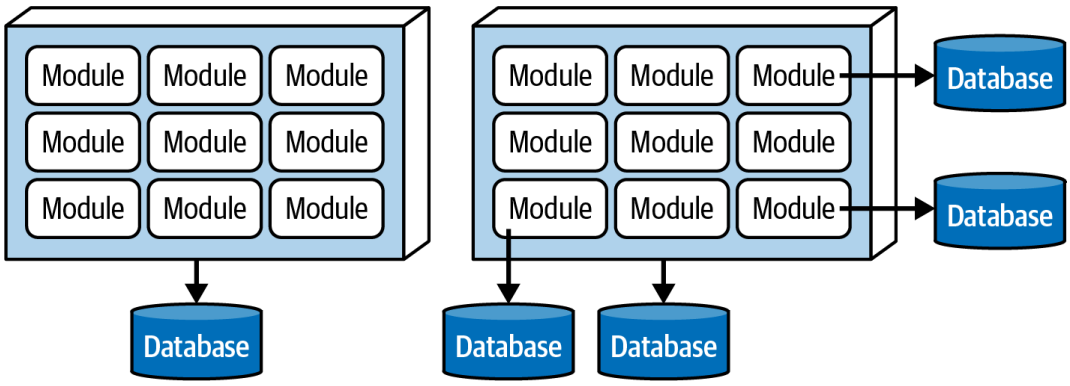

Figure 11-6. Data can be monolithic or modules can have their own databases

Cloud Considerations

Although modular monolithic architectures can be deployed in cloud environments (particularly if it's a small system), they're not generally well suited for cloud deployment: their monolithic nature renders them less likely to be able to take advantage of the on-demand provisioning that cloud environments offer. That said, smaller systems implemented in this architectural style can still leverage many cloud services, such as file storage, database, and messaging.

Common Risks

As with any monolithic system, the primary risk with the modular monolith architectural style is that it can get too big to properly maintain, test, and deploy. Monolithic architectures in and of themselves aren't bad; it's when they get too big that problems start to occur. What "too big" means varies from system to system, but here are some of the warning signs that the system might be too big:

- Changes take too long to make.
- When one area of the system is changed, other areas unexpectedly break.
- Team members get in each other's way when applying changes.
- It takes too long for the system to start up.

Another risk is going overboard with code reuse. Code reuse and sharing are a necessary part of software development, but in this architecture style, too much code reuse blurs the module boundaries, leading the architecture into the risky territory of the *unstructured monolith*: a monolithic architecture with such highly interdependent code that it cannot be unraveled.

Too much intermodule communication is another risk in this architectural style. Ideally, modules should be independent and self-contained. As we've noted, it's normal (and sometimes necessary) for some modules to communicate with others, particularly within a complex workflow. However, if there's too much intercommunication between modules, it's a good indication that the domains may have been ill-defined in the first place. In such cases, it's worth putting additional thought into redefining the domains to accommodate complex workflows and interdependencies.

Governance

The primary artifact in the modular monolith style is a *module*, which represents a particular domain or subdomain and is usually represented through the directory structure or namespace (or package structure in the Java platform). Therefore, one of the first forms of automated governance architects can apply is defining and ensuring compliance with the modules used in the architecture.

To write automated governance checks, architects use a host of tools, including ArchUnit for the Java platform, ArchUnitNet and NetArchTest for the .NET platform, PyTestArch for Python, and TSArch for TypeScript and JavaScript. The pseudocode in Example 11-1 ensures that all of the source code represented in the architecture example shown in Figure 11-2 falls under one of the listed namespaces that represent each defined module in the system.

Example 11-1. Pseudocode for ensuring the code follows the system's defined modules

```
# The following namespaces represent the modules in the system
LIST module_list = {
   com.orderentry.orderplacement,
   com.orderentry.inventorymanagement,
   com.orderentry.paymentprocessing,
   com.orderentry.notification,
   com.orderentry.fulfillment,
   com.orderentry.shipping
   }
```

```
# Get the list of namespaces in the system
LIST namespace_list = get_all_namespaces(root_directory)

# Make sure all the namespaces start with one of the listed modules
FOREACH namespace IN namespace_list {
   IF NOT namespace.starts_with(module_list) {
      send_alert(namespace)
   }
}
```

If a developer creates any additional high-level namespaces or directories outside the defined modules and their corresponding namespaces (or directories), they will receive an alert indicating the source code is not in compliance with the architecture.

This form of governance works well with the monolithic structure option of this architecture style (see "Monolithic Structure"), but is challenging with the modular structure option (see "Modular Structure") because the code might not be contained in the same monolithic source code repository. With the modular structure option, each module must be tested separately, as shown in Example 11-2.

