

# **A Distributed Computing Pattern Language**

**Part I:  
Distribution Infrastructure and  
Application Infrastructure Patterns**

**Part II:  
Concurrency Patterns**

**Part III:  
Event Handling Patterns**

**Frank Buschmann**

Siemens AG, Corporate Technology  
Software and Engineering  
`Frank.Buschmann@mchp.siemens.de`

**Kevlin Henney**

Curbralan Ltd.  
`kevin@curbralan.com`

*“With the exception of music, we have been trained  
to think of patterns as fixed affairs. It’s easier and  
lazier that way, but, of course, all nonsense.  
The right way to begin to think of the pattern which connects is to think of a  
dance of interacting parts, pegged down by various sorts of limits.”*

*Gregory Bateson — Cultural Anthropologist*

This paper contains parts of a pattern language for Distributed Computing. Using the Alexandrian form this language weaves together the patterns from the *Pattern-Oriented Software Architecture* series [POSA1] [POSA2], the *Gang-of-Four* [GoF95], and others, regarding this specific topic. The focus of this paper are the language’s distribution infrastructure, application infrastructure, concurrency, and event handling patterns.

## Introduction

A general critique of Gang-of-Four and POSA style pattern forms is their length and depth of detail. In particular it is critiqued that these forms do not highlight the patterns themselves, but their implementation: their timeless core as well as their connection and integration with other patterns likely gets hidden. In addition, the focus on implementation details may get dated as new programming paradigms, languages, and features emerge.

This critique is definitely valid. On the other hand, when coming down to a pattern's implementation in a real-world system, a description of its timeless essence is by far not sufficient: structure and dynamics, implementation guidelines, and concrete examples are needed in addition. The devil is always in the details and thus knowing a particular pattern's essence does not replace solid design and implementation skills.

The only conclusion we can draw from this discussion is that we actually need both types of descriptions. Alexandrian-style pattern language descriptions are useful for getting the big picture of the patterns and their connections as well as for emphasizing the timeless character of these patterns. Gang-of-Four and POSA style pattern descriptions help us in getting the individual patterns to work correctly by using contemporary programming paradigms and languages.

So let's try whether this works. This paper contains parts of a pattern language for Distributed Computing. Using the Alexandrian form this language weaves together the patterns from the *Pattern-Oriented Software Architecture* series [POSA1] [POSA2], the *Gang-of-Four* [GoF95], and other authors, regarding this specific topic. The language is *not* intended to replace the patterns' original descriptions, but to complement them with an additional perspective. For details on their implementation we simply refer to their original source.

The focus of this paper are the language's distribution infrastructure, application infrastructure, concurrency, and event handling patterns.

## Distribution Infrastructure

The explosive growth of the Internet and the World Wide Web in the mid-1990s moved distributed systems beyond their traditional application areas, such as industrial automation and telecommunication. But what are the advantages of distributed systems that make them so interesting? Tanenbaum [Tan92] and others [POSA2] suggest the following:

- *Collaboration and connectivity.* Probably the most important reason for the success of distributed systems is that they enable us to connect to and access vast quantities of geographically distributed information and services. The popularity of instant messaging and 'chat rooms' available on the Internet highlights another common motivation for distributed systems: keeping in touch with family, friends, co-workers, and customers.
- *Economics.* Computer networks that incorporate both workstations and servers offer a better price/performance ratio than mainframe computers. In particular, they yield decentralized and modular applications that can share expensive peripherals, such as high-capacity file servers and high-resolution printers. Similarly, selected application components and services can be delegated to run on hosts with specialized processing attributes, such as high-performance disk controllers, large amounts of memory, or enhanced floating-point performance.
- *Performance and scalability.* According to the Sun Microsystems philosophy 'The network is the computer,' distributed applications are capable of using resources available on a network. A huge increase in performance can be gained by using the combined computing power of several network nodes. In addition—at least in theory—multiprocessors and networks are easily scalable. For example, multiple computation and communication service processing tasks can be run in parallel on different hosts.
- *Inherent distribution.* Some applications are inherently distributed, for example telecommunication management network (TMN) systems, enterprise-wide systems within large organizations, and business-to-business (B2B) systems.
- *Toleration of partial failure.* In most cases, a machine on a network or a CPU in a multiprocessor system can crash without affecting the rest of the system. In particular, various application services can be replicated across multiple hosts. Such a replication minimizes single points of failure, thereby improving the system's reliability in the face of partial failures.

Distributed systems, however, have a significant drawback [Tan92]: 'They need radically different software than do centralized systems.' As components of a distributed system do not share a common address space, all communication between them must be based on mechanisms, policies, and infrastructures that are different to those for inter-component communication in a local application. Likewise, a distributed system must take into account many challenging forces that arise from the network domain that 'sits' between its constituent components, such as latency, jitter, and failure. We can distinguish three levels of *support* for distributed computing: *ad hoc networking*, *structured messaging*, and *middleware*.

At the *ad hoc networking* level reside interprocess communication (IPC) mechanisms, for example, shared memory, pipes, and Sockets [Ste98], that allow distributed components to

connect to one another as well as to exchange data and messages. IPC mechanisms thus address the most fundamental challenge of distributed computing: enabling components from different address spaces to cooperate with one another.

Programming distributed systems only on basis of ad hoc networking support incurs certain drawbacks, however. For instance, using Sockets directly within application code makes this code dependent on the Socket API. Porting this code to another IPC mechanism or re-configuring the original component distribution within the network thus becomes a tedious and error-prone programming job. Even porting this code to another operating system without moving to another IPC mechanism may require code changes, because another platform can offer a different APIs for this mechanism [POSA2]. In addition, programming directly to an IPC mechanism introduces a communication paradigm break. For example, local communication uses object-oriented method invocation, while remote communication uses Socket communication.

Some applications can tolerate these deficiencies of ad hoc networking, in particular systems that run in a distributed, yet homogeneous environment, that are programmed in 'lower-level' languages like C, and whose initial component configuration and choice of IPC mechanism never changes. Certain types of embedded systems fall into this category, for example applications in the automotive domain like engine control, safety feature control, and navigation. Other applications cannot tolerate these deficiencies, however, for example, if they follow a higher-level programming paradigm or if they run in a heterogeneous computing environment.

The next level of support for distributed computing—*structured messaging*—therefore targets to overcome the limitations of *ad hoc networking*. Its objective is to avoid dependencies of application code to low-level IPC mechanisms and to offer higher-level communication mechanisms to distributed systems instead. One of the most prominent representatives for structured messaging support are Remote Procedure Calls [Sun88]. Structured messaging allows distributed components to communicate with one another in almost the same manner as components do in a local environment: they invoke services on each other, pass parameters along with a particular service invocation, and receive results from the services they called, but they do not need to worry about details of specific IPC mechanisms and their platform-specific APIs.

On the other hand, components in a distributed system that use structured messaging to communicate with one another are still aware of the remoteness of their communication partners—sometimes even on their concrete location in the network. Likewise, there is still a paradigm break if the programming paradigm for the application is different from the programming paradigm for the structured messaging facilities, for example, when an object-oriented application uses Remote Procedure Calls.

While structured messaging is sufficient for certain types of distributed systems, for example systems that are implemented in only one programming language and whose component deployment never changes, it still does not resolve all challenges to which many distributed systems are exposed.

These challenges include:

- *Location-independence of components.* Ideally, client components in a distributed system should communicate with remote service components as if they were collocated in the same address space. This requires that both the clients and the service components do not include code that deals with remoting or with location-specifics of other components.

- *Flexible component deployment.* With an evolving network, for example, after upgrading available hardware or after adding new hosts, or in response to evolving usage scenarios, the original deployment of application components across the network could become suboptimal. A re-deployment of a distributed system should therefore be possible, ideally without changes in the system's code and without shutting the entire system down.
- *Integration of legacy code.* Most distributed systems are not developed from scratch. Instead they are constructed from components—or even from entire applications—that originally were not developed for being integrated into a distributed environment. There are several reasons for this: protection of existing investment into software assets, extensive software re-development costs, or the need for a short time-to-market, just to name a few. A specific challenge in this context is the integration of legacy artifacts whose source code cannot be modified for some reason.
- *Heterogeneous components.* With the advent of enterprise application integration (EAI) it also became important to integrate components and applications written in different programming languages into a single, coherent distributed system. Once integrated, these heterogeneous components should perform a common set of tasks in concert.

Mastering these challenges requires more than structured messaging support for distributed systems. Instead it requires dedicated *middleware*, distribution infrastructures that take care of above issues so that the application code of a distributed system can focus on its primary responsibility: implementing the required domain-specific functionality well. Realizing this need has motivated companies like Sun Microsystems and Microsoft and consortia like the Object Management Group (OMG) to develop their own technologies for distributed computing. In spite of their detailed differences all of these technologies share a common architectural vision. This vision is reflected in the entry point of our distributed computing pattern language:

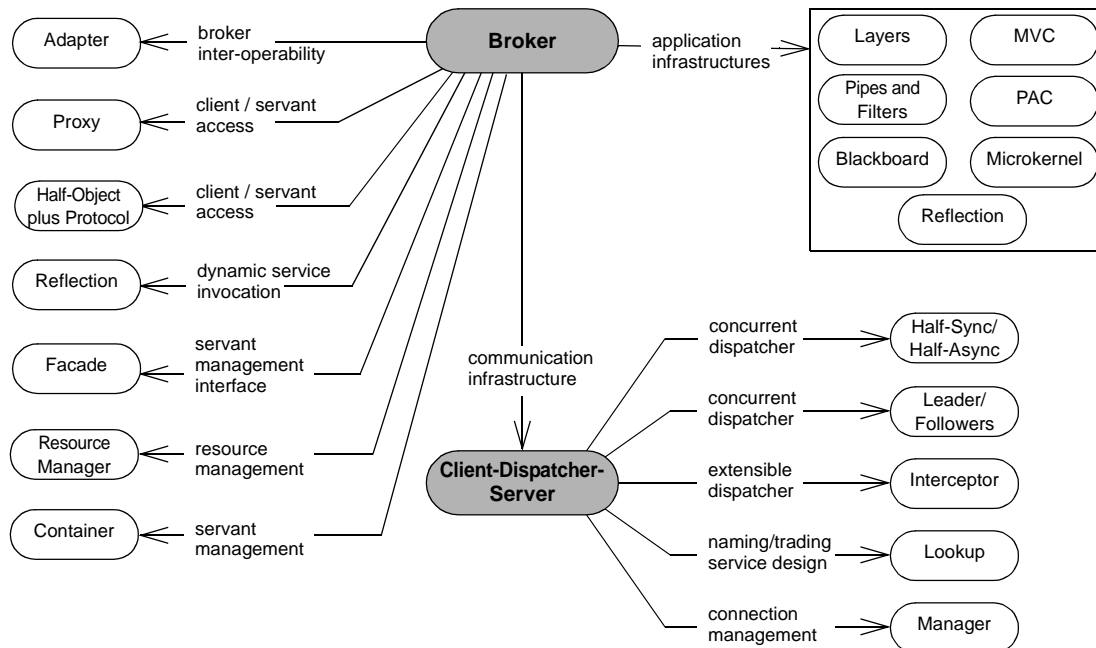
The *Broker* architectural pattern (7) [POSA1] helps to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

The vision and structure of *Broker* is realized with help of several other architectural and design patterns. One of these design patterns we also consider as a distribution infrastructure pattern:

The *Client-Dispatcher-Server* design pattern (10) [POSA1] introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.

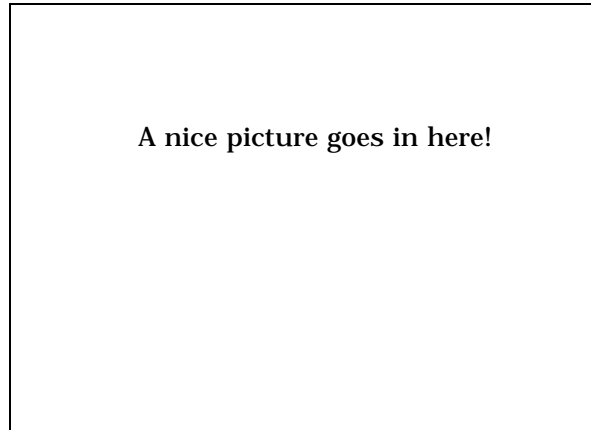
In contrast to our original classification in *A Systems of Patterns* [POSA1], our pattern language does not consider the *Pipes and Filters* (19) and *Microkernel* (33) patterns to be distribution infrastructures. Instead, we view them as patterns that help to *structure* distributed applications. While *Pipes and Filters* and *Microkernel* can *benefit* from a *Broker*-based distribution infrastructure, they do not by themselves introduce such an infrastructure. Likewise, we are aware that our distribution infrastructure patterns are not 'complete.' For example, peer to peer computing requires more than just a *Broker*-based middleware. Yet we hope that we or other members of the software and patterns community can fill these holes over time, when the missing patterns get discovered and documented.

Both patterns are integrated into our pattern language as follows:



It is important to note that no *Broker*-based middleware can resolve all challenges of distributed computing. A *Broker* is no more than a messenger between the application components of a distributed system. Thus, these components must be prepared for handling certain networking challenges by themselves. For example, in case of network failures or server crashes, a distributed application must not respond with total failure. Likewise, an application component that invokes services on a remote component must take into account the latency and jitter that a network adds to all remote communication. Finally, no distribution infrastructure can cure problems that result from a suboptimal deployment of application components across a network or from an inappropriate organization of their cooperation. The particular deployment of application components of a distributed system as well as the way these application components handle network failure, latency, and jitter thus has a significant impact on this system's stability, performance, and scalability. In other words, middleware like a *Broker* is an important part of every distributed system, but it cannot handle responsibilities that are application-specific and thus beyond its scope. Application components of a distributed system must be specified thoughtfully—always having the network in mind—even if a *Broker* allows them to be independent of the concrete location of other application components.

## Broker \*\*



We are designing a distributed software system.

\*\*\*

**A distributed system faces many challenges that do not arise for single-process software. Some challenges arise because communication between components in a distributed system is more diverse, ranging from local method calls to remote calls over the network. Other challenges arise from the benefits that distribution promises: the integration of heterogeneous service components into coherent applications that use a network's computing resources most optimally. However, if developers of a distributed system must master all these challenges within their application code, they likely will lose their primary focus: to develop service components that resolve their domain-specific responsibilities well and to compose entire applications from these components.**

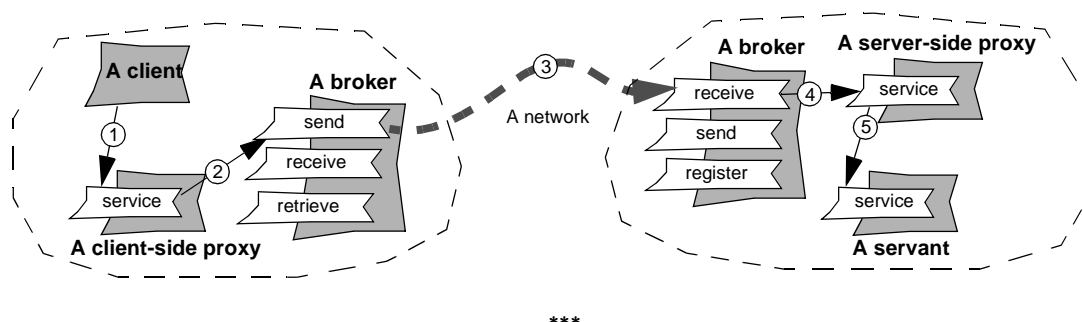
In addition, the application code will become dependent on particular solutions and implementations for these challenges. Consequently, re-deploying the system's components or porting them to another distribution environment becomes both time consuming and cost intensive. Reusing existing service components in other distributed systems can also require extensive code adaptation, even if their domain-specific functionality could be reused as is.

