Microkernel Architecture

March 14, 2022

Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Microkernel Architecture

Software Architecture

March 14, 2022

Richard Thomas

1 Introduction

The microkernel architecture aims to deliver sophisticated software systems while maintaining the quality attributes of simplicity and extensibility. This is achieved by implementing a simple core system that is extended by plug-ins that deliver additional system behaviour. Microkernel is also known as a "plug-in" architecture.

Many common applications use the microkernel architecture. Web browsers, many IDEs (notably Eclipse), Jenkins, and other tools all use this architecture. They deliver their core functionality and provide a plug-in interface that allows it to be extended. Eclipse[1] is famous as a simple text editor that can be extended to be a sophisticated software development tool through its plug-in interface.

Definition 1. Microkernel Architecture

A core system providing interfaces that allow plug-ins to extend its functionality.

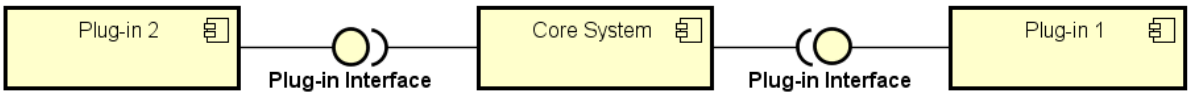

Figure 1: Microkernel architecture – generic structure.

For example, a web browser provides the core behaviour of rendering web pages. Plug-ins extend the browser with specialised behaviour, such as a PDF viewer or dark mode reader.

2 Terminology

The microkernel architecture consists of four elements. The *core system*, *plug-ins* and *plug-in interface* shown in figure 1, plus a *registry*.

Core system implements the system's base functionality.

Plug-ins extend the system by providing independent, specialised behaviour.

Plug-in interface describes how the core system and plug-ins interact.

Registry tracks which plug-ins are available to the core system and how to access them.

The core system implements the minimal functionality that provides the base, or framework, of the system. This may be the core functionality of the system, like the Eclipse text editor or a web browser's page rendering engine. Alternatively, the core system may implement a general processing path, like a payroll system's payment process.

[1] https://www.eclipse.org/downloads/packages/

The payment process may be simply, identify an employee, calculate their fortnightly pay, and send payment to their bank account. Calculating pay can be a very complex process[2]. There are different pay rates, bonuses, incentives, salaried staff, staff paid commission, staff paid hourly rates, overtime rates, penalty rates, deductions, taxes, and many other pay related calculations to consider for each individual.

Plug-ins are independent components that extend the behaviour of the core system. The simple approach is that the system is delivered as a monolith, including the core system and the plug-in. In this case the core system uses the plug-in via a method invocation.

In the payroll example, plug-ins can remove the complexity of different pay adjustment calculations from the core system, by moving each calculation to a separate component. This also improves extensibility, maintainability and testability of the system. New calculations can be added to the system by implementing a new plug-in. As each plug-in is independent of the others, it is easier to implement a new calculation, or changing an existing one, than trying to do so in a large monolith. New plug-ins can be tested independently and then integrated into the system for system testing.

There is usually a single standard interface between the core system and the plug-ins for a domain. The interface defines the methods that the core system can invoke, data passed to the plug-in, and data returned from the plug-in. The plug-in component implements the interface and delegates responsibilities within the component to deliver its behaviour.

Figure 2 is an example of a possible interface for the pay adjusment calculation plug-ins. Each pay adjustment plug-in returns a `MonetaryAmount` indicating the amount by which the employee's base pay is to be adjusted in the pay period. The amount may be positive or negative. The `employee`, `periodDetails`, `employeeConditions` and `basePay` objects are passed as parameters to the plug-in. The plug-in can extract the data it needs from these objects to perform its calculation. The `periodDetails` object provides data about the work performed by the employee during the pay period (e.g. time worked, overtime, higher duties adjustments, ...). The `employeeConditions` set provides data about each of the employee's pay conditions (e.g. before tax deductions, union membership, ...).

```
public interface PayAdjustmentPlugin {
    public MonetaryAmount adjustPay(Employee employee,
                                    WorkDetail periodDetails,
                                    Set<Condition> employeeConditions,
                                    MonetaryAmount basePay);
}
```

Figure 2: Example plug-in interface for payroll system.

