Chapter 13. Microkernel Architecture Style

The *microkernel* architecture style (also referred to as the *plug-in* architecture) was invented several decades ago and is still widely used today. This architecture style is a natural fit for *product-based applications*: that is, applications packaged and made available for download and installation as a single, monolithic deployment, typically installed on the customer's site as a third-party product. However, it is also widely used in nonproduct custom business applications, especially problem domains that require customization. For example, an insurance company in the US that has unique rules for each state, or an international shipping company that must adhere to various legal and logistical variations, would both benefit from this style.

Topology

The microkernel style is a relatively simple monolithic architecture consisting of two components: a core system and plug-ins. Application logic is divided between independent plug-in components and the basic core system, which isolates application features and provides extensibility, adaptability, and custom processing logic. Figure 13-1 illustrates the basic topology of the microkernel architecture style.

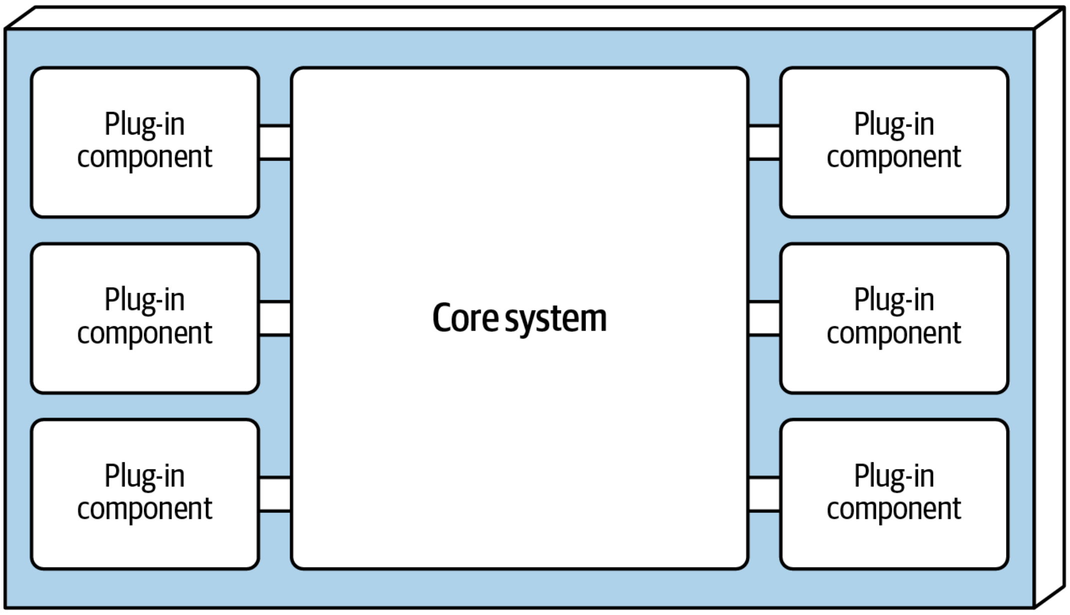


Figure 13-1. Basic components of the microkernel architecture style

Style Specifics

The essence of the microkernel architecture consists of two types of components: the *core system* and *plug-ins*.

Core System

The *core system* is formally defined as the minimal functionality required to run the system. Let's look at the Eclipse IDE for a good example of this. Eclipse's core system is just a basic text editor: open a file, change some text, and save the file. It's not until you add plug-ins that Eclipse starts becoming a usable product.

However, another definition of the core system is the *happy path*: a general processing flow through the application that involves little or no custom processing. A microkernel architecture takes the cyclomatic complexity of the application, removes it from the core system, and places it into separate plug-in components. This allows for better extensibility and maintainability, as well as increased testability.

To dive deeper, we'll return to Going Green, the electronic device recycling application we introduced in Chapter 7. Let's say you're working on Going Green's application, which must assess each electronic device it receives according to specific, custom assessment rules. The Java code for this sort of processing might look as follows:

```
public void assessDevice(String deviceID) {
   if (deviceID.equals("iPhone6s")) {
      assessiPhone6s();
   } else if (deviceID.equals("iPad1"))
      assessiPad1();
   } else if (deviceID.equals("Galaxy5"))
      assessGalaxy5();
   } else ...
      ...
   }
}
```

Rather than placing all this client-specific customization—which has lots of cyclomatic complexity—in the core system, you could create a separate plug-in component for each electronic device being assessed. Not only do specific client plug-in components isolate independent device logic from the rest of the processing flow, they also allow for expansion: adding a new device to assess is simply a matter of adding a new plug-in component and updating the registry. With the microkernel architecture style, assessing an electronic device only requires the core system to locate and invoke the corresponding device plug-ins, as illustrated in this revised source code:

```
public void assessDevice(String deviceID) {
        String plugin = pluginRegistry.get(deviceID);
        Class<?> theClass = Class.forName(plugin);
        Constructor<?> constructor = theClass.getConstructor();
        DevicePlugin devicePlugin =
                (DevicePlugin)constructor.newInstance();
        devicePlugin.assess();
}
```

In this example, all of the complex rules and instructions for assessing a particular electronic device are self-contained in a standalone, independent plug-in component that can be generically executed from the core system.