To overcome these effects of 'ad hoc distribution,' application components should be 'protected' from as many distribution issues as possible. In particular, client components in a distributed system should have location-independent access to services provided by other components. This independence should extend to a client's ability to locate these services and to a uniform mechanism for invoking local or remote services. Likewise, there should be no difference between defining service components that are accessed locally or remotely. Clients should also be able to use services of components written in other programming languages, as it should be possible to integrate existing legacy components and services into a distributed system. Last but not least, the deployment of components across a network should not be hard-coded, so that it can be tuned towards using the available computing power most optimally.

Therefore:

**Separate the communication infrastructure of a distributed system from its application functionality via a federation of brokers that hides and mediates all communication between the components of the system. There is at least one broker instance per participating network node. Register service components—the servants—that are available on the network nodes—the servers—dynamically with their local broker to avoid hard-coding a specific servant deployment. Servants become accessible within the system once the brokers are aware of their interfaces and location. To decouple brokers from servants and to allow legacy servants to be integrated into a distributed system, introduce a server-side proxy for each servant that maps between the broker interface and the servant interface. To allow clients to invoke services on the servants as if they were collocated in the same address space and were implemented in the same programming language, provide appropriate client-side proxies for all servants. Let clients retrieve these client-side proxies from their local broker.**

**A client initiates a service request by calling through the relevant client-side proxy. In collaboration with the client-side and server-side brokers, as well as the servant's server-side proxy, the client-side proxy can deliver this request to the servant and receive any results.**



Brokers enable the exchange of requests and responses, as well as the delivery of messages and notifications between the components of a distributed system. The core design of a broker is based on a *Client-Dispatcher-Server* (10) configuration that provides and encapsulates the functionality to register servants with the broker, to manage connections between clients and servants, and to perform IPC. Such a design *enables* location-independence: application components can use higher-level APIs instead of specific communication mechanisms and their platform-dependent implementations. The servant registration functionality of the broker also allows servants to be configured dynamically at run-time of a distributed system. This supports a flexible servant deployment across a network and its computing resources. Different broker implementations can differ in their concrete APIs. Thus, introduce *Adapters* [GoF95] to allow these broker implementations to communicate with one another.

Servants must be provided along with appropriate client-side and server-side *Remote Proxies* [POSA1], which are often generated statically from a specification of the servants' interfaces within an Interface Definition Language (IDL). Client-side proxies allow clients to access servants as if they were local objects: they offer the same interface as their associated servants and

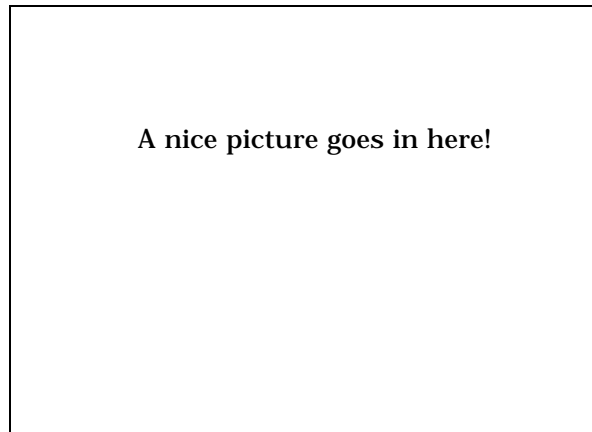


perform all (remote) communication with them. Client-side proxies are also implemented in the same programming language as the clients, even if ‘their’ servants are implemented in a different programming language. When a client instructs the broker to connect to a particular servant, the broker returns an appropriate client-side proxy once the connection is established. *Lookup Client-side Proxies* allow to encapsulate this functionality, thus enabling clients to be completely location-independent of the servants that they access. To assist clients in accessing servants whose interfaces they do not know, provide the broker with a *Reflection* (30) API. Individual service requests or even whole proxies can be assembled dynamically through this API. Server-side proxies allow brokers and servant implementations to be independent of one another: the proxies perform all mapping and dispatching from the broker interface to the servant interface. A dedicated servant interface is not necessary to make the servant accessible remotely via a broker. Thus, server-side proxies also allow to integrate (legacy) servants whose code cannot be modified. Together a client-side proxy and its corresponding server-side proxy form a *Half-Object plus Protocol* [PLoPD1] that allows both a client and a servant to cooperate with one another in a location-independent way.

The resources that a server can offer to its servants are limited— connections, file handles, memory, processes, threads. If servants keep these resources ‘at their own leisure,’ the server can saturate, degrading the system’s quality of service. A *Resource Manager* (*not yet documented*) centralizes a server’s policies for resource usage. Let server-side proxies instruct the *Resource Manager* to acquire all the necessary resources to execute requests before dispatching the corresponding servant methods. A *Facade* [GoF95] presents a simple and consistent interface for servants to access the server-side broker and the *Resource Manager*. This triad is often referred to as a *Container* [VSW02].

Several application infrastructure patterns help to partition the functionality of a distributed system: *Layers* (17) support building n-tier systems; *Pipes and Filters* (19) coordinate the processing of data streams; a *Blackboard* (21) supports heuristic computation; *Model-View-Controller* (24) and *Presentation-Abstraction-Control* (27) configurations structure interactive applications; and a *Microkernel* (33) or *Reflection* (30) underpin the architecture of systems that evolve over time. Typically, the most effective application infrastructure for a distributed system is a combination of several of the above fundamental architectures. For example, the model of a *Model-View-Controller* collaboration may expose a distributed *Layers* design with some layers having *Reflection* capabilities.

## Client-Dispatcher-Server \*\*



We are specifying the communication infrastructure of a *Broker* (7) architecture.

\*\*\*

**Components in a distributed system that want to cooperate with one another must first establish appropriate connections via which they can exchange requests, messages, and data. These connections must also be managed efficiently while the components collaborate. However, a primary requirement for many distributed systems is location-independence: application components should be able to cooperate as if they were collocated in the same address space.**

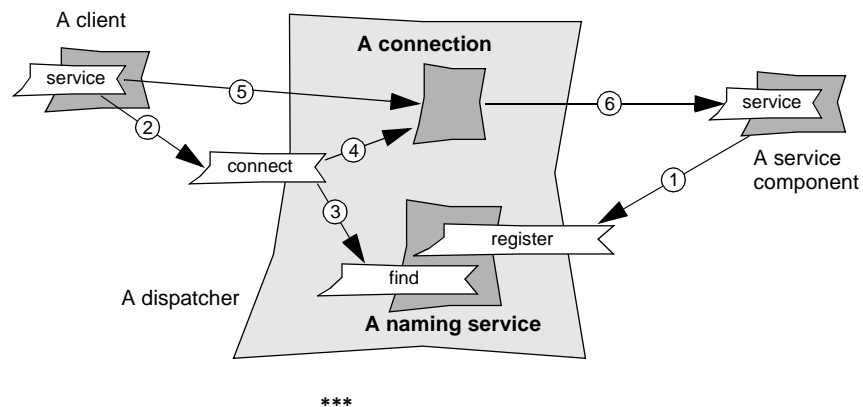
This requires to shield application components from the networking domain of a distributed system as much as possible: from locating remote components and connecting to them, from communicating between distributed components via a particular IPC mechanism, and from maintaining established connections and handling certain types of communication and network errors.

As this 'networking' functionality is central for every distributed system, it must perform with sufficient efficiency. Yet it should not hard-code a particular application component configuration. Instead, it should be possible to deploy application components across a network flexibly and dynamically.

Therefore:

**Encapsulate all IPC and connection management services of a distributed system within a specialized dispatcher that performs these services on behalf of application components. In addition, extend the dispatcher with a naming and/or trading service to avoid hard-coding a specific component deployment. Service components register their name, properties, and location with the dispatcher; clients access its naming/trading service to connect to a particular service component by using this component's name or a set of properties that they expect from it.**

**When the dispatcher receives a connection request from a client, it uses the name or properties passed along with this request to identify the associated service component and its physical location. Then the dispatcher establishes a connection between the client and the service component. Once this connection is established, client and service component can communicate with one another. The dispatcher is not involved in this communication but manages the connection transparently for both the client and the service component.**



The dispatcher, together with its naming and/or trading service, *enables* location-independence of application components within a distributed system: it is the responsibility of the dispatcher to locate service components on behalf of clients and to maintain the connections between them; naming and/or trading services support a flexible deployment of application components across a network. In addition, low-level IPC mechanisms are encapsulated by the dispatcher and the connections—application components are thus both independent and unaware of them. Yet be aware that the dispatcher is only the basis for location-independence: clients must still connect explicitly to a remote service component via the dispatcher.

As the dispatcher represents a central communication hub within a distributed system, its implementation must be efficient. Otherwise it can become a performance bottleneck that degrades the system's general performance. Most dispatcher implementations are therefore based on a suitable concurrency model, most likely on a *Half-Sync/Half-Async* (39) or a *Leader/Followers* (41) architecture. A concurrent dispatcher can establish and manage multiple connections simultaneously. In addition, it can leverage available hardware-parallelism transparently.

Different types of distributed systems may require different properties from the dispatcher's networking functionality. For example, some systems demand secure communication, others load balancing in order to improve throughput. Therefore, design the dispatcher as an *Interceptor* [POSA2]. This keeps its core lean, but extensible with additional 'out-of-band' services that are not needed by all distributed systems. The dispatcher's naming and/or trading service is typically based on a *Lookup* [JK00] mechanism; its connection management functionality on a *Manager* [PLoPD3].

## Application Infrastructure

Once we selected the appropriate distribution infrastructure for a system, the next major technical task is to establish its base-line architecture. In particular, we must complement the distribution infrastructure with a component and subsystem decomposition that expresses the system's functionality. Typically, we are aware of a whole slew of different aspects that we must consider in this respect, but have problems in organizing this mess into a workable structure. Brian Foote and Joseph Yoder call this situation a 'big ball of mud' [FoYo99]. This 'ball of mud' is usually all we have in the beginning, and we must transform it into a more organized structure.

Cutting the ball along lines visible in the application domain will not always help. On the one hand, the resulting software system needs to include many components and exhibit many properties that are unrelated to this domain. For instance, quality of service requirements are not modular and therefore cannot be addressed through component decomposition alone. On the other hand, we—as developers—want more than a system that simply meets the visible user requirements. The user's indifference to developmental qualities such as portability, maintainability, comprehensibility, extensibility, adaptability, and so forth should not be shared by the developer.

Finding a suitable application architecture therefore depends on framing answers to a few key questions and challenges:

- *How is application processing organized?* Some applications receive service requests from clients on which they react and respond. Other applications process streams of data. Yet other applications perform self-contained tasks without receiving stimuli from their environment. For some applications it may even be impossible to identify any concrete workflow and explicit cooperation amongst its components.
- *How does the application interact with its environment?* Some systems interact with different types of human user, others with other systems as peers, and others are embedded within even more complex systems. Inevitably, there are systems that have all of these interactions.
- *What is the life expectancy of the application?* Some systems are short-lived and thrown away when they are no longer used. Other systems will be in operation for 30 years or more and must respond to changing requirements, environment, and configurations.

Our distributed computing pattern language includes seven strategic patterns that help to break apart the 'big ball of mud.' Each pattern has its own answers to the questions raised above:

The *Layers* architectural pattern (17) [POSA1] helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

The *Pipes and Filters* architectural pattern (19) [POSA1] provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

The *Blackboard* architectural pattern (21) [POSA1] is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized

subsystems assemble their knowledge to build a possibly partial or approximate solution.

The *Model-View-Controller* architectural pattern (MVC) (24) [POSA1] divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

The *Presentation-Abstraction-Control* architectural pattern (PAC) (27) [POSA1] defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

The *Reflection* architectural pattern (30) [POSA1] provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

The *Microkernel* architectural pattern (33) [POSA1] applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

The seven application infrastructure patterns comprise all the patterns from *A Systems of Patterns* [POSA1] that we classified as 'from mud to structure,' user interaction, and system adaptation patterns. This re-categorization does not invalidate their original classification, but within our pattern language it does allow us to emphasize clearly the distinction between distribution infrastructure and application infrastructure.

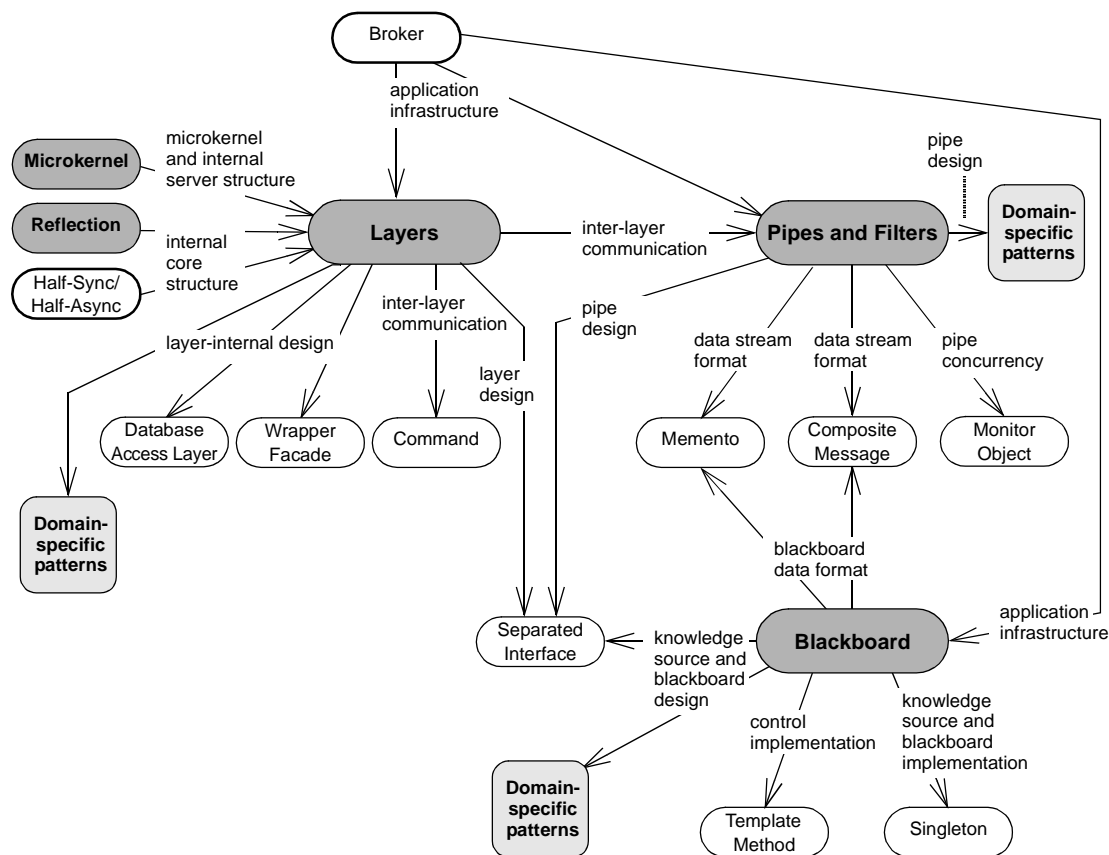
Note, however, that most real-world software systems cannot be formed reasonably from a lone strategic pattern. Different patterns offer different architectural properties; a system may find that it must grow from multiple strategic patterns to meet its system requirements. For example, you may have to build a system that has two distinct, and sometimes conflicting, design goals: adaptability of its user interface *and* portability to multiple platforms. For such systems you must combine several patterns to form an appropriate structure. Scaling to a distributed environment, these application infrastructure patterns must also be integrated with suitable distribution patterns.