Example 11-2. Pseudocode for validating the `InventoryManagement` module

```
# Get the list of namespaces in the system
LIST namespace_list = get_all_namespaces(root_directory)

# Make sure all the namespaces start with com.orderentry.inventorymanagement
FOREACH namespace IN namespace_list {
   IF NOT namepace.starts_with("com.orderentry.inventorymanagement") {
      send_alert(namespace)
   }
}
```

Another way to govern a modular monolith architecture is to control the amount of communication between modules. Defining what is "too much" communication is highly subjective and varies from system to system, but for the most part, architects should try to minimize the number of interdependencies between modules. Example 11-3 shows pseudocode for making sure the maximum total interdependency doesn't exceed a limit of five communication (or coupling) points.

Example 11-3. Pseudocode for limiting any given module's total number of dependencies

```
# Walk the directory structure, gathering modules and the source code files
# contained within those modules
LIST module_list = {
   com.orderentry.orderplacement,
   com.orderentry.inventorymanagement,
   com.orderentry.paymentprocessing,
   com.orderentry.notification,
   com.orderentry.fulfillment,
   com.orderentry.shipping
   }

MAP module_source_file_map
FOREACH module IN module_list {
  LIST source_file_list = get_source_files(module)
  ADD module, source_file_list TO module_source_file_map
}

# Determine how many references exist for each source file and send an alert if
# the system's total dependency count is greater than 5
```

```
FOREACH module, source_file_list IN module_source_file_map {
  FOREACH source_file IN source_file_list {
    incoming count = used_by_other_module(source_file, module_source_file_map) {
    outgoing_count = uses_other_module(source_file) {
    total_count = incoming count + outgoing count
  }
  IF total_count > 5 {
    send_alert(module, total_count)
  }
}
```

A final form of automated governance is to ensure that modules stay independent of one another by restricting one specific module from talking to another module. For example, in Figure 11-2, the `OrderPlacement` module should not communicate with the `Shipping` module. Example 11-4 shows the ArchUnit code in Java to govern this dependency.

Example 11-4. ArchUnit code for governing dependency restrictions between specific modules

```
public void order_placement_cannot_access_shipping() {
   noClasses().that()
   .resideInAPackage("..com.orderentry.orderplacement..")
   .should().accessClassesThat()
   .resideInAPackage("..com.orderentry.shipping..")
   .check(myClasses);
}
```

Team Topology Considerations

Because the modular monolith is considered a domain-partitioned architecture, it works best when teams are also aligned by domain area (such as cross-functional teams with specialization). When a domain-based requirement comes along, a domain-focused, cross-functional team can work together on that feature, from the presentation logic all the way to the database. Conversely, teams organized by technical categories (such as UI teams, backend teams, database teams, and so on) do not work well with this architectural style, primarily due to its domain partitioning. Assigning domain-based requirements to technically organized teams requires a lot of communication and collaboration, which often proves difficult.

Here are some considerations for aligning the specific team topologies outlined in "Team Topologies and Architecture" with the modular monolith style:

Stream-aligned teams

> Stream-aligned teams generally own the flow through the system from beginning to end, nicely matching the monolithic and generally self-contained shape of the modular monolith.

Enabling teams

> Due to this style's high level of modularity and separation of concerns, enabling team topologies also works well. Specialists and cross-cutting team members can make suggestions and perform experiments by introducing additional modules to the system, with minimal impact to other existing modules.

Complicated-subsystem teams

> Each module in a modular monolith architecture generally performs a specific role based on its domain or subdomain (such as

`PaymentProcessing`). This works well with the complicated-subsystem team topology, because different team members can focus on complicated domain or subdomain processing independent of other team members (and modules).

Platform teams

Developers can leverage the benefits of the platform-teams topology by utilizing common tools, services, APIs, and tasks, primarily due to the high degree of modularity found in this architectural style.

Style Characteristics

A one-star rating in the characteristics ratings table Figure 11-7 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The characteristics contained in the scorecard are described and defined in Chapter 4.

The modular monolith architecture style is a *domain-partitioned* architecture because its application logic is partitioned into modules. Because it is usually implemented as a monolithic deployment, its architectural quantum is typically 1.

Overall cost, simplicity, and modularity are the primary strengths of the modular monolith architecture style. Being monolithic in nature, these architectures don't have the complexities associated with distributed architecture styles. They're simpler and easier to understand, and relatively low cost to build and maintain. Architectural modularity is achieved through the separation of concerns between the various modules, representing domains and subdomains.