Depending on its size and complexity, you could implement the core system as a layered architecture or as a modular monolith (as illustrated in Figure 13-2). In some cases, you might even split the core system into separately deployed domain services, with each domain service containing plug-in components specific to that domain. For the sake of this example, we'll assume that you implement it as a layered architecture for Going Green.

Suppose `Payment Processing` is the domain service representing the core system. Each payment method (credit card, PayPal, store credit, gift card, and purchase order) would have a separate plug-in component specific to that payment domain.

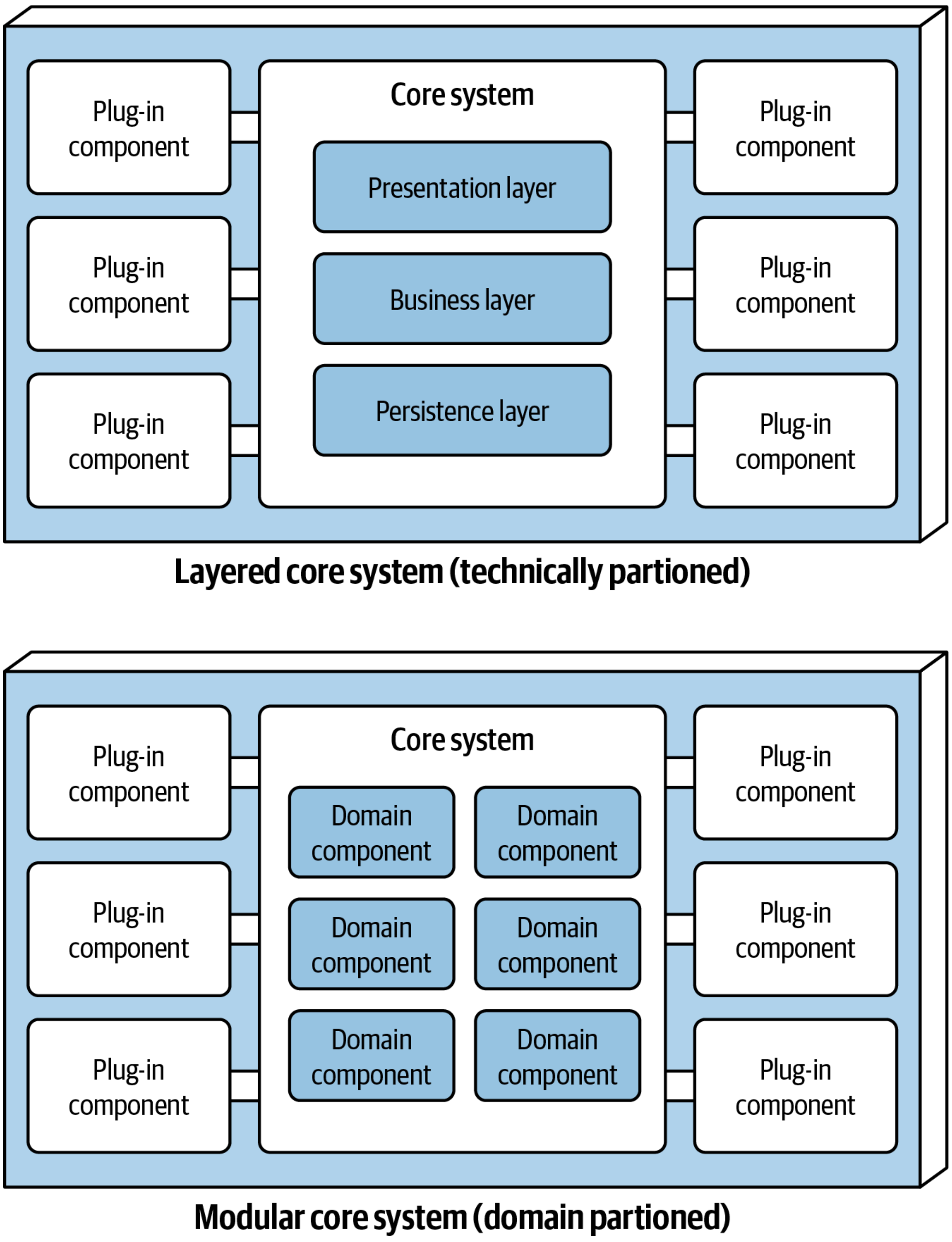


Figure 13-2. Variations of the microkernel architecture core system

The core system's Presentation layer can be embedded within the core system or implemented as a separate user interface, with the core system providing backend services. As a matter of fact, you could also implement a separate UI using a microkernel architecture style. Figure 13-3 illustrates these Presentation layer variants in relation to the core system.