The registery records which plug-ins are available to the core system and how they are accessed. For the simple case of plug-ins being used by method invocation, the registry just needs to record the name of the plug-in and a reference to the object that implements the plug-in's interface. For the payroll example, this could be a simple map data structure. The core system could lookup a plug-in by its name and then apply it by invoking the `adjustPay` method on the plug-in object.

3 Microkernel Principles

While the concept of a microkernel architecture is straightforward, there are some principles which should be maintained to produce a maintainable and extendable architecture.

[2] See the Queensland Health payroll disaster https://www.henricodolfing.com/2019/12/project-failure-case-study-queensland-health.html

Definition 2. Independent Plug-in Principle

Plug-ins should be independent, with no dependencies on other plug-ins. The only dependency on the core system is through the plug-in interface.

Plug-ins should be independent of each other. If a plug-in depends on other plug-ins it increases the complexity of the design. This complexity is called *coupling*[3], which is a measure of how dependent different parts of a system are on each other [1]. High coupling (many dependencies) makes it difficult to understand the software, which in turn makes it difficult to modify and test the software. Consequently, if a plug-in depends on other plug-ins it requires understanding the dependencies on the other plug-ins to modify or test the plug-in This can lead to an architecture that resembles a "big ball of mud".

Plug-ins and the core system should be *loosely* coupled. The core system should only depend on the plug-in interface and data returned via that interface, not any implementation details of individual plug-ins. Plug-ins should only depend on the data passed to them via the plug-in interface. Plug-ins should not rely on implementation details of the core system, nor its datastore. If plug-ins and the core system are not isolated from each other by an interface, then any changes to the core system may require changes to some or all of the plug-ins. Similarly, the plug-in interface should mean that any changes to the plug-in will have no impact on the core system.

Definition 3. Standard Interface Principle

There should be a single interface that defines how the core system uses plug-ins.

To provide an extensible design, the core system needs a standard way to use plug-ins. This means that the plug-in interface needs to be the same for all plug-ins. That way the core system can use any plug-in without needing to know details about how the plug-in is implemented. This again is about reducing coupling between the core system and the plug-ins. The standard interface principle means that there is no additional complexity if the core system uses two plug-ins or thousands of plug-ins.

4 Architecture Partitioning

4.1 Technical Partitioning

The notes about layered architecture[4] described the idea of partitioning an architecture into layers, as in figure 3.

This approach to partitioning the architecture is called *technical partitioning*. Each layer represents a different technical capability. The presentation layer deals with user interaction. The business layer deals with the application's core logic. The persistence layer deals with storing and loading data. The database layer deals with file storage and access details.

An advantage of technical partitioning is that it corresponds to how developers view the code. Another advantage is that all related code resides in a single layer of the architecture, making each layer cohesive[5] from a technical perspective. This makes it easier to work with different parts of the code that deal with the same technical capability. The *layer isolation*, *neighbour communication*, *downward dependency*, and *upward notification* principles help reduce coupling between layers.

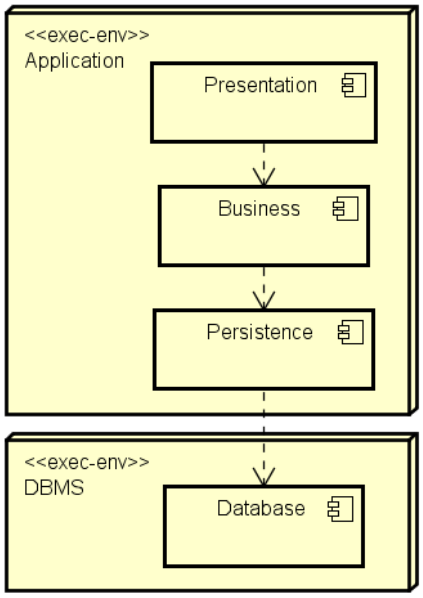

Figure 3: Layered architecture as example of technical partitioning.