However, the selection of a strategic pattern—or a combination of several—is but the first step of many when designing a software system. It is *not* a complete software architecture. Instead it

remains a structural framework for a software system that must be further specified and refined. This includes the task of expressing the application's concrete functionality within the framework, detailing its components and relationships.

In our distributed computing pattern language we support this process of refining and detailing such base-line architectures with the help of more tactical patterns.

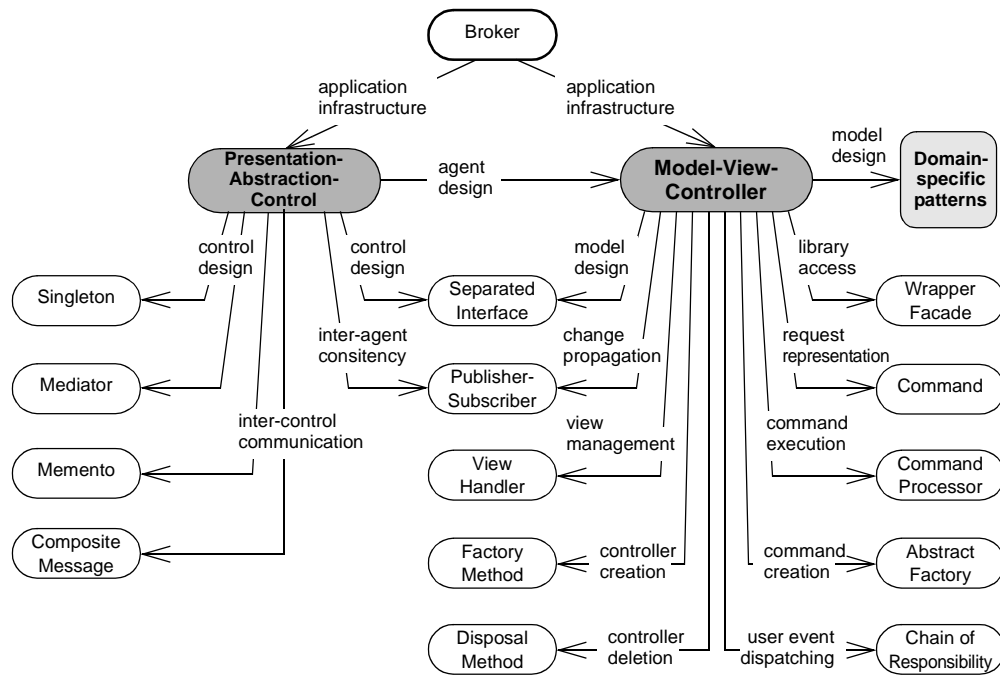
The diagram below illustrates what other patterns of our language help with refining the *Layers* (17), *Pipes and Filters* (19), and *Blackboard* (21) application infrastructures. All three patterns specify basic decomposition strategies for software systems.



Note that in this diagram we can only include those patterns that help to implement key *structural properties* of the respective application infrastructures. We cannot provide hints about which patterns and pattern languages support the decomposition of the application functionality that they 'host.' The reason for this is obvious. What functional design is most appropriate for a given system strongly depends on its concrete responsibilities as well as on its

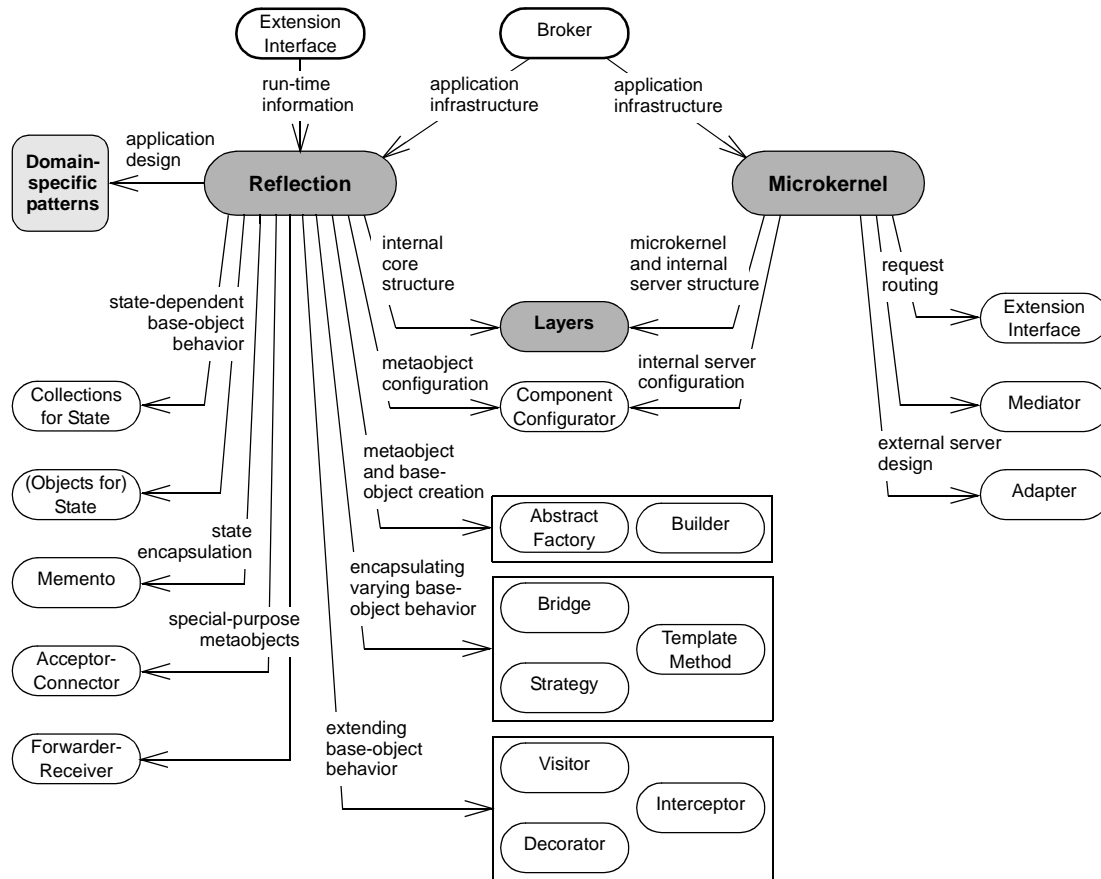
particular functional requirements. At this point, only domain-specific patterns and pattern languages can help. We indicate this fact with a special symbol in the diagram.

The second diagram presents how the *Model-View-Controller* (24) (MVC) and *Presentation-Abstraction-Control* (27) (PAC) architectures can be refined with other patterns of our language. Both MVC and PAC specify application infrastructures for interactive software.



Note that in this diagram the *Presentation-Abstraction-Control* pattern does not need to reference domain-specific patterns and pattern languages directly. The relationship is transitive, because the fundamental design of a PAC architecture is based on the *Model-View-Controller* pattern, which already references domain-specific patterns and pattern languages to design its model component.

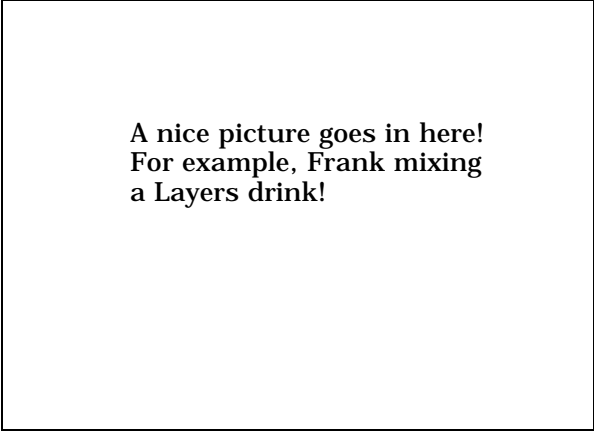
The final diagram depicts the integration of the *Reflection* (30) and *Microkernel* (33) application infrastructures into our pattern language. Both patterns address software evolution.



Analogously to the previous diagram, the *Microkernel* pattern does not need to reference domain-specific patterns and pattern languages directly, because of its transitive relationship to these types of pattern via the *Layers* application infrastructure. In contrast, the *Reflection* pattern references domain-specific patterns and pattern languages—even though it references *Layers* as well. The reason for this is that designing a high-quality *Reflection* architecture requires to use domain-specific pattern and pattern-languages prior to applying the *Layers* pattern.



## Layers \*\*



A nice picture goes in here!  
For example, Frank mixing  
a Layers drink!

We are partitioning an application's functionality. Or we are refining the clients and servers in a *Broker* (7) architecture. Or we are specifying a *Reflection* (30) architecture, or a *Microkernel* (33), or a *Half-Sync/Half-Async* (39) concurrency model.

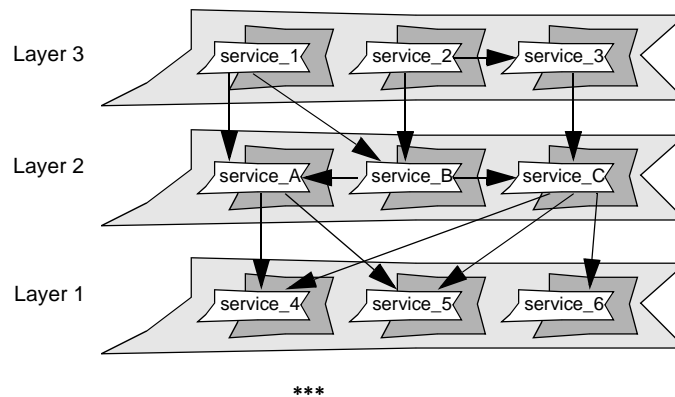
\*\*\*

**Some applications consist of services that reside at different abstraction levels, with the higher-level services building upon the lower-level services. However, despite their cooperation relationships and dependencies, it is often desirable to decouple the services of different abstraction levels as much as possible.**

For instance, services may be reusable across applications, which would be next to impossible if they were tightly coupled with an application-specific context. Prime examples for such reusable services are object-oriented abstractions atop the network or operating system, or common infrastructural services like naming and load balancing. Even services specific to a particular domain can likely be reused within other applications for the same domain. Yet not all services are reusable in isolation. Sets of related services strongly complement each other, therefore they are reusable only wholesale, for instance, services that shield an object-oriented application from the specifics of a relational database. Services at a particular level of abstraction may also evolve, for example, because better algorithms are discovered or a system is ported to a new platform. Such changes should not ripple through the entire application and all its abstraction levels.

Therefore:

**Decouple the abstraction levels of an application by splitting it into several loosely connected layers—one layer for each abstraction level. Let every layer comprise exactly those application services that reside at the abstraction-level which it represents. To keep inter-layer dependencies as loose as possible, let services of a particular layer only use services offered by the same or lower layers. To keep layers independent of implementation details of other layers, provide every layer with an interface that is separate from the implementation of its services and program only towards these interfaces.**



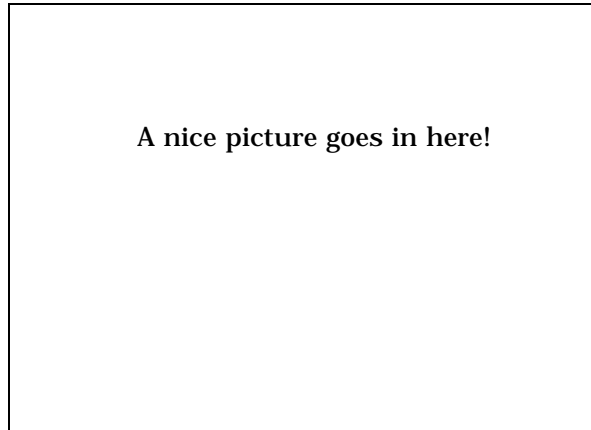
In a layered system, a particular layer uses and combines the services offered by lower layers to higher-level services. The lowest layer typically provides services that reside atop hardware devices, their interfaces, or the operating system. With rising level of abstraction, layers and their constituent services become more domain-specific. The topmost layer usually implements an application's user interface. Provide a *Separated Interface (not yet documented)* for all services that a layer publishes to other layers and—within every layer—program only towards these interfaces.

This particular combination of layering and separating service interfaces from their implementations achieves the desired decoupling within an application. Inter-layer dependencies are minimized as every layer only depends on service interfaces of other layers. Thus, it is possible to modify a particular layer implementation without affects onto other layers as well as to reuse a particular layer implementation wholesale within a different application. Both changeability and reusability properties also hold for individual services within each layer. In addition, *Separated Interfaces* for every published service enable remote access to layers as well as the implementation of concurrent services.

The internal design of every layer strongly depends on the type and complexity of the services that it offers. Appropriate domain-specific patterns and pattern languages can help with an effective layer decomposition. A layer that resides directly atop the operating system or a platform-specific library typically consists of a set of *Wrapper Facades* [POSA2] that shield the higher layers from the details of these platforms. A layer that decouples an application from a database is often specified as a *Database Access Layer* [PLoPD3].

In most layered systems, control flow is strictly top-down. However, in some layered applications, for example in protocol stacks like TCP/IP or UDP, it may be necessary that a service from a lower level must call services from a higher level. In this case do not call the higher level services directly. Use *Commands* [GoF95], messages, or events instead. Such a design preserves the rule that a lower layer must not call services of a higher layer, which would make it impossible to reuse the lower-level layers independently. An even stronger decoupling of layers can be achieved by introducing a *Pipes and Filters* (19) communication between services. In this design, layers and their services correspond to filters that exchange messages and data through shared pipes—top-down as well as bottom-up. No service depends on any interface of any other service. Services—as well as their respective layers—become completely independent of one another.