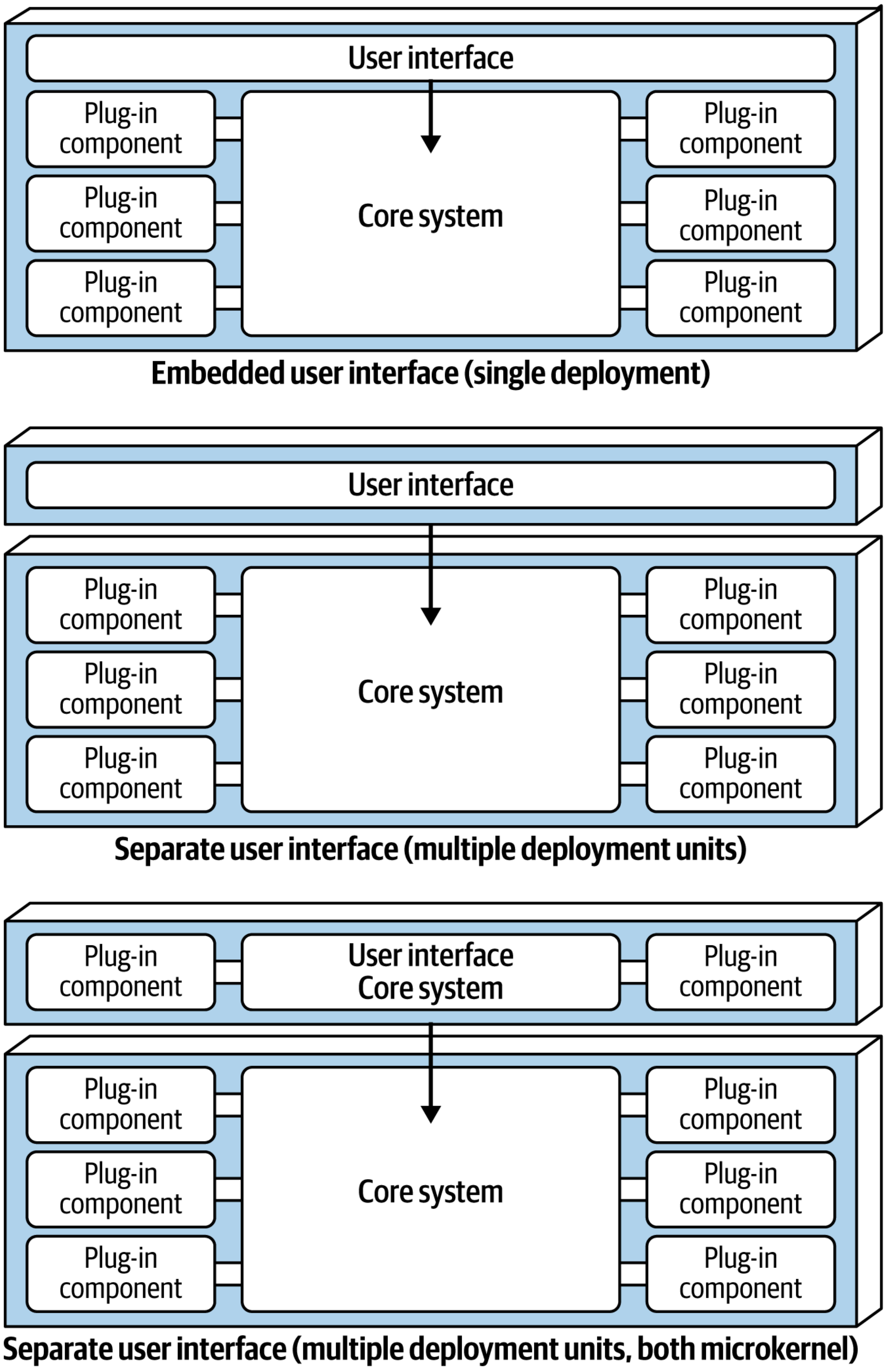

User interface
Plug-in component
Core system
Embedded user interface (single deployment)
Separate user interface (multiple deployment units)
User interface Core system
Separate user interface (multiple deployment units, both microkernel)

Figure 13-3. User interface variants

Plug-In Components

Plug-in components are standalone, independent components that contain specialized processing, additional features, and custom code meant to enhance or extend the core system. Additionally, they isolate highly volatile code, creating better maintainability and testability within the application. Ideally, plug-in components should have no dependencies between them.

Communication between the plug-in components and the core system is generally *point-to-point*, meaning the "pipe" that connects the plug-in to the core system is usually a method invocation or function call to the entry-point class of the plug-in component. In addition, the plug-in component can be either compile based or runtime based. *Runtime* plug-in components can be added or removed at runtime without redeploying the core system or other plug-ins, and are usually managed through frameworks such as Open Service Gateway Initiative (OSGi) for Java, Penrose (Java), Jigsaw (Java), or Prism (.NET). *Compile-based* plug-in components are much simpler to manage, but modifying, removing, or adding them requires the entire monolithic application to be redeployed.

Point-to-point plug-in components can be implemented as shared libraries (such as a JAR, DLL, or Gem), package names in Java, or namespaces in C#. In our Going Green application, each electronic device plug-in can be written and implemented as a JAR, DLL, or Ruby Gem (or any other shared library), with the name of the device matching the name of the independent shared library, as illustrated in Figure 13-4.