[3] https://leanpub.com/isaqbglossary/read#term-coupling
[4] https://csse6400.uqcloud.net/handouts/layered.pdf
[5] https://leanpub.com/isaqbglossary/read#term-cohesion

A disadvantage of technical partitioning is that business processes cut across technical capabilities. Consider when a customer adds a new product to their shopping cart in the Sahara eCommerce example from the architectural views[6] notes. In a technically partitioned architecture, code to implement this is required in all layers of the architecture. If the team is structured with specialist developers who work on different layers of the architecture, they all need to collaborate to implement the behaviour, which adds communication overheads to the project.

Another disadvantage is the potential for higher data coupling, if all domain data is stored in a single database. If it is decided that the system needs to be split into distributed software containers for scalability, the database may need to be split to have separate databases for each distributed container. If the database was originally designed to cater for a monolith architecture, large parts of the database may need to be refactored.

4.2 Domain Partitioning

Domain partitioning is an alternative approach, where the architecture is split into partitions (or major components) corresponding to independent business processes or workflows (the *domains*). The implementation of a domain is responsible for delivering all of that workflow's behaviour. Eric Evans popularised domain partitioning in his book *Domain-Driven Design* [2]. Figure 4 is an example of domain partitioning for an on-line store, similar to the Sahara eCommerce example from the architectural views[7] notes.

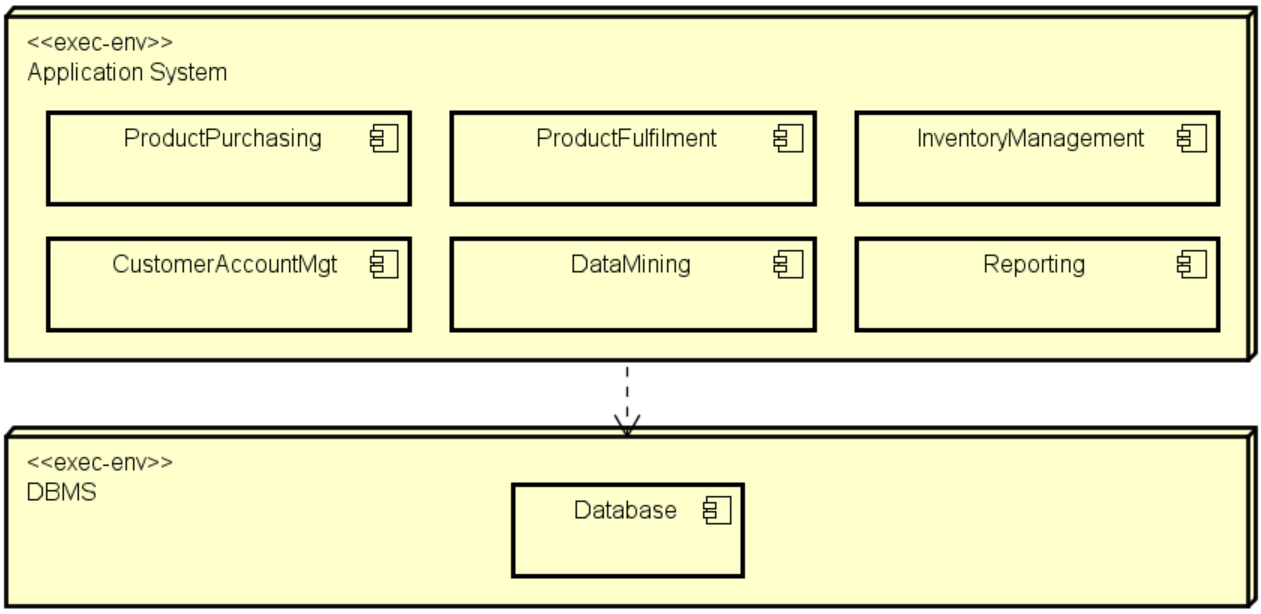

Figure 4: Example of domain partitioning.