## Pipes and Filters \*\*



We are structuring an application that processes streams of data. Or we are specifying the clients and servers in a *Broker* (7) architecture. Or we are designing a message-based communication mechanism for a *Layers* (17) architecture.

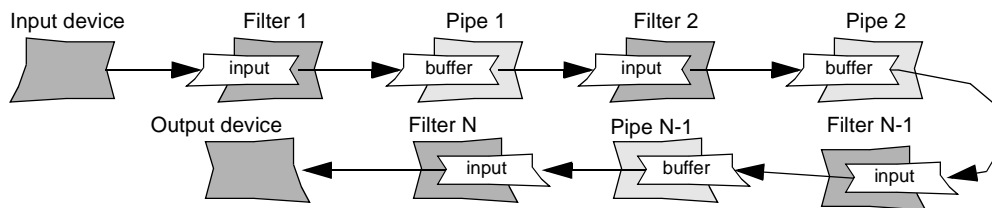
\*\*\*

**Some applications process streams of data: input data streams are transformed step-wise into output data streams. However, structuring such an application using common and familiar (object-oriented) request/response semantics is infeasible. Instead we must specify an appropriate data flow model.**

To increase the throughput of the application, this data flow model should allow to read, process, and write data streams incrementally, rather than wholesale. In addition, long-duration processing activities must not become a performance bottleneck. If possible, data should be processed concurrently.

Therefore:

**Divide the application's task into several self-contained processing steps and connect these steps to a data processing pipeline. Implement each processing step as a separate filter component that consumes and delivers data incrementally. This maximizes each filter's individual throughput. Chain the filters such that they model the application's main data flow. In this pipeline, data that is produced by one filter is consumed by its subsequent filters. Decouple adjacent filters via pipes that buffer the data to be exchanged between the filters. To increase the throughput of the entire processing pipeline, design this decoupling such that filters can execute concurrently. Optimize the pipeline's throughput by inserting multiple instances for those filters that perform long-duration tasks.**

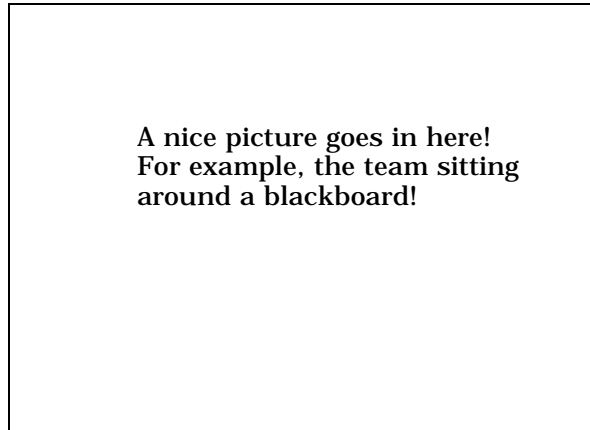


\*\*\*

Filters are the units of computation within a *Pipes and Filters* chain. Each filter represents a self-contained data processing step. As a filter's internal design depends on its particular responsibility, consider to structure it with the help of appropriate domain-specific patterns and pattern languages. Provide each filter with a *Separated Interface (not yet documented)* to receive and deliver data. This separation of interface from implementation supports to access filters remotely and to exchange their implementations transparently for other filters. Enable concurrent and incremental data processing by running each filter within its own thread of control. If a filter performs a long-duration activity, consider to integrate more than one instance of this filter into the processing pipeline. Such a design can increase system throughput: some filter instances can start processing new data streams while others still process previous data streams.

Pipes are the units of data exchange and coordination within a *Pipes and Filters* architecture: each implements a policy for buffering and passing data along the filter chain. There is one pipe per group of adjacent filters. Data producing filters write data into the pipe, data consuming filters read this data from it. Design pipes as *Monitor Object* (47) data queues to allow a *Pipes and Filters* system to self-coordinate the execution sequence of its concurrent filters [POSA2]. The data to be passed along a pipe is application-specific and can be encapsulated inside *Composite Messages* [PLoPD2] and *Mementos* [GoF95].

## Blackboard



We must resolve a task for which no feasible deterministic solution strategy is known. Or we are specifying the clients and servers in a *Broker* (7) architecture.

\*\*\*

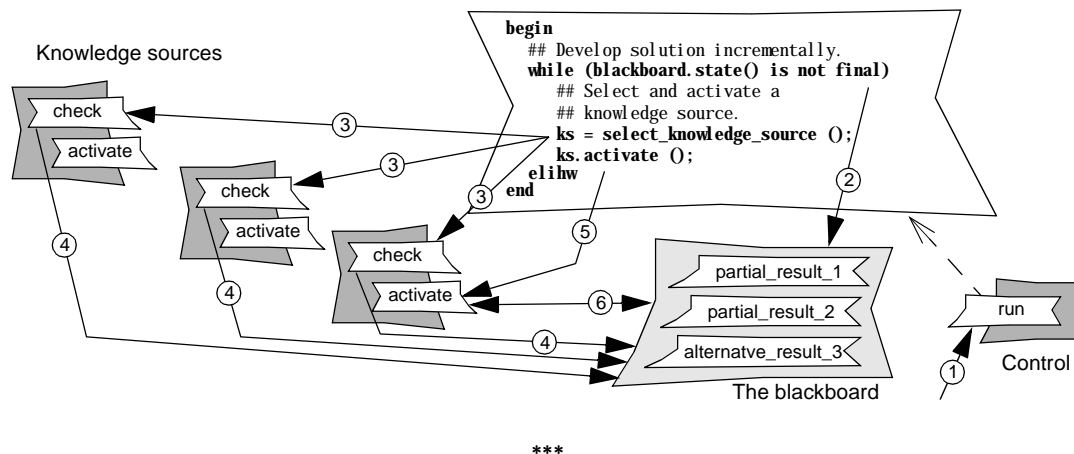
**For some tasks it is hard to specify a feasible deterministic solution strategy, as only approximate or uncertain knowledge about them is available. Examples include speech recognition, submarine detection based on sonar signals, and the inference of protein molecule structures from X-ray data. Yet it is necessary to develop productive applications for these types of task.**

This requires to resolve several true challenges: input data is often fuzzy or inaccurate, the path towards a solution must be explored, every processing step may generate alternative results, and there is often no optimal solution known. Yet it is important to compute valuable solutions in a reasonable amount of time.

Therefore:

**Use heuristic computation and resolve the task via a step-wise improvement of intermediate solution hypothesis. Divide the overall task of the system into a set of smaller, self-contained subtasks for which deterministic solution algorithms are known and assign the responsibility for each subtask to an independent program, a knowledge source. To allow the chosen heuristics to execute knowledge sources in arbitrary order, let them cooperate via a non-deterministic data-driven approach. Using a shared data repository, the blackboard, knowledge sources can evaluate whether there is input data available for them, process this input, and deliver back their results—which may then form the input for arbitrary other knowledge sources. Coordinate the computation with help of a control component that uses an opportunistic heuristics to select and activate appropriate knowledge sources if the data on the blackboard does not yet represent a useful final result, and finishes the computation if it does. Such a strategy works towards**

**a solution via incremental improvement of partial results and evaluation of alternative hypothesis instead of using a deterministic solution algorithm.**



To implement a *Blackboard* system, first decompose the task that it should resolve: what is the kind of input that the system receives, what the kind of output that it is expected to produce, what are potential solution paths, what are useful (types of) partial results on these paths, what are the application services that can contribute to the solution, what input or partial results can each service process, and what partial or final results can each service deliver.

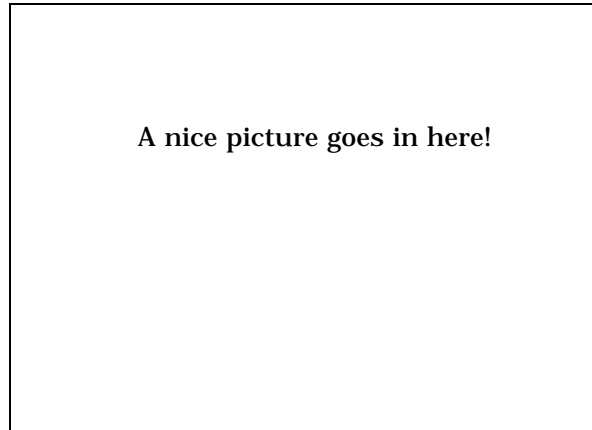
On basis of this analysis, define self-contained and independently executable knowledge sources for every service that is involved in the tasks's solution. Such independence allows to arrange knowledge sources to *arbitrary* execution orders—a necessary pre-condition for a heuristic solution strategy. To allow this heuristic solution strategy to determine a *particular* execution order, split each knowledge source into two separate parts. A condition part only examines if the knowledge source can make a contribution to the computation's progress—by inspecting the data written on the blackboard. An action part implements the knowledge source's functionality: it reads one or more input data from the blackboard, processes it, and writes one or more output data back onto the blackboard. Alternatively, the action part could erase data from the blackboard, because it identifies this data as not contributing to the overall task's solution.

The internal structure of a knowledge source depends on the type and complexity of its responsibility. Domain-specific patterns and pattern languages can help with an effective knowledge source decomposition. A *Separated Interface* (not yet documented) for every knowledge source helps to exchange their implementations transparently, for example, when better solution strategies are discovered. They also allow to access knowledge sources remotely.

Implement the blackboard as a *Singleton* [GoF95] data repository that maintains all partial and final results that the knowledge sources produce. The formats of these data are application-specific, often even knowledge-source-specific. *Composite Messages* [PloPD2] and *Mementos* [GoF95] help to encapsulate this data. A *Separated Interface* provides a remote and thread-safe access to the blackboard.

A *Singleton* [GoF95] control component realizes the heuristic solution strategy of a *Blackboard* system. First it reads the system's input and stores it on the blackboard. Then it enters a loop that executes three steps. The initial step calls the condition parts of all knowledge sources to determine if they *can contribute* in the current state of computation. The second step uses a heuristic that analyzes the results returned by the condition parts to determine the particular knowledge source that can *best contribute* to the progress of the computation. The last step invokes the action part of the selected knowledge source, which then modifies the blackboard's content. Once this knowledge source finished its execution, the loop starts over again. It ends if the blackboard contains a valid final result. This data is considered as the system's final output. Implementing the control component as a *Template Method* [GoF95] supports to vary the heuristics that selects the knowledge source to be executed and the evaluation of whether or not the blackboard contains a final result.

## Model-View-Controller \*\*



We are designing an interactive software system. Or we are specifying the clients and servers in a *Broker* (7) architecture or an agent in a *Presentation-Abstraction-Control* (24) configuration.

\*\*\*

**Most human-computer interfaces are prone to change requests: many customers call for specific adaptations and often they must support different look and feels. However, changes to an application's interface must not affect its core functionality, as it is generally independent of presentational aspects.**

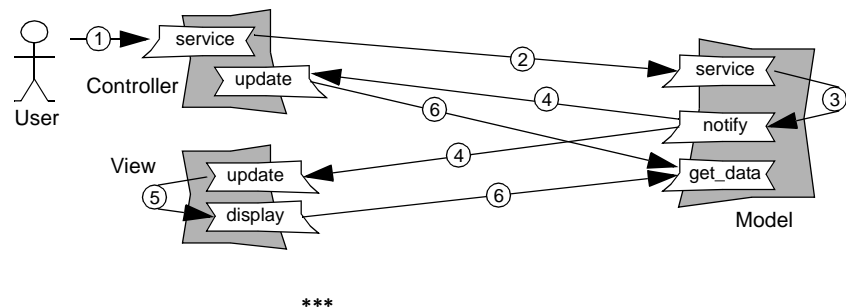
On the one hand, changes to a human-computer interface should be both easy and local to the modified interface part. On the other hand, a changeable user interface must not degrade the quality of service of an application: at any time its interface must display the current state of computation and respond to state changes immediately.

Therefore:

**Divide an interactive application into three loosely coupled parts: processing, input, and output. Encapsulate the application's core functionality inside a pure service component, the model, whose implementation is independent of specific user interface APIs and data. Such encapsulation avoids to touch this core in response to user interface changes. To display data to the user, introduce separate view components for each functional aspect of the model. Multiple views onto the same functional aspect are also possible. Such separation ensures that views can change independently—without affecting one another or the model. Associate each view with a set of separate controller components that receive user input and translate this input into service requests for either the model or their associated view. Let users interact with the system solely through the controllers. Thus, the way how users interact with the application can change without affecting the views and the model.**



**To support an effective cooperation between model, views, and controllers without breaking their separation and decoupling, connect them via a change propagation mechanism. When the model changes its state in response to a request, notify all views and controllers about this change so that they can update their state accordingly and immediately via the model's APIs.**



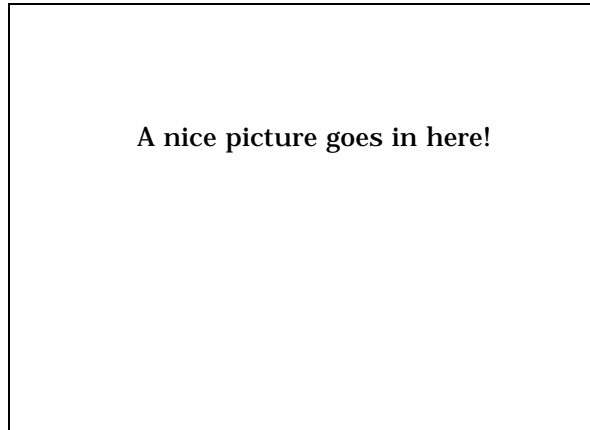
To ensure the model's independence of the user interface, make sure that its implementation does not rely on specific user interface APIs and I/O data formats. The internal design of the model depends on the type and complexity of the application's particular functionality. Domain-specific patterns and pattern languages can help with an effective model decomposition. To keep views and controllers independent of the model's design, provide it with a *Separated Interface* (not yet documented). This supports changing the model's implementation without affecting views and controllers. In addition, it allows to access the model remotely from (thin) user interface clients and is the key for implementing a concurrent model.

For every data or state information that the application presents to its users in a specific form, introduce a self-contained view. Each view encapsulates the functionality to retrieve the required data from the model and to display this data to the user. Such self-containment allows to change views without side-effects onto other application parts. To keep views independent of the system's platform and its output devices, access low-level APIs for device drivers and graphical libraries only via *Wrapper Facades* [POSA2]. A *View Handler* [POSA1] helps with organizing the entirety of all views within the application.

To manipulate the model, define one or more controllers for every view in the system. Examples include windows that frame the documents to be displayed and menus and dialogs to manipulate these documents. Controllers are typically created by *Factory Methods* [GoF95] when their associated view is activated and deleted via *Disposal Methods* [Hen02] when their view is closed. Each controller receives user input through an associated input device, such as a keyboard or a mouse, and translates this input into service requests for either its corresponding view or the model. To decouple controllers from views and the model, encapsulate user requests into *Commands* [GoF95] and pass them to a *Command Processor* [POSA1] for execution, and simplify *Command* creation and disposal by using an *Abstract Factory* [GoF95]. Such a design allows to change controllers transparently for the views and the model, and to treat requests as first class objects, which in turn enables an application to offer 'house-keeping' services like undo/redo and request scheduling. To keep controllers independent of the system's platform, access low-level APIs for device drivers only via *Wrapper Facades* [POSA2].