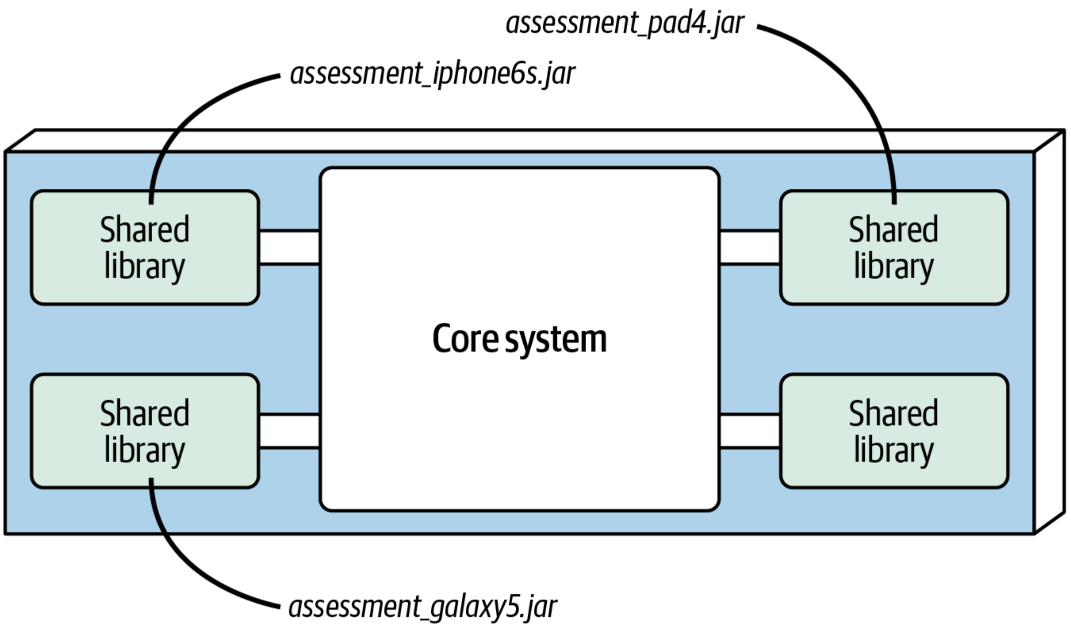


Figure 13-4. Shared library plug-in implementation

An easier approach, shown in Figure 13-5, is to implement each plug-in component as a separate namespace or package name within the same code base or IDE project. When creating the namespace, we recommend the following semantics: *app.plug-in.<domain>.<context>*. For example, consider the namespace *app.plug-in.assessment.iphone6s*. The second node (`plug-in`) makes it clear that this component is a plug-in and therefore should strictly adhere to the basic rules (they must be self-contained and separate from other plug-ins). The third node describes the domain (in this case, `assessment`), allowing plug-in components to be organized and grouped by a common purpose. The fourth node (`iphone6s`) describes the specific context for the plug-in, making it easy to locate the device plug-in for modification or testing.

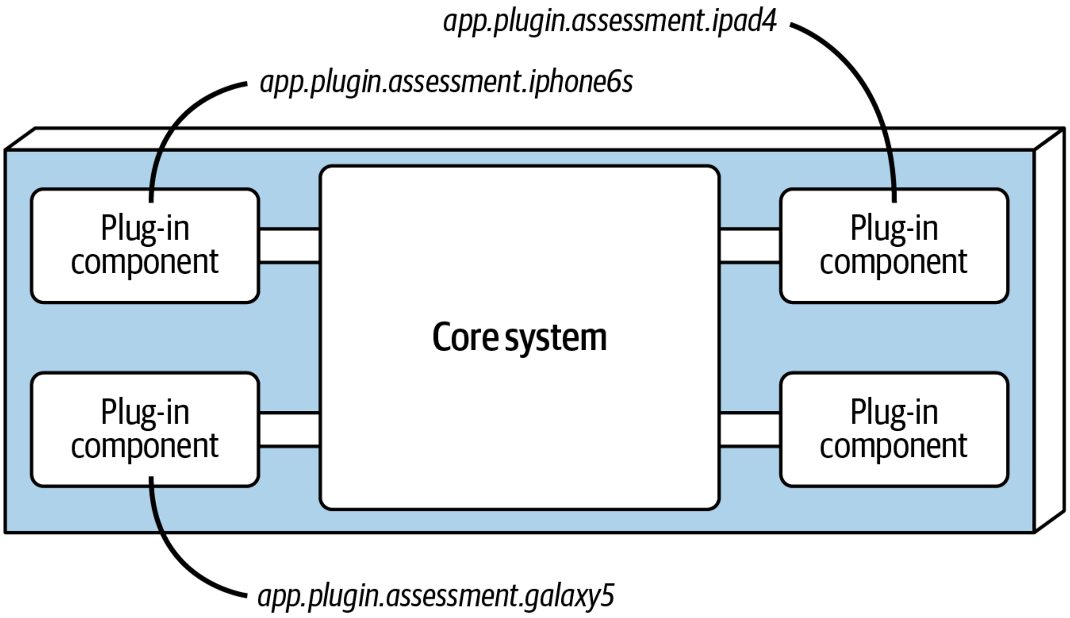

app.plugin.assessment.ipad4
app.plugin.assessment.iphone6s
Plug-in component
Plug-in component
Core system
Plug-in component
Plug-in component
app.plugin.assessment.galaxy5

Figure 13-5. Package or namespace plug-in implementation

Plug-in components do not always have to use point-to-point communication with the core system. Alternatives include using REST or messaging to invoke plug-in functionality, with each plug-in being a standalone service (or maybe even a microservice, implemented using a container). Although this may sound like a good way to increase overall scalability, note that this topology (illustrated in Figure 13-6) is still only a single architecture quantum, due to the monolithic core system. Every request must first go through the core system to get to the plug-in service.