An advantage of domain partitioning is that it can model how the business functions, rather than it being structured based on technical decisions. This results in message flows within the software matching how communication happens in the problem domain. This makes it easier to discuss features, and the high-level view of how they are implemented, with business stakeholders who are familiar with how the business operates.

If the domain partitions are independent of each other, it usually makes it easier to split the system into distributed software containers. Even if the system starts with a single database, the independence of domain partitions is likely to lead to a design with lower data coupling, making it easier to split the database.

[6] https://csse6400.uqcloud.net/handouts/views.pdf
[7] https://csse6400.uqcloud.net/handouts/views.pdf

Another advantage of domain partitioning is that it fits well with an agile or continuous delivery process. A single user story will be implemented entirely within one domain. This also makes it easier to implement multiple user stories concurrently, if they are all from different domains.

A disadvantage of domain partitioning is that *boundary* code is distributed throughout all the partitions. This means that any large changes to a particular boundary (e.g. changing the persistence library) will result in changes to all domain partitions. Boundary code is code that interacts with actors or agents outside of the system. The two common technical partitions of the presentation and persistence layers are examples of boundary code. The presentation layer is the boundary between the system and the users. The persistence layer is the boundary between the system and the storage mechanisms. Other boundary partitions may be for message queues or network communication.

It is possible for each domain partition to use technical partitioning within it, to provide some of the lower coupling benefits of a layered architecture for the domain.

4.3 Achitecture Patterns and Partitioning

Different architecture patterns are biased more towards technical or domain partitioning. Those that focus on technical structure are biased towards technical partitioning (e.g. layered architecture). Those that focus on message passing between components are biased towards domain partitioning (e.g. microservices architecture). The core system of the microkernel architecture can follow either technical or domain partitioning.

A consequence of this leads to a variation of principle 3.

Definition 4. Domain Standard Interface Principle

Each domain should have a single interface that defines how the domain uses plug-ins.

Some domains may share a plug-in interface, but allowing domains to have different plug-in interfaces means that each business process can be customised independently by a set of plug-ins.

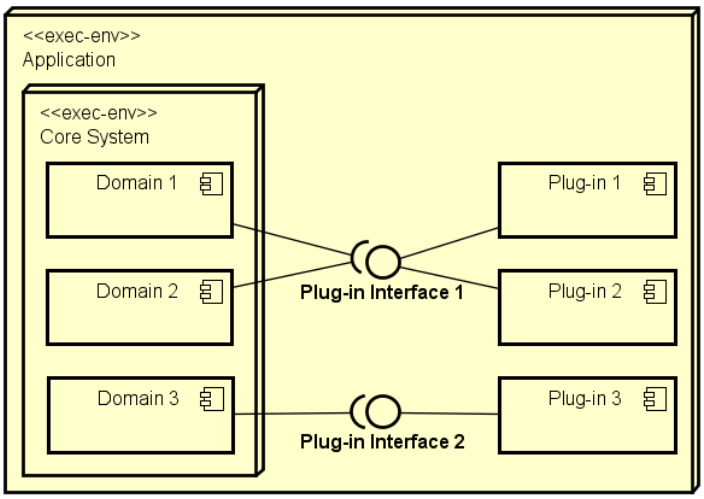

Figure 5: Microkernel architecture with domains.

5 Extensions

While the microkernel architecture often implements the core system as a monolith and it, along with all its plug-ins, is often deployed as a single monolith (e.g. a web browser), that is not the only approach to using the microkernel architecture.

5.1 Distributed Microkernel

The internal partitions of the core system can be distributed to run on different computing infrastructure. Figure 6 is an example of implementing a distributed microkernel architecture using three computing tiers.