Typically, multiple controllers are active at the same time, but each user input can only be processed by one specific controller. To avoid implementing a complex user input dispatching mechanism, connect all controllers to a *Chain of Responsibility* [GoF95]. Input is passed along this chain until it arrives at the controller that can handle it. To support an effective and efficient collaboration between model, views, and controllers without breaking the model's independence, connect them via a *Publisher-Subscriber* [POSA1] arrangement. The model is a publisher; views and controllers are its subscribers. All views and controllers register with the model to be notified about changes of its state. When this happens, the model notifies all registered views and controllers, which in turn update their own state by retrieving the corresponding information from the model.

## Presentation-Abstraction-Control \*



We are designing an interactive software system. Or we are specifying the clients and servers in a *Broker* (7) architecture.

\*\*\*

**A human-computer interface allows users to communicate and interact with a software system via a particular ‘paradigm,’ such as command lines, forms, or menus and dialogs. However, some interactive applications support multiple distinct services where each service demands a specialized interface. Sometimes, these interfaces must even follow different paradigms.**

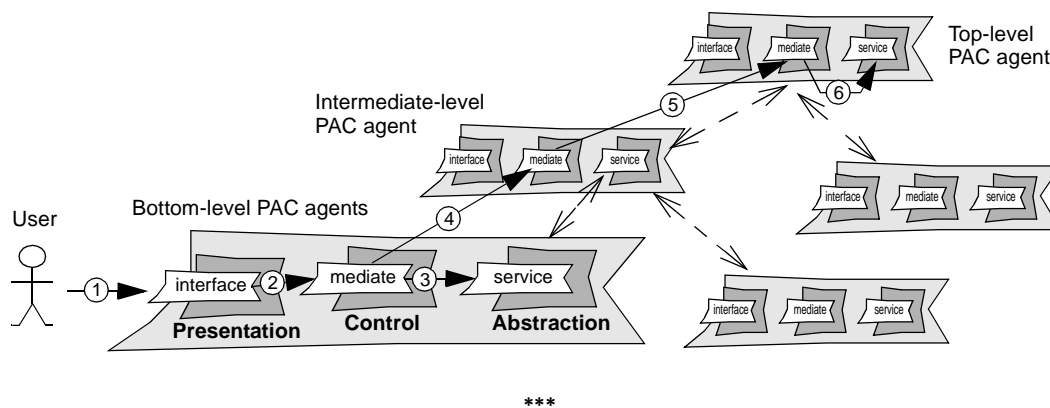
For example, administration services often require a different interface than services for entering and manipulating data. Yet we must ensure that all services as well as their associated interfaces form a coherent system. In addition, changes to any interface should neither affect its corresponding service implementation nor that of other services and their associated interfaces. Likewise, changes to the implementation of any distinct service should not affect interfaces and implementations of other services.

Therefore:

**Structure the interactive application as a tree-like hierarchy of loosely coupled agents: one top-level agent, several intermediate-level agents, and even more bottom-level agents. Every agent is responsible for a specific part of the application’s functionality—including its associated user interface. Bottom-level agents implement self-contained services with which users can interact, for instance administration, error handling, or entering, manipulating, and visualizing data. Intermediate-level agents coordinate multiple related bottom-level agents, for example, all views that visualize a particular type of data. The top-level agent provides core functionality that is shared by all agents, such as access to a data base. Such decoupling supports to modify agents independently—without affecting other agents.**

**Split every agent into three parts. A presentation defines the agent's user interface. An abstraction maintains the agent's data and provides functionality that operates on this data. A control connects the presentation with the abstraction, and allows the agent to communicate with other agents. This strong separation of an agent's functionality from its user interface supports to vary each part as independently as possible.**

**Minimize inter-agent dependencies by loosely coupling the controls of related agents. Users interact with an agent solely via its presentation. Mediate all user requests via the agent's control to the respective service implementation in its abstraction. If some user action requires to access or coordinate other agents, mediate this request to the controls of these agents, either up or down the hierarchy, and from there to their abstractions.**



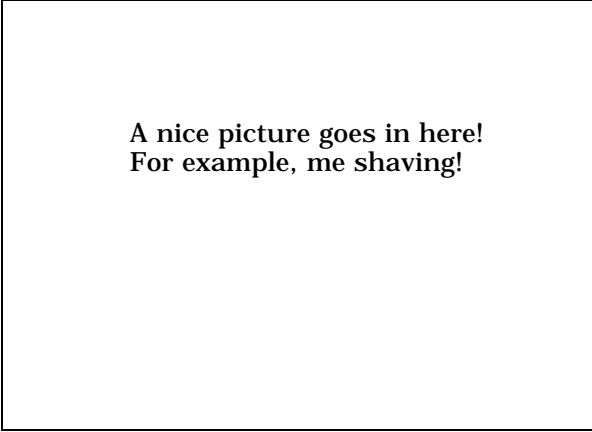
To specify a *Presentation-Abstraction-Control* (PAC) architecture, identify all self-contained services or sets of related services that the application should offer to its users. Each service (set) becomes a separate bottom-level agent. If several agents share functionality or need coordination, factor this (coordination) functionality out into an intermediate-level agent. There may be multiple levels of intermediate-level agents within a PAC architecture. If certain functionality and services are shared by all agents, encapsulate them into the top-level agent. Such a partitioning keeps an application's services as independent as possible and allows each agent to provide its own user interface. Provide all agents with a *Model-View-Controller* (24) architecture: the abstraction corresponds to the model of MVC and the presentation to the views and controllers. Thus, changes to an agent's interface do affect its implementation. Due to the strong agent separation such changes also do not affect the interfaces and implementations of other agents.

Decouple an agent's abstraction from its presentation via a *Singleton* [GoF95] control that is a *Mediator* [GoF95] with a two-fold responsibility. First, it must route all user requests from the agent's presentation to the appropriate service implementation in its abstraction. Likewise, it must route all change propagation notifications from the abstraction to the views in the presentation. Second, the control must coordinate the cooperation between agents. If a user request that was received through the agent's presentation cannot be handled by the agent alone, the control routes this request to the controls of appropriate higher-level or lower-level agents, together with its associated data. Likewise, the control of an agent can receive requests and data

from the controls of other agents, to route them to its abstraction. The data to be routed is application-specific and can be encapsulated inside *Composite Messages* [PloPD2] or *Mementos* [GoF95]. Controls enable a loose coupling between agents. If an agent's abstraction changes, affects to other agents are limited to their controls. If the control has a *Separated Interface* (*not yet documented*), it is possible to access an agent remotely as well as to implement concurrent agents.

To keep agents consistent to one another, connect them via a *Publisher-Subscriber* [POSA1] arrangement. An agent that is interested in the state of its higher-level or lower-level agents registers its control as a subscriber of these other agents' controls, which play the role of publishers. Whenever one of these 'publisher' agents changes its state, its control notifies the control of the 'observing' agent, which can then react appropriately.

## Reflection \*\*



A nice picture goes in here!  
For example, me shaving!

We are designing a long-living application that must evolve over time. Or we are specifying a *Broker* (7) architecture or a run-time information mechanism for a component's *Extension Interfaces* [POSA2].

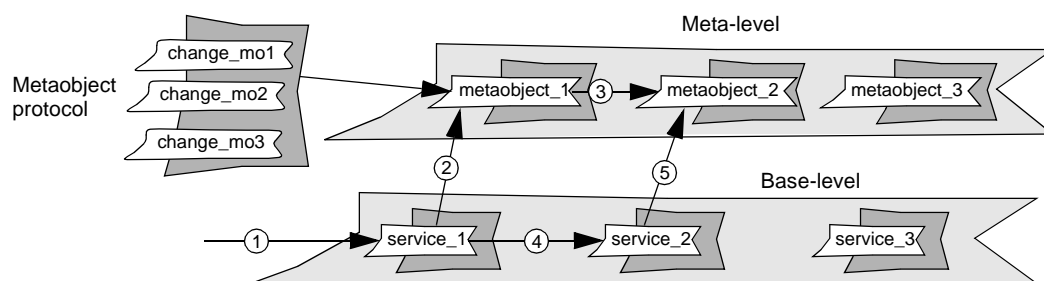
\*\*\*

**Long-living applications likely evolve over time. For example, they must respond to changing technologies, requirements, and platforms. However, it is next to impossible to anticipate and forecast all future change requests for an application at the time this application is under development.**

In addition, changes to the application can be of any scale, ranging from local adjustments of an algorithm to fundamental modifications of its distribution structure. Changes can also happen at every time in the life of the application, in particular while it is in operation. Yet, modifying the application should be easy: the complexity associated with changes should be hidden from maintainers and there should be a uniform mechanism for performing all changes.

Therefore:

**Objectify all varying aspects and properties of the application's structure, behavior, and state into a set of metaobjects to make them explicitly accessible and thus (ex)changeable. Separate the metaobjects from the rest of the application logic by introducing a two-layer architecture: a meta level contains the metaobjects, a base level the application logic. This decouples the invariant aspects of an application from its varying aspects. Provide the meta-level with a metaobject protocol, a specialized interface that maintainers can use to configure and modify all metaobjects at run-time—under the supervising control of the application. Evolving an application thus becomes as easy, uniform, dynamic, and secure as possible. Connect the base level with the meta level such that base-level objects first consult an appropriate metaobject before they execute behavior or access state that potentially can vary. Thus, changes to metaobjects immediately impact the application's subsequent behavior.**



\*\*\*

To specify a *Reflection* architecture for a given application, first design an 'ordinary' architecture for it that does not consider evolution at all. The shape of this architecture depends on the type, functionality, and complexity of the application under development. Architectural patterns as well as domain-specific patterns and pattern languages can help with an effective architecture specification.

Using a suitable method, such as the Open Implementation Analysis and Design Method [KLLM95] or the Commonality/Variability Analysis [Cope98], then identify all structural and behavioral aspects of the application that can vary. Variant behavior often includes application services and algorithms, component lifetime control, transaction protocols, IPC mechanisms, and strategies for security and failure handling. There can even be the need for adding completely new behavior to the system; or for removing existing behavior. Structural aspects that can vary include the application's thread or process model, the assignment of components to processes and threads, the location of components, or even the system's type structure. In addition to identifying the varying behavior and structure itself, determine all system-wide properties and global state which can influence the variable structure or behavior of the application.

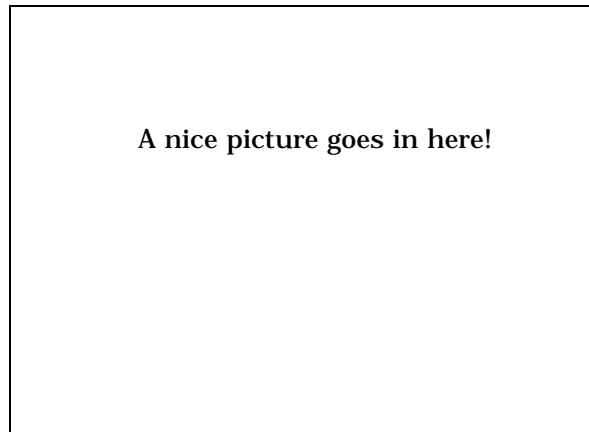
Encapsulate every varying behavioral and structural aspect, system property, and state that was identified in the above analysis into a separate metaobject that is connected loosely with its corresponding application components. For instance, *Strategies* [GoF95], *Bridges* [GoF95], and *Template Methods* [GoF95] help to encapsulate varying behavior. *Visitors* [GoF95], *Decorators* [GoF95], and *Interceptors* [POSA2] support adding or removing behavior. *States* [GoF95] allow to implement behavior that depends on a certain state. Concrete state itself can be encapsulated by a *Memento* [GoF95]. *Abstract Factories* [GoF95] and *Builders* [GoF95] help to reify component lifetime management, an *Acceptor-Connector* (62) arrangement the connection strategies for distributed peers, and a *Forwarder-Receiver* [POSA1] structure an application's IPC mechanisms. All objectified system aspects become first class entities within the application. This enables software evolution, because we can exchange these aspects' implementations without the need to modify the application components that depend on them.

Separate metaobjects from application components by introducing a *Layers* (17) architecture: a meta level contains the metaobjects, a base level the application components. This supports a centralized management, maintenance, and control of all metaobjects. Provide a metaobject protocol for the meta level that allows maintainers—or even the reflective application itself—to create, configure, modify, and dispose metaobjects via appropriate *Abstract Factories* [GoF95]

and *Builders* [GoF95]. A *Component Configurator* [POSA2] supports to reconfigure metaobjects at run-time and to dynamically upload new metaobjects that were developed after the reflective application was put into operation. Thus, the metaobject protocol hides the complexity of software evolution behind a 'simpler' interface. It also allows the reflective application to supervise its own evolution so that uncontrolled changes are minimized. Via the *Component Configurator* it is also possible to replace the *Abstract Factories* and *Builders* that manage the metaobjects. Such a design adds flexibility to the meta level itself and thus even more flexibility to the reflective application.



## Microkernel \*\*



We are building an application that exists in different versions. Or we are specifying the clients and servers in a *Broker* (7) architecture.

\*\*\*

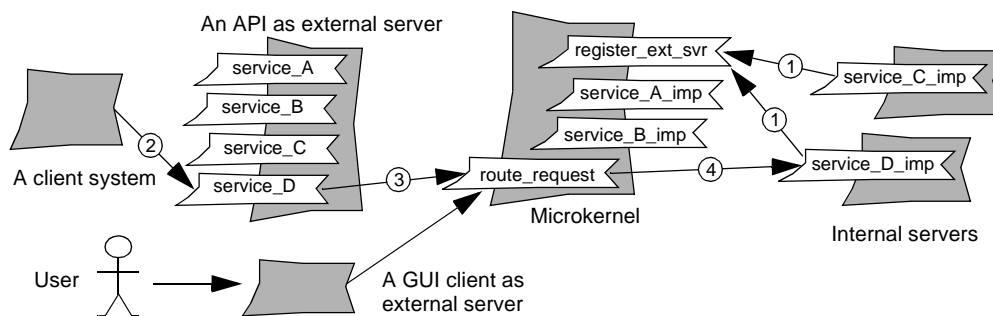
**Some applications exist in multiple versions. Each version offers a slightly different set of services to its users or varies from other versions regarding certain properties, such as its user interface or the platform on which it runs. However, despite their differences, all versions of the application should be based on a common architecture and functional core—to avoid an architectural drift between the versions and to minimize development and maintenance effort for shared functionality.**

In addition, upgrading one version of the application to another by adding and removing services, or by changing their implementation, should require no or only minimal modifications to the system. Likewise, it should be easy to both provide a particular application version with different user interfaces and to let it run on different platforms. Only then clients can use it most appropriately within their environment. ‘Traditional’ frameworks are infeasible for resolving all aspects of this problem, however: they only support application families well whose members provide different implementations of, and user interfaces for, the same set of services, but not application families whose members can differ in the services they offer.