The benefits of the remote-access approach to plug-in components implemented as individual services is that it allows for better overall component decoupling, better scalability and throughput, and runtime changes without any special frameworks (like OSGi, Jigsaw, or Prism). It also allows for asynchronous communications to plug-ins which, depending on the scenario, could significantly improve overall user responsiveness. Using the Going Green example, rather than having to wait for the electronic device assessment to run, the core system could make an asynchronous *request* to kick off an assessment for a particular device. When the assessment completes, the plug-in can notify the core system through another asynchronous messaging channel, which in turn notifies the user that the assessment is complete.

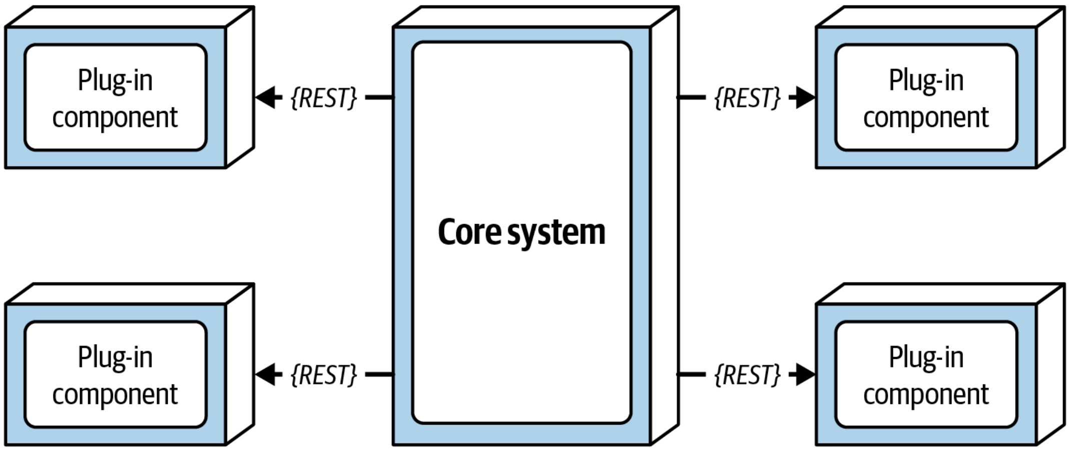


Figure 13-6. Remote plug-in access using REST

With these benefits come trade-offs. Remote plug-in access turns the microkernel architecture into a distributed architecture rather than a monolithic one, making it difficult to implement and deploy for most third-party on-prem products. Furthermore, it creates more overall complexity and cost, and complicates the overall deployment topology. If a plug-in becomes unresponsive or stops running, particularly if the system is using REST, the request cannot be completed. This would not be the case with a monolithic deployment. The choice between point-to-point and remote communication should be based on the project's specific requirements and a careful trade-off analysis.

The Spectrum of "Microkern-ality"

Not all systems that support plug-ins are microkernels, but all microkernels support plug-ins. A system's degree of "microkern-ality," as we call it, depends on how much standalone functionality exists in the core system. This is reflected in the spectrum illustrated in Figure 13-7.

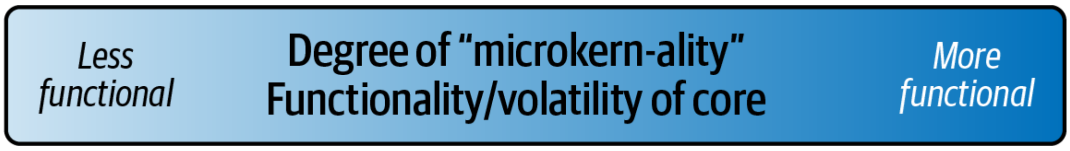


Figure 13-7. The spectrum of "microkern-ality"

In Figure 13-7, "pure" microkernel architectures (like the aforementioned Eclipse IDE or linter tools) have very little core functionality. For example, a linter parses source code and delivers the abstract syntax tree so that a developer can write rules about language use. The core parses the code, but until someone writes a plug-in to take advantage of it, it's of little use. Contrast that with a web browser, which supports plug-ins but is perfectly functional without them; browsers fall on the righthand side of the spectrum.

Determining the volatility of the core goes a long way toward helping architects decide between a system that just supports plug-ins or a more "pure" microkernel.

Registry

The core system needs to know which plug-in modules are available and how to get to them. One common way of implementing this is through a *plug-in registry*. This registry contains information about each plug-in module, including things like its name, data contract, and remote access protocol details (depending on how the plug-in is connected to the core system). For example, a plug-in for tax software that flags high-risk tax-audit items might have a registry entry that contains the name of the service (AuditChecker), the data contract (input data and output data), and the contract format (XML).

The registry can be as simple as an internal map structure owned by the core system containing a key and the plug-in component reference, or it can be as complex as a registry and discovery tool embedded within the core system or deployed externally (such as Apache ZooKeeper or Consul). Using the electronics recycling example, the following Java code implements a simple registry within the core system, showing examples of a point-to-point entry, a messaging entry, and a RESTful entry for assessing an iPhone 6S device:

```
Map<String, String> registry = new HashMap<String, String>();
static {
  //point-to-point access example
  registry.put("iPhone6s", "Iphone6sPlugin");

  //messaging example
  registry.put("iPhone6s", "iphone6s.queue");

  //restful example
  registry.put("iPhone6s", "https://atlas:443/assess/iphone6s");
}
```

Contracts

The contracts between plug-in components and the core system are usually standard across a domain of plug-in components and include behavior, input data, and output data returned from the plug-in component. Custom contracts are typically used in situations where the plug-in components are developed by a third party, so the architect has no control over the contract the plug-in uses. In such cases, it is common to create an adapter between the plug-in contract and your standard contract so that the core system doesn't need specialized code for each plug-in.