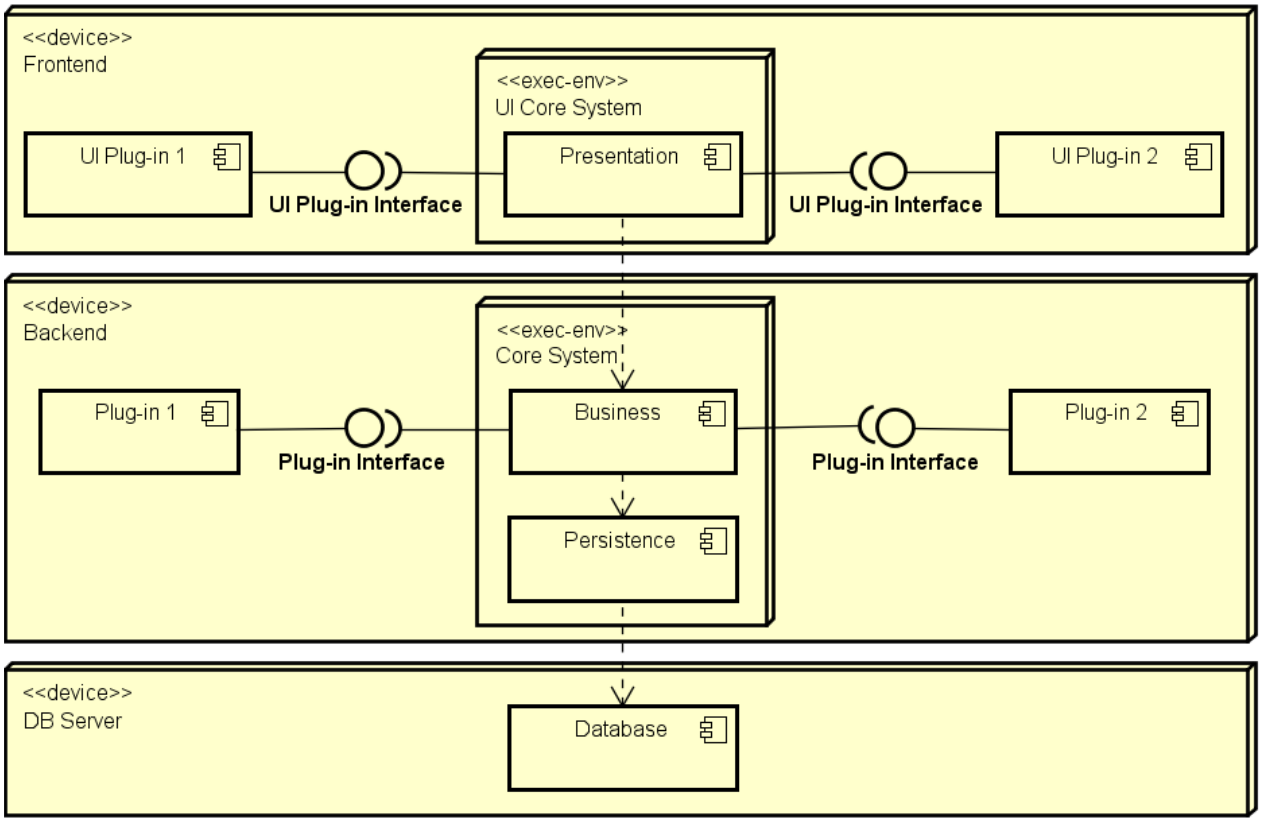

Figure 6: Microkernel architecture distributed by technical partitions.

The presentation layer runs on a frontend such as a web or mobile application. It is possible that the presentation layer could have its own set of plug-ins. The business and persistence logic run on a backend server and have their own set of plug-ins. The database runs on a separate database server, and typically does not have any plug-ins from the application's perspective.