Therefore:

**Compose different versions of the application from three types of components. A microkernel implements services shared by all application versions as well as a ‘plug-and-play’ infrastructure for integrating version-specific services. Internal servers implement self-contained version-specific services and external servers version-specific user interfaces or APIs. Configure a specific application version by connecting the corresponding internal servers with the microkernel and providing appropriate external servers to access its functionality. Consequently, all versions of the application share a common functional and infrastructural core but provide a tailored service set and look-and-feel.**

**Clients of the application—whether they are human users or other software systems—access its services solely via the interfaces or APIs provided by the external servers. These forward all requests that they receive to the microkernel. If the microkernel implements the requested service itself, it executes this service. Otherwise it routes the request to the corresponding internal server. Results are returned accordingly so that the external servers can display or deliver them to the client.**



\*\*\*

A *Microkernel* architecture ensures that every application version is tailored exactly for its purpose. Users or client systems only get the services and look-and-feel that they require; they must not pay for anything that they do not need. In general, evolving a particular version towards new or different services and properties only requires to reconfigure it with appropriate internal and external servers. The microkernel itself is not affected by such upgrades, neither are existing internal and external servers, nor other application versions. In addition, a *Microkernel* architecture minimizes development and maintenance efforts for all members of the application family: every service, user interface, or API is implemented only once.

The structure of the microkernel typically follows a *Layers* (17) architecture. The bottom-most layer abstracts from the underlying system platform; thus it supports the portability of all higher levels. The second layer implements all system services on which the functionality provided by the microkernel depends, such as resource management services. The layer above hosts the functionality that is shared by all application versions. The topmost layer includes the mechanisms for configuring internal servers with the microkernel as well as for routing requests from external servers to their intended recipient. There are two basic options for implementing the latter functionality. It can be implemented as a *Mediator* [GoF95] that receives requests through a uniform interface and dispatches these requests onto corresponding functional services in the microkernel or the internal servers. Alternatively it can be implemented via *Extension Interfaces* [POSA2] with at least one interface for the microkernel and every internal server. To keep resource consumption low, particularly memory, the routing layer can use a *Component Configurator* [POSA2] to load internal servers on demand and to unload them after usage. This design also supports to upgrade a particular application version with new, different, or modified functionality dynamically at run-time.

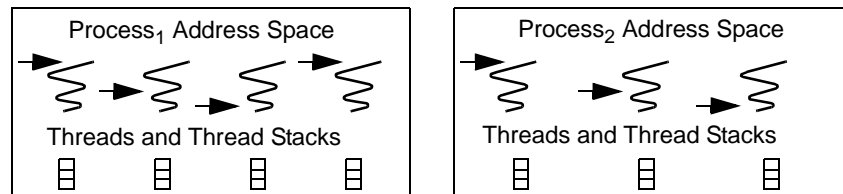
Internal servers follow a similar design, but do not provide a routing layer. In addition, if the functionality of an internal server builds on system services and platform abstractions that are offered by the layers in the microkernel, they can save these services and abstractions in their own implementation and instead call back the corresponding layers in the microkernel. This keeps the server's footprint small. To minimize network traffic in a distributed system and to increase the performance of internal servers, however, it may be beneficial to provide them with all system services and platform abstractions that they need.

The design of external servers strongly depends on their complexity. It can range from simple *Adapters* [GoF95] that map the application's published APIs onto its internal APIs to complex (graphical) user interface designs.

## Concurrency

Many distributed systems can benefit from concurrency, in particular servers and server-side components that must handle requests from multiple clients simultaneously. Therefore, developers of distributed systems often need to become proficient with various process and thread management mechanisms.

A *process* is a collection of resources, such as virtual memory, I/O handles, and signal handlers, that provide the context for executing program instructions. Thus, processes serve as *units of protection and resource allocation* within hardware-protected address spaces. In contrast, a *thread* is a single sequence of instruction steps executed in the context of a process [Lew95]. In addition to an instruction pointer, a thread consists of resources, such as a run-time stack of function activation records, a set of registers, and thread-specific data. Thus, threads serve as *units of execution* that run inside a process and share its address space with other threads:



Using multiple processes and threads yields several benefits. In particular, processes and threads can help to:

- *improve performance transparently* by using the parallel processing capabilities of hardware and software platforms.
- *improve performance explicitly* by allowing programmers to overlap computation and communication service processing.
- *improve perceived response time* for interactive applications, such as graphical user interfaces, by associating separate threads with different service processing tasks in an application.
- *simplify application design* by allowing multiple service processing tasks to run independently using synchronous programming abstractions, such as two-way method invocations.

It is remarkably hard, however, to develop efficient, predictable, scalable, and robust concurrent applications [Lea99]. Concurrent programming is much more than just starting individual components, objects, or services in their own threads of control and letting these threads run at their own discretion. Instead, executing complex and cooperating services concurrently as well as efficiently and reliably requires appropriate, often sophisticated concurrency models. This is due to several challenges of concurrent programming:

- *Application diversity.* Different types of application or component expose different structural and behavioral characteristics. For example, some applications or components use a mixture of both asynchronous and synchronous service processing, others are event-driven, and yet others must handle service requests with different priorities. Therefore, each application or component type demands its specialized concurrency architecture in order to execute most

effectively as well as to provide the required quality of service to its users—there exists no ‘one-size-fits-all’ concurrency model.

- *Multi-threading costs.* Designers of concurrent programs must always take into account that multi-threading comes at certain costs, in particular costs related to thread management, context switches, synchronization, and data movement. Thus, a naïve use of threading mechanisms can result in overhead that reduces, or even outweighs, the benefits of concurrency. It is a challenge to define a concurrency model that minimizes—or even avoids—the costs that are inherent to using multiple threads.
- *Multi-threading hazards.* Accessing a component or resource that is shared by multiple threads requires to protect its internal state from corruption through undesired simultaneous access. However, using available synchronization mechanisms incorrectly and inappropriately does not only incur unnecessary synchronization overhead. It can also result in application misbehavior due to race conditions and deadlocks. Thus, appropriate synchronization *strategies* are required to perform a proper, yet efficient component or resource access synchronization. In addition, different types of component and resource often require different synchronization strategies, dependent, for example, on their service and access profiles as well as on their size and internal complexity.
- *Portability.* Additional accidental complexity in concurrent programming arises from limitations with existing development methods, tools, and operating system platforms. In particular, the heterogeneousness of contemporary hardware and software platforms complicates the development of concurrent applications and tools that must run on multiple operating systems.

Consequently, a concurrency architecture can never be specified *ad hoc* or very late in the development life-cycle of a software system. Instead, it must be specified *a priori* when creating the system’s base-line architecture or the core design of its subsystems and components. Only then we can take into explicit account the many challenges and complexities that arise in the context of concurrency and handle them conciously and thoughtfully.

Our distributed computing pattern language therefore includes five patterns that offer proven and efficient solutions to various types of concurrency architecture and design problems:

The *Half-Sync/Half-Async* architectural pattern (39) [POSA2] decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance. It introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing.

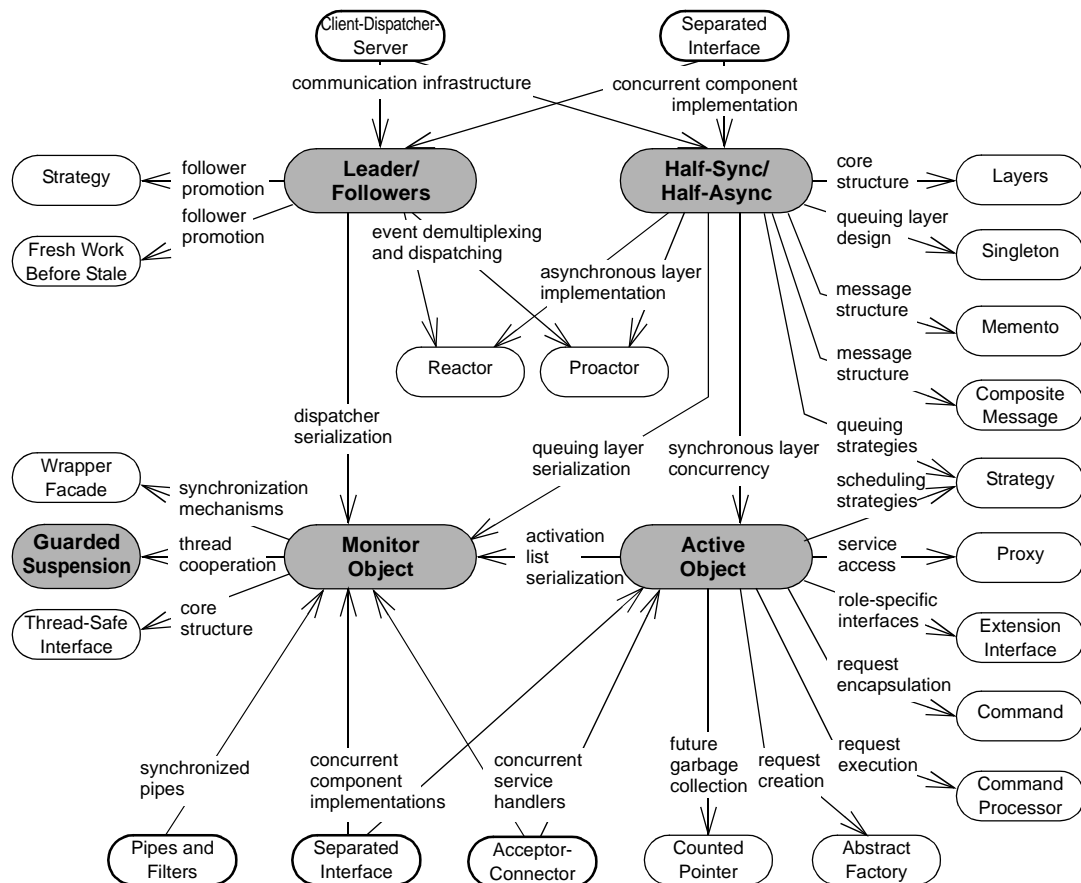
The *Leader/Followers* architectural pattern (41) [POSA2] provides an efficient concurrency model: multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process requests that occur on the event sources.

The *Active Object* design pattern (44) [POSA2] decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

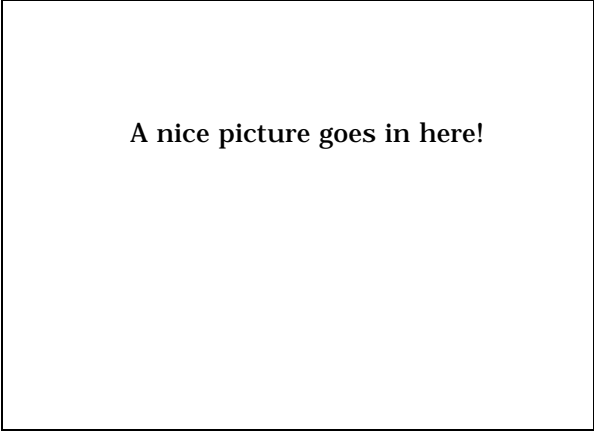
The *Monitor Object* design pattern (47) [POSA2] synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object’s methods to cooperatively schedule their execution sequences.

The *Guarded Suspension* design pattern (50) [Lea99] provides a client-thread-transparent access coordination to shared objects whose methods can only execute when certain conditions hold.

Note that the set of our concurrency patterns does not include *Thread-Specific Storage* (277), which was classified as a concurrency pattern originally [POSA2]. From today's perspective, however, we think that *Thread-Specific Storage* is less about concurrency but more about avoiding locking overhead. Therefore, we assigned it to Section 4.8, *Synchronization*. Note also, that the current version of this chapter only includes the concurrency patterns published in the *Pattern-Oriented Software Architecture* series [POSA1] [POSA2]. Other known concurrency patterns, for example, the *Scheduler* pattern [Lea99] are not yet included in this version of our pattern language. The five concurrency patterns are embedded into our distributed computing pattern language as follows:



## Half-Sync/Half-Async \*



A nice picture goes in here!

We are developing a concurrent application that performs both synchronous and asynchronous service processing. Or we are designing a concurrent component with a *Separated Interface* (*not yet documented*) or the communication infrastructure of a *Client-Dispatcher-Server* (10) arrangement.

\*\*\*

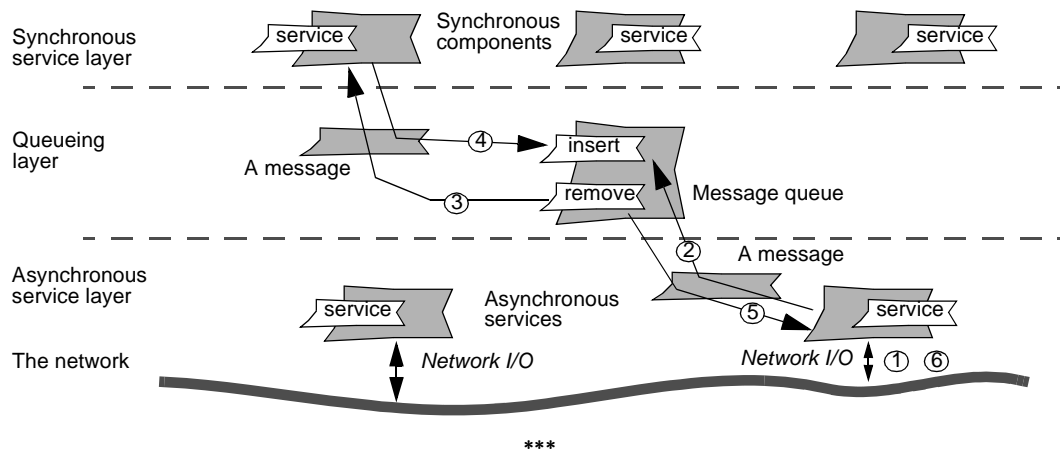
**Many complex concurrent applications perform a mixture of asynchronous and synchronous processing. Asynchronous processing is preferred to execute low-level system services most efficiently. Synchronous processing is used to offer developers a simple programming model for implementing application components. However, asynchronous services and synchronous components rarely act independently of one another. Instead, their control flows are interwoven in order to provide and perform the application's functionality in concert. Yet each type of processing should not suffer from the deficiencies of the other.**

In particular, the performance of asynchronous services should not degrade due to their dependencies to 'slow' synchronous components. Likewise, the programming simplicity of synchronous components should not be affected by the complexity associated with asynchrony. Nevertheless, asynchronous and synchronous services must be able to cooperate effectively with one another.