Deployability and testability, while only two stars, rate slightly higher in modular monolith than the layered architecture due to its level of modularity. That said, this architecture style is still a monolith: as such, ceremony, risk, frequency of deployment, and completeness of testing negatively impact these scores.

Modular monolith architecture elasticity and scalability rate very low (one star), primarily due to monolithic deployments. Although it is possible to make certain functions within a monolith scale more than others, this effort usually requires very complex design techniques (such as multithreading, internal messaging, and other parallel-processing practices) for which this architecture isn't well suited.

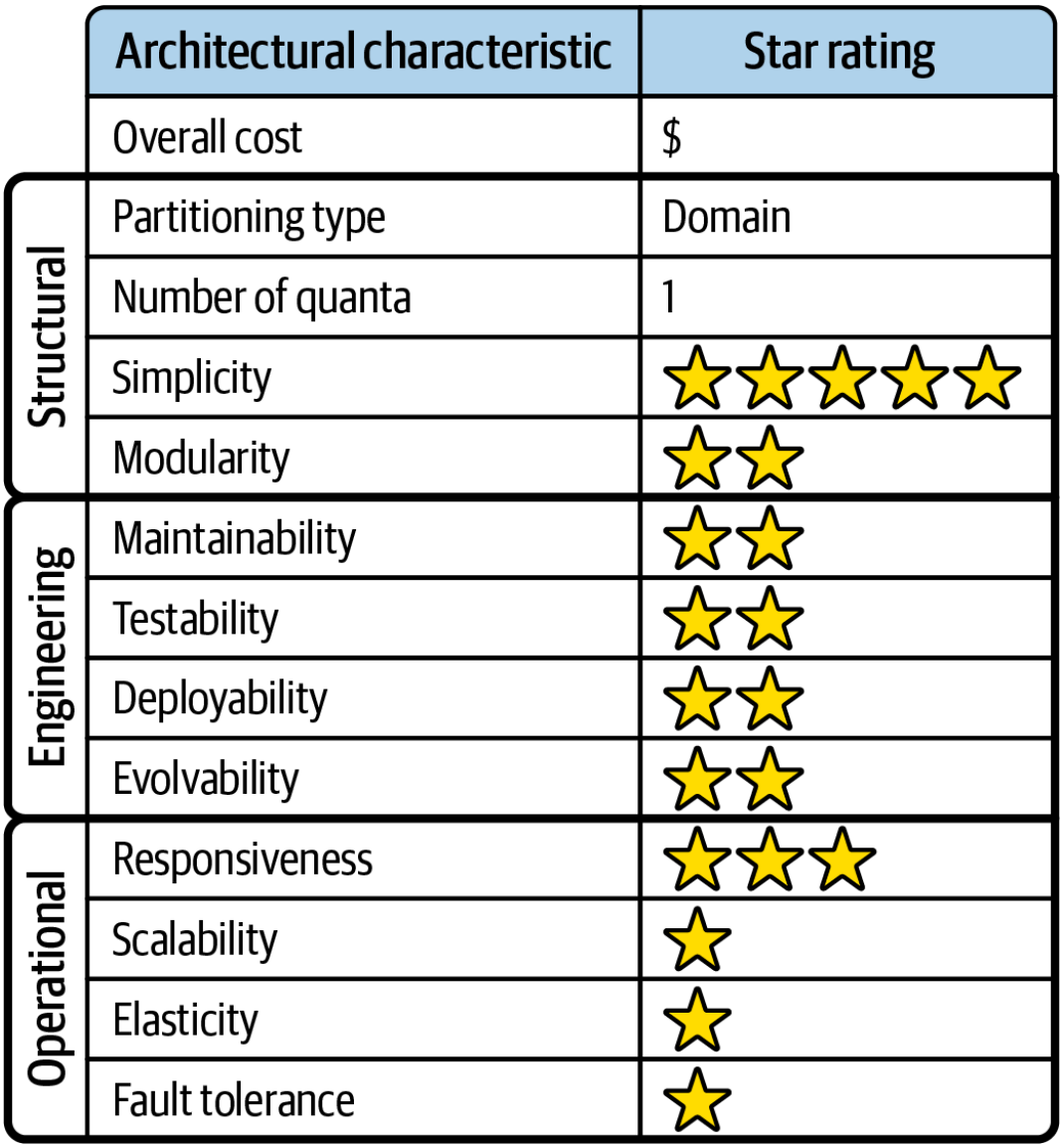

Figure 11-7. Architectural characteristics star ratings for the modular monolith

Modular monolith architecture monolithic deployments don't support fault tolerance. If one small part of the architecture causes an out-of-memory condition, the entire application unit crashes. Furthermore, as in most monolithic applications, overall availability is affected by the high mean time to recovery (MTTR), with startup times usually measured in minutes.