5.2 Alternative Contracts

Figure 2 was an example interface that could be used if plug-ins were invoked directly within the core system. This is possible if the plug-ins are implemented within the application or are loaded as code modules at run time (e.g. via jar or dll files).

There may be times when plug-ins are external services used by the core system. In those situations communication is via some communication protocol. Remote method invocation would still allow the core system to use a compiled interface. Registration of components would become a little more complicated, as the core system would need to a communication interface to be notified of the availability of plug-ins.

More commonly, the communication protocol will be some other mechanism (e.g. REST, as shown in figure 7). In this case, the registry needs to record more information than just the name and interface of a plug-in. The registry will need information about the communication pathway (e.g. URL of the REST endpoint). It will also need details about the data structure passed to the plug-in and returned by the plug-in.

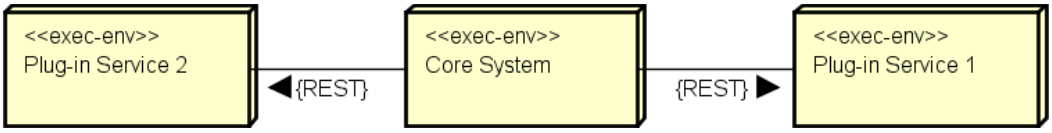

Figure 7: Microkernel architecture – separate plug-in services with REST communication.

5.3 Plug-in Databases

As mentioned in the discussion of principle 2, plug-ins should not use the core system's data directly. The core system should pass to the plug-in any data that it needs. For sophisticated plug-ins, they may need their own persistent storage. They should not request the core system to manage this. Rather, these plug-ins should maintain their own databases, as shown in figure 8.

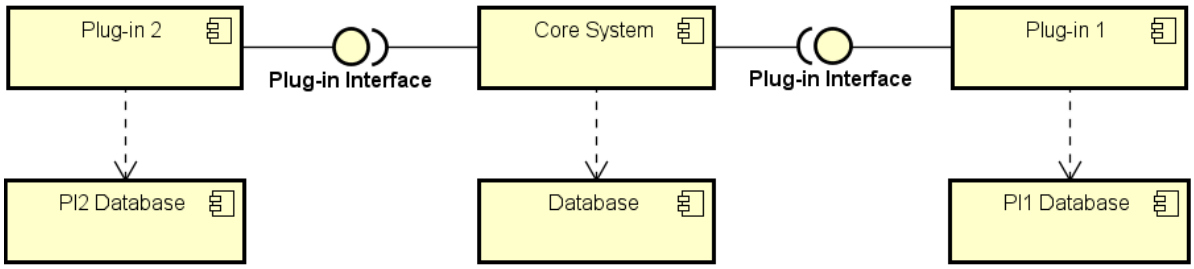

Figure 8: Microkernel architecture with plug-ins maintaining their own databases.

5.4 Non-Standard Interfaces

Principles 3 and 4 indicate that each domain, or the entire core system, should use a single plug-in interface. This is not always possible when the plug-ins are services provided by external agencies. In these situations you would use the adapter design pattern[8] to isolate the external service's interface from the core system. To do this, you would implement a plug-in that is the adapter that then uses the external service. From the core system's perspective, your plug-in is just like any other plug-in.

6 Conclusion

The microkernel architecture is a suitable option for systems were extensibility is a key quality attribute. Following the microkernel design principles will lead to a design where the core system and plug-ins are not affected by changes to each other. Existing plug-ins can be replaced and new plug-ins added with no impact on the core system.

References

[1] G. Starke, U. Becker, C. Lilienthal, M. Mahlberg, S. Kölsch, A. Lorz, A. Rausch, R. Rhoades, S. Fichtner, P. Ghadir, M. Gharbi, M. Bohlen, M. Hillert, P. Hruschka, and W. Fahl, *iSAQB Glossary of Software Architecture Terminology*. International Software Architecture Qualification Board, November 2020. https://leanpub.com/isaqbglossary/read.

[2] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, August 2003.

[8] https://refactoring.guru/design-patterns/adapter