Plug-in contracts can be implemented in XML, in JSON, or even in objects passed back and forth between the plug-in and the core system. In the electronics recycling application, the following contract (implemented as a standard Java interface named `AssessmentPlugin`) defines the overall behavior (`assess()`, `register()`, and `deregister()`), along with the corresponding output data expected from the plug-in component (`AssessmentOutput`):

```
public interface AssessmentPlugin {
        public AssessmentOutput assess();
        public String register();
        public String deregister();
}

public class AssessmentOutput {
        public String assessmentReport;
        public Boolean resell;
        public Double value;
        public Double resellPrice;
}
```

In this contract example, the device assessment plug-in is expected to return the assessment report with:

- A formatted string
- A resell flag (true or false) indicating whether this device can be resold on a third-party market or safely disposed of
- If the item can be resold, its calculated value and recommended resell price

Notice the roles and responsibility model between the core system and the plug-in component in this example, specifically with the `assessmentReport` field. It is not the responsibility of the core system to format and understand the details of the assessment report, only to either print it out or display it to the user.

Data Topologies

Generally, teams implement microkernel architectures as monolithic architectures, using a single (typically relational) database.

It's uncommon practice for plug-in components to connect directly to a centrally shared database. Instead, the core system takes on this responsibility, passing whatever data is needed to each plug-in. The primary reason for this practice is decoupling. Making a database change should only affect the core system, not the plug-in components. That said, plug-ins can have separate data stores that are only accessible to that plug-in. For example, each device assessment plug-in in the Going Green system could have its own simple database or rules engine containing all of the specific assessment rules for that product. The data store owned by the plug-in component can be external (as shown in Figure 13-8), or embedded as part of the plug-in component or monolithic deployment (as in the case of an in-memory or embedded database).

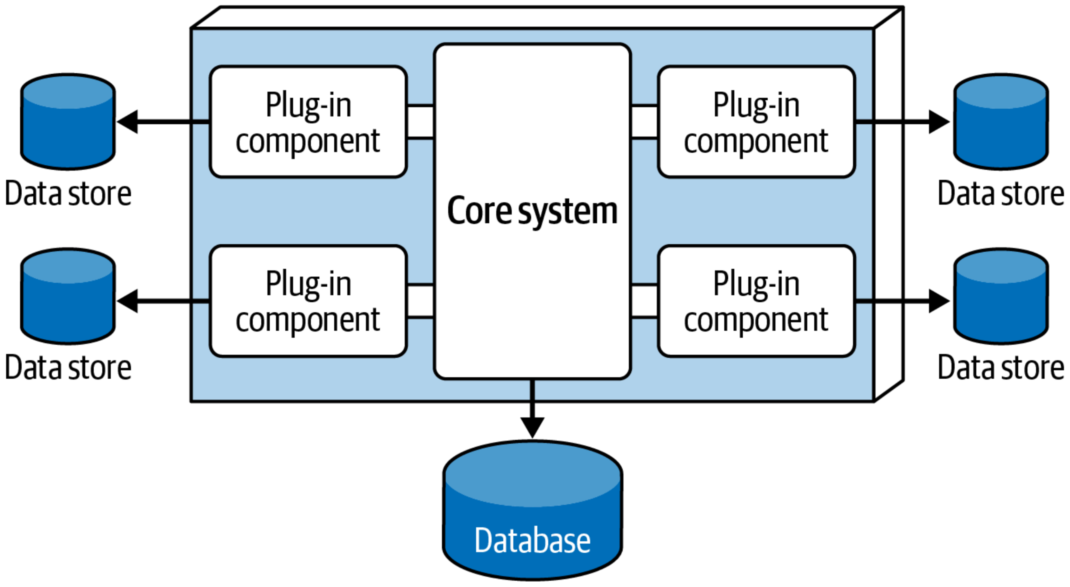


Figure 13-8. A plug-in component can own its own data store

Cloud Considerations

Because microkernel architectures are typically monolithic, the cloud represents a few coarse-grained options. The first option is to deploy the entire application on the cloud, using cloud facilities or containers. The second is to place just the data in the cloud and implement the microkernel as an on-premises system. The third option segregates the core system as on-premises and places the plug-ins in the cloud. While this may seem like a good option from a modularity standpoint, it comes with some challenging implications for responsiveness. Generally, in a microkernel architecture, plug-in calls happen frequently and each call passes a fair amount of information, since teams implement key workflows using plug-ins. The latency incurred by separating the core and plug-ins may lead to undesirable overhead.

Common Risks

The common risks associated with this architecture mainly arise from misapplying it.

Volatile Core

The core in a microkernel architecture is supposed to be as stable as possible after initial development; part of the benefit of this style is that it isolates changes to plug-ins. Building a core that undergoes constant change undermines the philosophy of this architecture, but it's a common mistake. Often this is the result of architects misjudging the core's volatility; they must then address that volatility via refactoring.

Plug-In Dependencies

Microkernels work best when the plug-ins communicate only with the core system, not with each other. Most systems that use plug-ins that are not microkernels use *dependency-free plug-ins*, which are plug-ins that have no dependencies other than the core. In other words, the plug-ins don't communicate with each other and therefore have no shared dependencies that must be resolved by the core. However, complex applications of microkernel architecture, such as the Eclipse IDE, sometimes build dependencies between components, making the core resolve transitive dependency conflicts between them.