When to Use

Because of its simplicity and low cost, the modular monolith architecture style is a good choice when faced with tight budget and time constraints. It's also a good choice for starting out with a new system. If the system's architectural direction is still unclear, it's often more effective to begin with a modular monolith and later move to a more complicated and expensive distributed architecture style, such as service based (see Chapter 14) or microservices (see Chapter 18), than to jump straight into the distributed architecture.

Modular monolith is also a good choice for domain-focused teams, such as cross-functional teams with specialization. This allows each team to focus on a specific module within the architecture from end to end, with minimal coordination with other domain teams. This architectural style is also well suited for situations where a majority of changes to the system are domain based (such as adding expiration dates to items in a customer's wishlist).

Lastly, because the modular monolith is a domain-partitioned architecture, it's well suited to teams engaging in DDD.

When Not to Use

The primary reason not to use this architectural style is when systems or products require high levels of certain operational characteristics, such as scalability, elasticity, availability, fault tolerance, responsiveness, and performance. Like most monolithic architectures, modular monolith is ill-suited for these architectural concerns.

Avoid using modular monolith when a majority of the changes are technically oriented, such as continuously replacing the user interface or database technology. Because this architecture is domain partitioned, such changes impact every module and usually require significant communication and coordination between domain teams. In these situations, the layered architecture style (see Chapter 10) is a much better choice.

Examples and Use Cases

EasyMeals is a new delivery-based neighborhood restaurant, catering to working people who don't always have time to cook meals after getting home from a busy day. Hungry customers can order a nice dinner online and have it delivered to their doorstep within an hour.

As a small, local restaurant, they don't have high scalability or responsiveness needs. And since their budget is limited, they don't want to spend a lot of money on an elaborate software system. The shape of this business problem makes the modular monolith a good choice for EasyMeals.

Figure 11-8 shows what EasyMeals' simple restaurant management system might look like using the modular monolith architectural style.