Therefore:

**Decompose the functionality of the concurrent application into two separated layers—synchronous and asynchronous—and add a queuing layer to mediate the communication between them. Such a design centralizes where asynchronous and synchronous processing happens within the application, ensures that both locations are as decoupled as possible, and allows asynchronous and synchronous services to cooperate with one another. Execute higher-level application components and services, such as for domain functionality, long-duration database queries, or file transfers, synchronously in separate**

**threads or processes to simplify concurrent programming. Conversely, process lower-level system services asynchronously to enhance performance, such as short-lived protocol handlers that are driven by interrupts from network interface hardware. If services in the synchronous layer must communicate or synchronize their processing with services in the asynchronous layer, allow them to pass messages to one another via the queueing layer.**



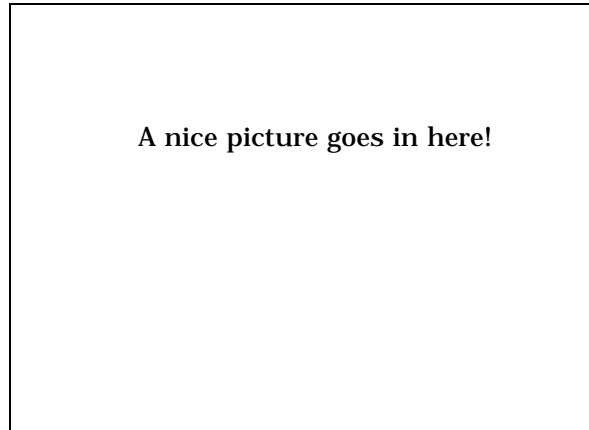
To keep the three layers independent and self-contained—which allows them to preserve their designated properties—structure them as a *Layers* (17) architecture.

A convenient way to support concurrency for the components in the synchronous layer without breaking their self-containment is to implement each component as an *Active Object* (44). In contrast, services in the asynchronous layer are programmed by using either low-level asynchronous interrupts or signals, or a *Proactor* (58) that resides atop the operating system. The latter strategy is specifically useful if asynchronous services are based on higher-level operating system APIs, such as Windows overlapped I/O and I/O completion ports or the POSIX `aio_*` family of asynchronous I/O system calls. An alternative strategy is to implement the asynchronous layer using a *Reactor* (55). Though a *Reactor* is not truly asynchronous, it shares key properties with asynchronous services if—and only if—these services implement short duration operations. In this case, event handlers dispatched by a *Reactor* cannot block for a long time and exclude other event sources from being serviced. The advantage of a *Reactor*-based design is its simplicity compared to using native asynchrony mechanisms or a *Proactor*.

The queueing layer consists of a *Singleton* [GoF95] message queue that is shared by all services in the synchronous and asynchronous layers. Such a design ensures that both layers do not maintain direct dependencies to one another. In most *Half-Sync/Half-Async* architectures the message queue is implemented as a *Monitor Object* (47). This serializes the access to the message queue transparently for asynchronous and synchronous services. If it is necessary to configure the message queue according to different requirements, encapsulate its queuing and communication functionality within *Strategies* [GoF95]. *Strategies* also support to vary this functionality independently of the services in both the asynchronous and synchronous layer. Messages to be routed by the queueing layer are application-specific and can be encapsulated inside *Composite Messages* [PLoPD2] or *Mementos* [GoF95].



## Leader/Followers \*\*



We want to react on and process multiple events both concurrently and efficiently. Or we are designing a concurrent event-driven component with a *Separated Interface (not yet documented)* or the communication infrastructure of a *Client-Dispatcher-Server* (10) design.

\*\*\*

**Multi-threading is a common technique to allow event-driven applications to process multiple events concurrently. However, it is surprisingly hard to implement high-performance multi-threaded event-driven applications because such applications cannot tolerate much concurrency overhead.**

For instance, though it is necessary to prevent race conditions if multiple threads demultiplex events on a shared set of event sources, key sources of concurrency-related overhead, such as context switching, synchronization, and cache coherency management, must be minimized to avoid performance penalties. A thread-per-request concurrency model is thus no reasonable design option. Likewise, though it is necessary to define efficient demultiplexing associations between threads and event sources, associating a dedicated thread for each event source is often infeasible, due to scalability limitations of applications, operating systems, and networks.

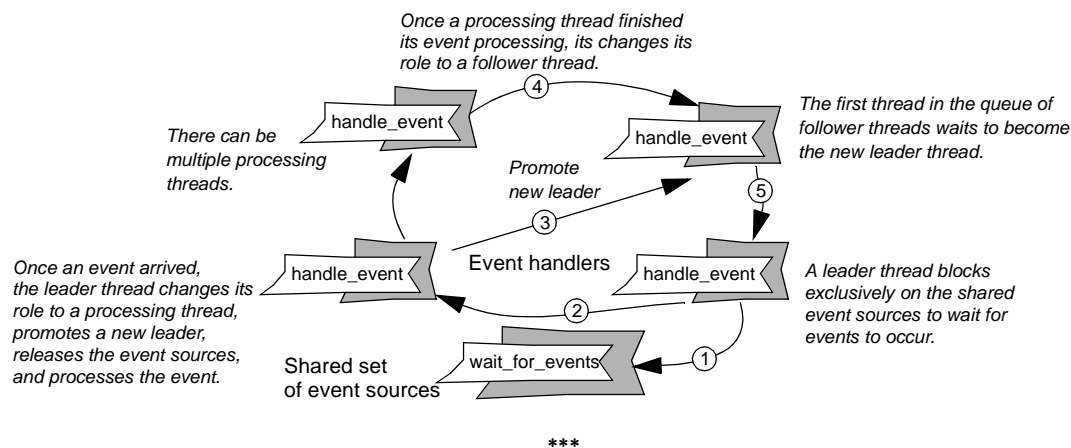
Yet, despite all these challenges, a concurrent reactive application should offer a simple programming model to developers.

Therefore:

**Introduce a thread pool in which multiple threads can coordinate themselves while detecting, demultiplexing, dispatching, and processing events. Using a thread pool avoids overhead related to dynamic thread creation and deletion. In this thread pool allow only one thread at a time—the leader—to wait for an event to occur on a set of shared event sources. This simple event source access strategy incurs only minimal synchronization overhead.**

**While the leader is listening on the event sources for an event to occur, other threads—the followers—can queue up sleeping until they become the leader. Sleeping followers do**

not waste precious CPU cycles. After the current leader thread detects an event from the event sources, it first promotes a follower thread to become the new leader. Letting the leader thread perform this promotion avoids performance bottlenecks related to a centralized thread management. The old leader then plays the role of a processing thread: it demultiplexes and dispatches the event to a designated event handler that performs application-specific event handling in the processing thread. Multiple processing threads can handle events concurrently while the current leader thread waits for new events to occur on the set of event sources shared by the threads. This design maximizes the number of threads that concurrently can handle and process events. After handling its event, a processing thread reverts to the follower role and waits to become the leader thread again.



Encapsulate the shared event sources within a dispatcher component that is either a *Reactor* (55) or a *Proactor* (58). This design separates the event demultiplexing and dispatching mechanism from application logic and makes event handling efficient. Yet it offers a simple and flexible programming model to application developers. In addition to methods for accessing the shared event sources, provide the dispatcher with methods for deactivating and reactivating a specific event source. Otherwise, there is a potential for race conditions to occur in a *Leader/Followers* architecture in the time interval that begins when a new leader thread is selected and ends when the processing of the most recent event finished. If during this interval the new leader waits on the event source on which the last event occurred, it could demultiplex this event a second time. This is erroneous because the event's dispatch is already in progress.

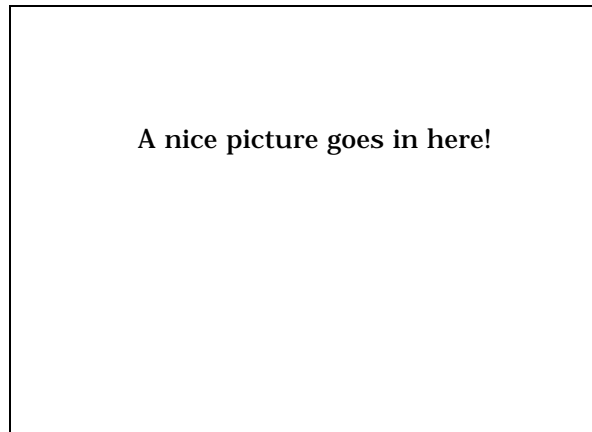
Specify the thread pool as a *Monitor Object* (47) that maintains the dispatcher and is responsible for synchronizing the access to the shared event sources. Such a design ensures that the thread pool infrastructure is accessible from *within all* threads in the pool and also enables these threads to coordinate their execution sequence cooperatively. This, in turn, supports scalability because a self-organizing concurrency model avoids performance and failure penalties resulting from consulting a separate, singleton 'bottleneck' component for every thread scheduling and cooperation decision.

The thread pool itself offers two methods to its threads. One method allows newly initialized threads to join the pool. This method implements an infinite loop where the joining thread first waits for becoming the new leader by suspending its own execution on the thread pool's monitor condition, and second—once it received this role—accesses the shared event sources to wait for and process an incoming event. Suspending a thread from execution while it is waiting prevents this thread from being scheduled for execution and waisting CPU-cycles. This improves performance: the maximum CPU-power is available for the leader and the processing threads. The second method allows the current leader thread to promote a new leader by notifying a sleeping follower via the thread pool's monitor condition. The notified follower then resumes the execution of the thread pool's join method and accesses the shared event sources to wait for the next event to occur. Multiple promotion protocols can be provided via *Strategies* [GoF95]. One strategy is to promote the follower thread that waited the shortest time to become the new leader. This *Fresh Work Before Stale* [PLoPD2] promotion maximizes performance by maximizing CPU cache coherency [Sol98].

Initialize all threads of a *Leader/Followers* thread pool during system startup. This avoids run-time thread creation and deletion overhead and helps to keep explicit control over all resources that the threads in the pool consume. When the system initializes, all threads that join the pool try to become the first leader. Only one thread will succeed; all other threads will be put asleep and become the followers. The leader then accesses the shared event sources via the dispatcher. Once an event occurs, the dispatcher calls back to an event handler within the leader thread to process this event. The leader then transitions into the role of a processing thread: its event handler deactivates the event source on which the event occurred, promotes a new leader via the thread pool's promotion method, processes the event, and reactivates the event source. When this event processing finished, control returns to the infinite loop of the thread pool's join method and the thread reverts to the follower role: it is put asleep until it is promoted to be the leader again.

Note how the threads in a *Leader/Followers* arrangement coordinate themselves implicitly via the thread pool *Monitor Object*. This avoids unnecessary performance overhead and maximizes concurrency: while one or more threads are processing events, one thread is waiting for the next event to occur, and yet other threads are ready to get access to the shared event sources. Performance can improve even further if all threads are able to process all events that the application can receive: then it does not matter which thread processes what event. If this is not feasible, for instance due to limited memory, introduce an event handover protocol [POSA2].

## Active Object \*\*



We are implementing components that run concurrently within their own threads of control. Or we are designing a concurrent component with a *Separated Interface* (*not yet documented*), or the synchronous service layer in a *Half-Sync/Half-Async* (39) architecture, or concurrent service handlers in an *Acceptor-Connector* (62) configuration.

\*\*\*

**Many applications can benefit from using concurrent components to improve their quality of service, for example by allowing these components to handle multiple client requests simultaneously. If a component runs concurrently, however, we must synchronize and serialize access to its services and state if this component is shared by multiple client threads.**

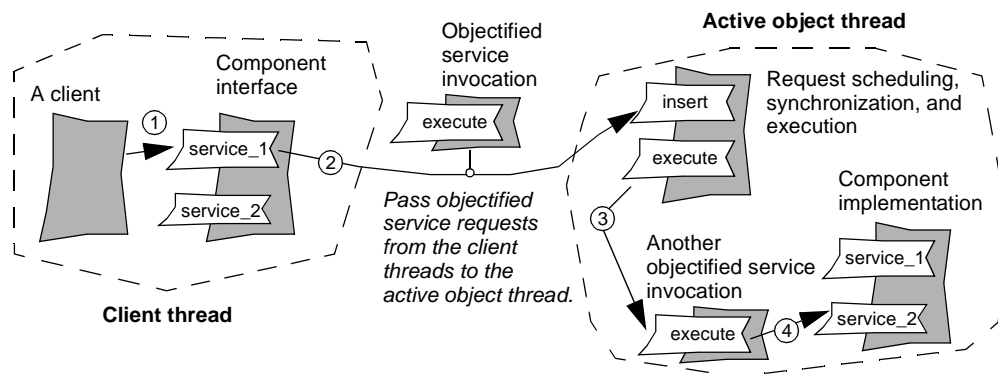
Yet, despite this need for synchronization, it is desirable to not degrade concurrency within the entire application unnecessarily. In particular, while the concurrent component executes a specific client request, other clients should still be able to issue new requests without blocking until these requests can be serviced. Likewise, a processing-intensive service invoked on the component should not block its client for the entire service execution time.

To support the component's quality of service even further, it should also be possible to schedule client requests according to certain criteria, such as request priorities or deadlines. Yet, all service invocations should be serialized and scheduled transparently to both the concurrent component and its clients—in order to keep them independent of one another and to support reusing the component within applications that require different synchronization strategies.

Therefore:

**For each concurrent component, separate service invocation from service execution. Service invocation should occur in the client's thread, whereas service execution should occur in a separate thread. This allows service invocations and service executions to run concurrently—clients thus do not block unduly when issuing a request to the component.**

**Design the decoupling so the client thread appears to invoke an ordinary service that is not subject to synchronisation constraints. This keeps clients independent from the component's synchronisation. In addition, objectify service invocations and let the component schedule their execution independently of the point in time they were issued. This enables the component to handle the entirety of all service requests according to a given quality of service criteria. Within the component, separate synchronisation and scheduling support from application functionality in order to keep the latter independent from all concurrency concerns.**



Let a *Proxy* [POSA1] or several *Extension Interfaces* [POSA2] represent the concurrent component in the threads of its clients. This allows the clients to access the component as if it were collocated in their own thread. Design the *Proxy* or the *Extension Interfaces* so that their method signatures do not include synchronisation parameters. Clients thus appear to have an exclusive access to the concurrent component even if it shared by multiple client threads.

At run-time, let the *Proxy* or *Extension Interfaces* objectify all service invocations into service requests, which are *Command* [GoF95] objects that also check all necessary synchronization constraints of their corresponding service invocations. This request objectification enables the decoupling of service invocation from service execution in both space and time. Thus, each client can invoke services on the component without blocking other clients. Simplify the creation of service requests—and their later disposal—by using an *Abstract Factory* [GoF95] and store the newly created service requests into a shared activation list, which maintains all service requests on the concurrent component that are still pending execution. Implement the activation list as a *Monitor Object* (47) to ensure a thread-safe concurrent access from within each client thread.