While it's thus possible to have dependencies, they create myriad complexities around transitive dependency management. What happens if two plug-ins depend on different versions of the same core library? The core must resolve those dependencies and facilitate communication between the different plug-in versions. Anyone who has struggled with adding plug-ins in an environment where transitive dependencies are in play understands the headache of untangling conflicting versions. It is best to try to avoid dependencies between plug-ins whenever possible.

Governance

Governance in a microkernel architecture revolves around checking how well the architects are honoring its philosophy.

Common governance checks include:

- Volatility checks for the core—which are fitness functions wired into checking churn in version control, rather than a specific code check
- Rate of change in the core
- Contract tests (especially if some plug-ins support different versions than others because of gradual evolution)
- Other structural verifications for the topology

Team Topology Considerations

The obvious split for teams in this architecture is between core and plug-ins, reflecting its topology:

Stream-aligned teams

> The core is an obvious sweet spot for stream-aligned teams, which build the core functionality of the system. Plug-ins may also fall to this team, depending on the type of application.

Enabling teams

The microkernel architecture is extremely well suited for enabling teams, since it segregates some behavior in plug-ins to allow for A/B testing and other experiments.

Complicated-subsystem teams

Microkernels are also well suited for complicated-subsystem teams because they defer specialized behavior to plug-ins. For example, specialized processing like analytics might be isolated in plug-ins, allowing the stream-enabled team to work on core behavior and call out to sophisticated plug-ins for specialized behavior.

Platform teams

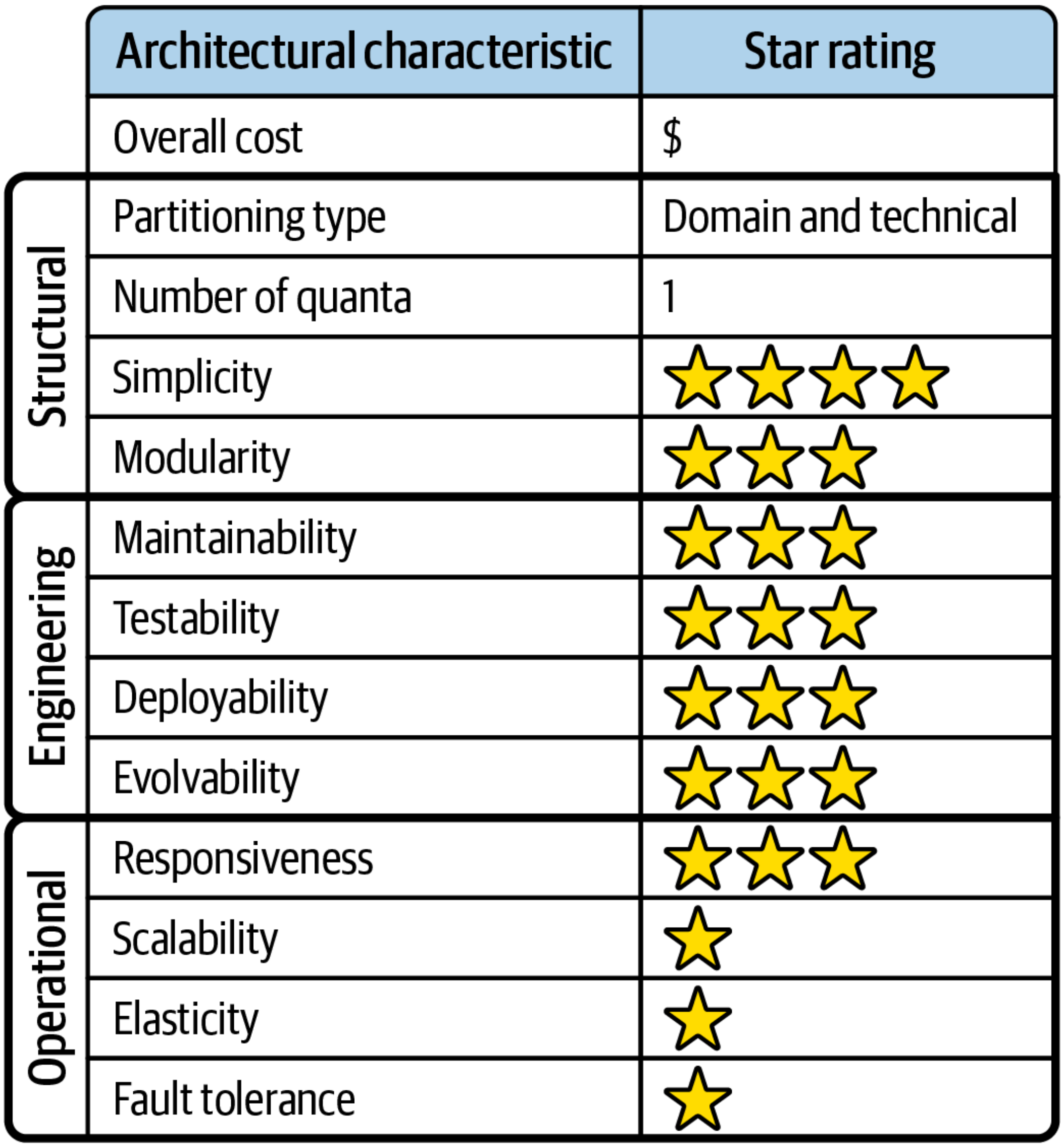
Platform teams mostly concern themselves with the operational details for this architecture, as with other monolithic architectures.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings in Figure 13-9 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 definition for each characteristic identified in the scorecard can be found in Chapter 4.

