

Structure The Microkernel pattern defines five kinds of participating components:

- *Internal servers*
- *External servers*
- *Adapters*
- *Clients*
- *Microkernel*

The *microkernel* represents the main component of the pattern. It implements central services such as communication facilities or resource handling. Other components build on all or some of these basic services. They do this indirectly by using one or more interfaces that comprise the functionality exposed by the microkernel.

Many system-specific dependencies are encapsulated within the microkernel. For example, most of the hardware-dependent parts are hidden from other participants. Clients of the microkernel only see particular views of the underlying application domain and the platform specifics.

The microkernel is also responsible for maintaining system resources such as processes or files. It controls and coordinates the access to these resources.

In summary, a microkernel implements *atomic* services, which we refer to as *mechanisms*. These mechanisms serve as a fundamental base on which more complex functionality, called *policies*, are constructed.

<p>Class Microkernel</p>	<p>Collaborators</p> <ul style="list-style-type: none"> • Internal Server
<p>Responsibility</p> <ul style="list-style-type: none"> • Provides core mechanisms. • Offers communication facilities. • Encapsulates system dependencies. • Manages and controls resources. 	

➔ In Hydra we want to support UNIX System V and OS/2 Warp, amongst other operating systems. We face a problem when implementing Hydra's process model. A system call such as that to create a new child process is implemented in UNIX by cloning an existing process, copying the whole address space. OS/2 Warp handles process creation totally differently, in that it does not copy the address space of the parent process. In other words, OS/2 Warp and UNIX offer different policies for processes. Hydra is therefore designed to supply basic services such as mechanisms for creating processes as well as mechanisms for cloning existing process spaces. These are combined in various ways for implementing both the process model of UNIX System V and the process model of OS/2 Warp. □

An *internal server*—also known as a *subsystem*—extends the functionality provided by the microkernel. It represents a separate component that offers additional functionality. The microkernel invokes the functionality of internal servers via service requests. Internal servers can therefore encapsulate some dependencies on the underlying hardware or software system. For example, device drivers that support specific graphics cards are good candidates for internal servers.

One of the design goals should be to keep the microkernel as small as possible to reduce memory requirements. Another goal is to provide mechanisms that execute quickly, to reduce service execution time. Additional and more complex services are therefore implemented by internal servers that the microkernel activates or loads only when necessary. You can consider internal servers as extensions of the

<p>Class Internal Server</p>	<p>Collaborators • Microkernel</p>
<p>Responsibility</p> <ul style="list-style-type: none"> • Implements additional services. • Encapsulates some system specifics. 	

microkernel. Note that internal servers are only accessible by the microkernel component.

An *external server*—also known as a *personality*—is a component that uses the microkernel for implementing its own view of the underlying application domain. As already mentioned, a view denotes a layer of abstraction built on top of the atomic services provided by the microkernel. Different external servers implement different policies for specific application domains.

External servers expose their functionality by exporting interfaces in the same way as the microkernel itself does. Each of these external servers runs in a separate process. It receives service requests from client applications using the communication facilities provided by the microkernel, interprets these requests, executes the appropriate services and returns results to its clients. The implementation of services relies on microkernel mechanisms, so external servers need to access the microkernel's programming interfaces.

<p>Class External Server</p>	<p>Collaborators • Microkernel</p>
<p>Responsibility • Provides programming interfaces for its clients.</p>	

➔ In Hydra we want to implement an OS/2 Warp external server and a UNIX System V external server. Both these servers use the mechanisms of the underlying microkernel to implement a complete set of OS/2 Warp and UNIX System V system calls. □

A *client* is an application that is associated with exactly one external server. It only accesses the programming interfaces provided by the external server.

A problem arises if a client needs to access the interfaces of its external server directly. Each client has to use the available communication facilities to interoperate with the external servers.

Every communication with an external server must therefore be hard-coded into the client code. Such a tight coupling between clients and servers, however, leads to various disadvantages:

- Such a system does not support changeability very well.
- If external servers emulate existing application platforms, client applications developed for these platforms will not run without modification.

We therefore introduce interfaces between clients and their external servers to protect clients from direct dependencies. *Adapters*—also known as *emulators*—represent these interfaces between clients and their external servers, and allow clients to access the services of their external server in a portable way. They are part of the client's address space. If the external server implements an existing application platform, the corresponding adapter mimics the programming interfaces of that platform. Clients written for the emulated platform can therefore be compiled and run without modification. Adapters also protect clients from the specific implementation details of the microkernel.