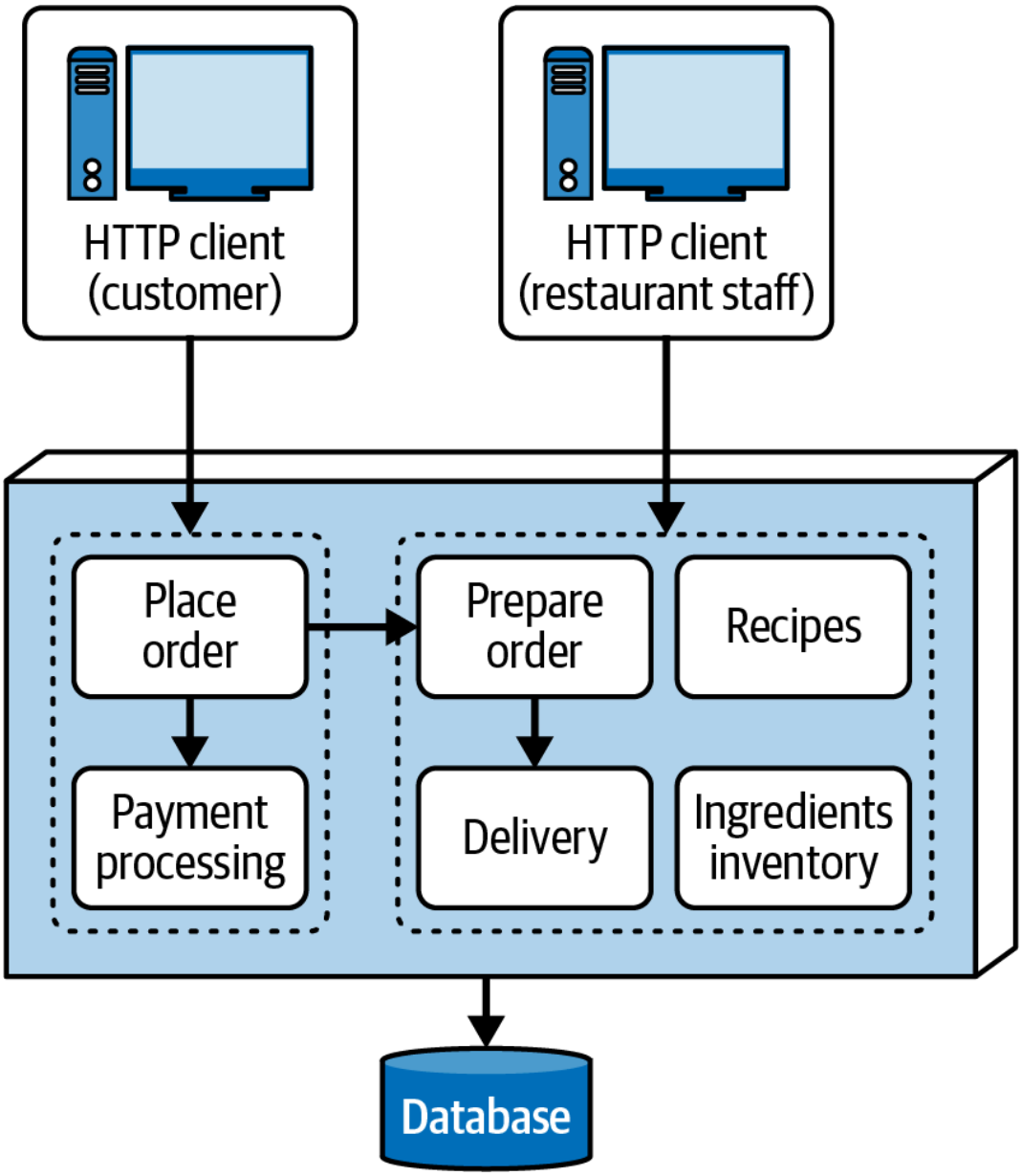

Figure 11-8. A small restaurant ordering and management system using the modular monolith style

Customers access the `PlaceOrder` and `PaymentProcessing` modules through a dedicated user interface through a different UI. The following namespaces represent each of the modules for this system:

```
com.easymeals.placeorder
com.easymeals.payment
com.easymeals.prepareorder
com.easymeals.delivery
com.easymeals.recipes
com.easymeals.inventory
```

The `PlaceOrder` module allows each customer to view the menu; select items; add their name, address, and payment information; and submit the order. The components for this module could be represented through the following namespaces, with source code implementing each of these main functions:

```
com.easymeals.placeorder.menu
com.easymeals.placeorder.shoppingcart
com.easymeals.placeorder.customerdata
com.easymeals.placeorder.paymentdata
com.easymeals.placeorder.checkout
```

This example illustrates that a module in the modular monolith is made up of one to many *components* (see Chapter 8).

The `PaymentProcessing` module is responsible for applying payment. EasyMeals accepts credit cards, debit cards, and PayPal; the modularity of this architecture makes it easy to add an additional payment type (such as loyalty points). Customers enter this information in the `PlaceOrder` module, which passes it to the `PaymentProcessing` module. The components for this module could be represented through the following namespaces:

```
com.easymeals.payment.creditcard
com.easymeals.payment.debitcard
com.easymeals.payment.paypal
```

Once the order is paid for, the `PlaceOrder` module communicates with the `PrepareOrder` module, which displays the entire order to the kitchen staff. After cooking, the kitchen staff marks the order as ready for delivery and it is then sent to the `Delivery` module. The following namespaces represent the components within the `PrepareOrder` module:

```
com.easymeals.prepareorder.displayorder
com.easymeals.prepareorder.ready
```

The `Delivery` module assigns a delivery person to the order and indicates the delivery address. It allows the delivery person to mark the order as delivered, ending the lifecycle for that particular order, and to record any issues (such as an aggressive dog at the front gate or a customer who isn't home). The following namespaces represent the components within the `Delivery` module:

```
com.easymeals.delivery.assign
com.easymeals.delivery.issues
com.easymeals.delivery.complete
```

The `Recipes` module allows the cooks and management staff to add items to the menu and maintain the list of ingredients and measurements for each menu item. The following namespaces represent the components within the `Recipes` module:

```
com.easymeals.recipes.view
com.easymeals.recipes.maintenance
```

Finally, the `IngredientsInventory` module makes sure that there are enough ingredients on hand for the recipes on the menu. This module is a bit more complex than the others: it has a sophisticated AI component that forecasts sales volume to automate the process of procuring ingredients for the week.

The following namespaces represent the components within the `IngredientsInventory` module:

```
com.easymeals.inventory.maintenance
com.easymeals.inventory.forecasting
com.easymeals.inventory.ordering
com.easymeals.inventory.suppliers
com.easymeals.inventory.invoices
```

And that's it! Modular monolith's simplicity and level of modularity makes it relatively easy to locate and maintain code to fix a bug or add a new feature. This illustrates the power of this simple and straightforward architectural style.