In the microkernel architecture style, similar to layered, simplicity and overall cost are the main strengths, and scalability, fault tolerance, and elasticity the main weaknesses. These weaknesses are due to the typical monolithic deployments found with the microkernel architecture. Also like the layered architecture style, the architecture quanta is always singular (1) because all requests must go through the core system to get to the independent plug-in components. That's where the similarities end.

Microkernel is unique in that it is the only architecture style that can be both domain partitioned *and* technically partitioned. While most microkernel architectures are technically partitioned, the domain-partitioning aspect comes about mostly through a strong domain-to-architecture isomorphism. For example, problems that require different configurations for each location or client match extremely well with this architecture style. So do products or applications that emphasize user customization and feature extensibility (such as Jira or an IDE like Eclipse).



	Architectural characteristic	Star rating
	Overall cost	$
Structural	Partitioning type	Domain and technical
	Number of quanta	1
	Simplicity	★★★★
	Modularity	★★★
Engineering	Maintainability	★★★
	Testability	★★★
	Deployability	★★★
	Evolvability	★★★
Operational	Responsiveness	★★★
	Scalability	★
	Elasticity	★
	Fault tolerance	★

Figure 13-9. Microkernel architecture characteristics ratings

Testability, deployability, and reliability rate a little above average (three stars), primarily because functionality can be isolated to independent plug-in components. If done right, this reduces the overall testing scope of changes as well as the overall risk of deployment, particularly if plug-in component deployment is runtime based.

Modularity and evolvability also rate a little above average (three stars). With the microkernel architecture style, additional functionality can be added, removed, and changed through independent, self-contained plug-in components, making it relatively easy to extend and enhance applications, and allowing teams to respond to changes much faster. Consider the tax-preparation software example from the previous section. If the US tax law changes (which it does all the time), requiring a new tax form, that new tax form can be created as a plug-in component and added to the application without much effort. Similarly, if a tax form or worksheet is no longer needed, that plug-in can simply be removed from the application.

Responsiveness is always an interesting characteristic to rate with the microkernel architecture style. We gave it three stars (a little above average), mostly because microkernel applications are generally small and don't grow as big as most layered architectures. Also, they don't suffer as much from the Architecture Sinkhole antipattern we discussed in Chapter 10. Finally, microkernel architectures can be streamlined by unplugging unneeded functionality, making the application run faster. A good example of this is WildFly (previously the JBoss

Application Server). Unplugging unnecessary functionality like clustering, caching, and messaging makes the application server perform much faster than it does with these features in place.

Examples and Use Cases

Most tools for developing and releasing software are implemented using the microkernel architecture. Some examples include the Eclipse IDE, PMD, Jira, and Jenkins, to name a few. Internet web browsers, such as Chrome and Firefox, also commonly use the microkernel architecture: viewers and other plug-ins add additional capabilities not otherwise found in the basic browser (the core system). We could point to endless examples of product-based software, but what about large business applications? The microkernel architecture applies to these situations as well.

To illustrate this point, consider our earlier example of tax-preparation software. The US tax agency, the Internal Revenue Service, has a basic two-page tax form called the 1040 form that contains a summary of all the information needed to calculate a person’s tax liability. Each line in the 1040 tax form has a single number, such as gross income, and arriving at each of those numbers requires many other forms and worksheets. Each additional form and worksheet can be implemented as a plug-in component, with the 1040 summary tax form being the core system (the driver). This way, changes to tax law can be isolated to an independent plug-in component, making changes easier and less risky.

Another example of a large, complex business application that can leverage the microkernel architecture is insurance claims processing. Claims processing is a very complicated process. Each jurisdiction has different rules and regulations for what is and isn’t allowed in an insurance claim. For example, some jurisdictions (such as US states) allow insurance companies to provide free windshield replacement if your windshield is damaged by a rock; others do not. This creates an almost infinite set of conditions for a standard claims process.

Most insurance claims applications leverage large, complex rules engines to handle much of this complexity. A *rules engine* is a framework or library that allows developers (or end users) to define a series of rules or steps to define a workflow declaratively, either using visual tools or a domain-specific language. However, these rules engines can grow into the Big Ball of Mud antipattern, where a simple rule change requires an army of analysts, developers, and testers to make sure nothing breaks. Using the microkernel architecture pattern can solve many of these issues.

The claims rules for each jurisdiction can be contained in separate, standalone plug-in components, implemented as source code or as a specific rules-engine instance accessed by the plug-in component. This way, rules can be added, removed, or changed for a particular jurisdiction without affecting any other part of the system. Furthermore, new jurisdictions can be added and removed without affecting other parts of the system. The core system, in this example, would be the standard process for filing and processing a claim—something that doesn’t change often.

The microkernel architecture style is extremely common; once you’ve seen it, you start noticing it everywhere. It’s a case where an architecture structure (core + plug-ins) matches the common domain problem of customization, and it turns out that customization comes up a lot in software.