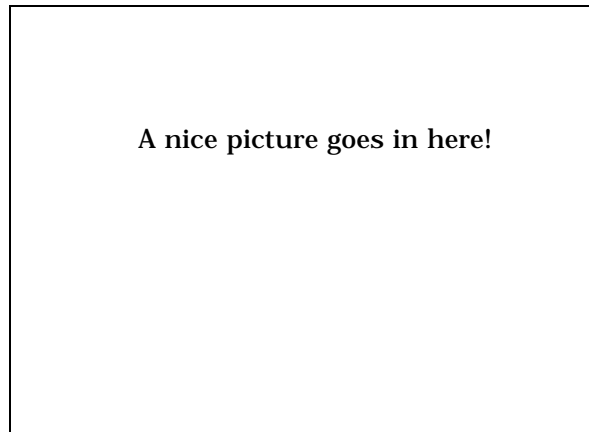
Let a separate *Active Object* thread contain the implementation of the concurrent component. This allows service invocations and service executions to run concurrently and independently, but coordinated via the thread-safe activation list and the service requests: service invocations can run in the client threads, service executions in the *Active Object* thread.

Within the *Active Object* thread, a servant provides the component's application functionality and a scheduler executes service requests on this servant according to a particular scheduling policy. Such a decomposition separates application functionality from scheduling and synchronisation

functionality. This, in turn, allows a concurrent component to handle the entirety of all pending service requests according to a given quality of service criteria. Also, both the scheduler and the servant become reusable within different concurrent applications and components. Design the scheduler as a *Command Processor* [POSA1] that implements the component's main event loop: it constantly observes the activation list to identify a service request that becomes executable, removes this request from the activation list, and executes it on the servant. *Strategies* [GoF95] support multiple scheduling policies within the scheduler.

Via a future a client can obtain the result of a service invocation on the concurrent component. The *Proxy* or *Extension Interfaces* return this future to the client after the service's invocation; the associated service request 'fills' the future after the servant finished with the service's execution. If the client accesses the future before it contains the service's result, the client blocks until this result is available. Thus, try to not access the future directly after its corresponding service invocation: this only 're-models' a truly synchronous service call. Instead, let the client execute as many different operations or instructions as possible after the service invocation and access the future at the latest point in time before using the service's result. The more time is passed between the service invocation and the access to the future, the less likely the client will block—which increases concurrency within a multi-threaded application. Used futures can be reclaimed safely via garbage collection or a *Counted Pointer* [POSA1].

## Monitor Object \*\*



We are implementing objects that are shared between multiple threads. Or we are designing a concurrent component with a *Separated Interface* (*not yet documented*), or synchronized pipes in a *Pipes and Filters* (19) system, or a *Leader/Followers* (41) thread pool, or the message queue in a *Half-Sync/Half-Async* (39) architecture. Or we are constructing the activation list of an *Active Object* (44) or concurrent service handlers in an *Acceptor-Connector* (62) configuration.

\*\*\*

**Concurrent applications and components often contain objects whose methods are invoked by multiple client threads. To prevent corrupting the internal state of such shared objects, it is therefore necessary to synchronize and schedule client access to them. However, clients should not need to distinguish between accessing shared objects and accessing non-shared objects—in order to simplify concurrent programming.**

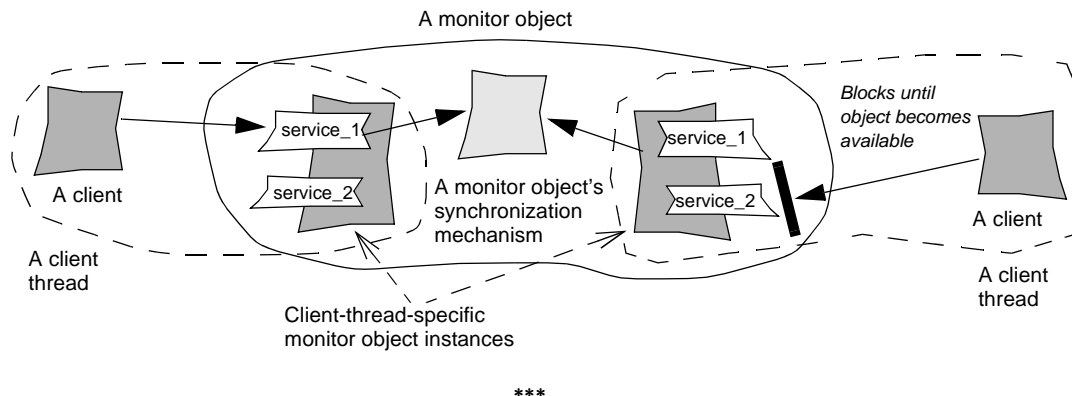
Instead, every shared object should ensure that its methods are serialized transparently without requiring explicit client intervention. To not degrade the quality of service of its clients, a shared object should also relinquish its thread of control voluntarily if either of its methods must block during execution—leaving the object in a stable state, so that other client threads can access the object safely.

Therefore:

**Execute an instance of the shared object in each of its client threads and let these instances self-coordinate a serialized, yet interlaced execution sequence. Access the shared object only through synchronized methods. This keeps the synchronization of the object transparent to its clients. In addition, only one such method can execute at any one time within all threads that contain the shared object—which ensures a proper protection of its internal state from concurrent access.**

**If a synchronized method cannot make immediate progress, suspend its execution and 'leave' the shared object temporarily by blocking its client thread. This allows clients**

**residing in other threads to access and modify the shared object. If a synchronized method could execute successfully and also changed the shared object's state, notify one or more suspended synchronized methods in other threads so that they can try to resume their execution. Such an interlaced execution sequence maximizes the availability of the shared object to its clients.**



Coordinate the execution sequence of the object's client threads via *Guarded Suspension* (50). Within a synchronized method, first acquire a monitor lock, which is a mutex lock that is typically implemented as a *Wrapper Facade* [POSA2]. As the monitor lock is shared by all client-thread-local instances of the shared object, this ensures that none of its other synchronized methods can execute—independent from which client thread it is accessed. Once the monitor lock is held, evaluate if the shared object's current state allows an execution of the synchronized method. If it does, execute it. Otherwise, suspend the execution of the synchronized method on a monitor condition. A monitor condition is a *Wrapper Facade* [POSA2] that if called suspends the thread of its caller until it is notified to awake it. When suspending a thread, the monitor condition also releases the monitor lock; when resuming this thread, it re-acquires this monitor lock. Suspending a synchronized method thus allows other client threads to access the shared object via its synchronized methods. If a synchronized method could execute, finished this execution successfully, and also changed the shared object's state, some synchronized methods that suspended themselves earlier may now be able to resume their execution. Therefore, let the finished synchronized method notify the monitor condition on which these suspended synchronized methods are waiting so that this monitor condition can awake their threads. Before terminating a synchronized method, release the monitor lock so that other synchronized methods called by other threads can execute.

Note how the interplay between the monitor conditions, the monitor lock, and the evaluation of whether a given synchronized method can execute enables multiple synchronized methods running in separate threads to schedule their execution sequences cooperatively, which maximizes the shared object's availability to its clients.

Provide the shared object with a *Thread-Safe Interface* [POSA2] that splits its methods into publicly accessible interface methods and associated private implementation methods. An interface method only synchronizes the access to the shared object: it acquires and releases its monitor lock, evaluates whether the method can execute, and waits on, and notifies, its monitor



conditions. Otherwise it only delegates control to its associated implementation method—if the synchronized method can execute. An implementation method realizes application functionality under the assumption that the monitor lock is held. It also does not wait on, or notifies, any monitor condition. Neither does it invoke other interface methods of the shared object. Thus, an implementation method can call all other implementation methods of the shared object without incurring self-deadlock or unnecessary locking overhead. A *Thread-Safe Interface* therefore decouples the synchronization of a shared object from its application functionality and allows each to vary independently.

## Guarded Suspension \*\*

We are implementing a *Monitor Object* (47) or another object that is shared between multiple threads. Or we want two or more threads to schedule their execution sequence cooperatively.

\*\*\*

Sometimes we can execute a method invoked on an object *only* if a certain guard condition holds. For example, we can insert a message into a fixed size message queue only if this queue is not full. In sequential programs, if the condition's state is false, there is no point in waiting for it to become true—we must abort the method and report failure to its caller. In concurrent programs, in contrast, we often face the situation that the state of a method's guard condition eventually becomes true as a result of another concurrent action on the same object. In our example, if another client thread removes a message from the queue, new space becomes available, and the previously invoked insert method that could not execute because the queue was full can now proceed and finish successfully. If the current state of the guard condition of a shared object's method is only temporal, however, it is not always feasible to abort this method if it cannot be executed immediately—specifically not if this method's caller cannot proceed its own computation reasonably in case the method fails.

Therefore, instead of aborting the method, suspend its client thread so that other client threads can access the shared object safely and change the state of the method's guard condition. If this state changes, resume the suspended thread so that this thread can try to continue the execution of the interrupted method.

\*\*\*

Guarded suspension allows a shared object to control and schedule the execution sequence of its client threads transparently to these threads. This minimizes concurrency overhead in the client threads and increases the shared object's availability. There are multiple ways to implement a guarded suspension [Lea99]: waits and notifications via condition variables and associated mutexes, busy-waits via spin-loops, or even suspending and resuming client threads directly.

## Event Handling

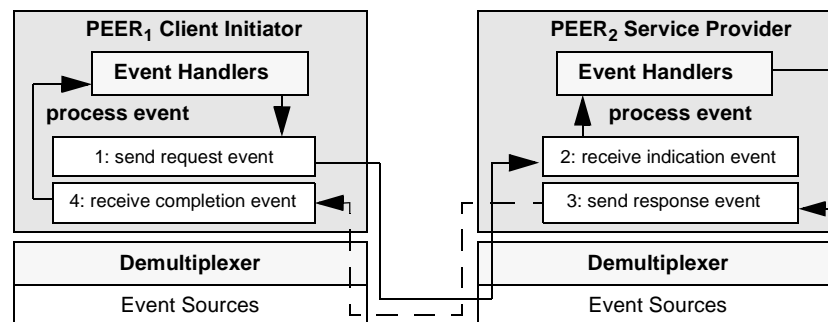
At its very heart, all distributed object computing middleware is event-driven—even if it offers a more sophisticated communication model to applications, for example, a request/response semantics or asynchronous messaging. To provide an appropriate quality of service to applications using such middleware it is thus important that their event-driven core executes efficiently. However, there are several challenges that differentiate event-driven applications from those with a ‘self-directed’ flow of control [PLoPD1]:

- *Asynchronous arrival of events.* All behavior in an event-driven application is triggered by external or internal events that can arrive at the application asynchronously, which means *at any time*, in particular while it is processing other events. However, most events must be handled promptly—even under heavy workload of the application or while it executes long-duration services—to prevent CPU starvation, improve perceived response time, and keep hardware devices with real-time constraints from corrupting data or otherwise failing.
- *Simultaneous arrival of multiple events.* An event-driven application typically receives events from multiple independent event sources, such as I/O ports, sensors, keyboards or mice, signals, timers, or asynchronous software components. Consequently, multiple events can arrive at the application at the same point in time. To react promptly on any event from any event source, therefore, an event-driven application must be able to listen for events to occur on the entirety of its event sources simultaneously. At any time no single event source must be excluded from being serviced.
- *Non-deterministic arrival of events.* Event-driven applications generally have little or no control over the order in which events arrive. Yet, an event-driven application must trigger the correct—or at least a defined—behavior in response to each event, regardless of the order in which they arrive. This requires a sophisticated event demultiplexing and dispatching infrastructure within an event-driven application. Applications that process events in a specific order must also be able to detect illegal event sequences to prevent misbehavior or incorrect state transitions.
- *Multiple event types.* Most event-driven applications distinguish multiple types of event with each type instigating a particular control flow within these applications. For example, in an event-driven, networked application, a CONNECT event indicates a request to establish a connection between two peers whereas a DATA event indicates an operation request and its parameters. In contrast to method invocations on an object, however, dispatching the correct application service in response to events is the responsibility of the event-driven application, not that of its clients. This requires an effective, yet efficient mechanism to demultiplex events onto their intended receiver and to dispatch the correct service or operation on this receiver.
- *Hiding the complexity of event demultiplexing and dispatching.* Most application behavior can be best expressed in terms of a service-oriented, synchronous request/response semantics instead of an event-driven model in which events can arrive asynchronously from multiple independent event sources. Thus, to simplify the development of event-driven applications, abstractions are needed that hide the complexity of receiving, demultiplexing, and dispatching events onto concrete application services.

To master the above challenges both elegantly and efficiently, event-driven applications are often structured as layered architectures [POSA1] with an inverted flow of control [John97]. Each layer in this architecture is responsible for handling a particular aspect in event-driven computation and hides the complexity that is associated with this aspect from the other, higher layers. A typical event-driven application distinguishes three layers :

- At the lowest layer are *event sources*, such as Sockets [Ste98], which detect and retrieve events from various hardware devices or low-level services that reside within an operating system.
- In the next layer up is an *event demultiplexer*, which uses functions like `select()` [Ste98], `poll()` [Rago93], `WaitForMultipleObjects()`, or `GetQueuedCompletionStatus()` [Sol98] to wait for events to arrive on the various event sources and then dispatches events to their corresponding *event handler* callbacks.
- The *event handlers*, together with the application code, form yet another layer that performs application-specific processing in response to callbacks.

The following diagram illustrates the core architecture of an event-driven system as well as its upward—and hence inverted—event-related control flow:



The events that are exchanged between peers in this architecture play four different roles [Bl91]:

- PEER<sub>1</sub>, the client initiator application, invokes a send operation to pass a *request event* to PEER<sub>2</sub>, the service provider application. The event can contain data necessary for PEER<sub>1</sub> and PEER<sub>2</sub> to collaborate. For example, a PEER<sub>1</sub> request may contain a `CONNECT` event to initiate a bidirectional connection, or a `DATA` event to pass an operation and its parameters to be executed remotely at PEER<sub>2</sub>.
- The PEER<sub>2</sub> service provider application is notified of the request event arrival via an *indication event*. PEER<sub>2</sub> can then invoke a receive operation to obtain and use the indication event data to perform its processing. The demultiplexing layer of PEER<sub>2</sub> can wait for a set of indication events to arrive from multiple peers.
- After the PEER<sub>2</sub> service provider application finishes processing the indication event, it invokes a send operation to pass a *response event* to PEER<sub>1</sub>, acknowledging the original event and returning any results. For example, PEER<sub>2</sub> could acknowledge the `CONNECT` event as part of an initialization 'handshake', or it could acknowledge the `DATA` event in a reliable two-way remote method invocation.

- The  $PEER_1$  client initiator application is notified of a response event arrival via a *completion event*. At this point it can use a receive operation to obtain the results of the request event it sent to the  $PEER_2$  service provider earlier.

If after sending a request event the  $PEER_1$  application blocks to receive the completion event containing  $PEER_2$ 's response, it is termed a *synchronous* client.<sup>1</sup> In contrast, if  $PEER_1$  does not block after sending a request it is termed an *asynchronous* client. Asynchronous clients can receive completion events via asynchrony mechanisms, such as UNIX signal handlers [Ste99] or Win32 I/O completion ports [Sol98].

Though the Layers approach [POSA1] decouples different concerns in an event-driven application in a way that allows us to handle each concern separately, it does not tell us how to resolve a particular concern most optimally under a given set of forces. For example, just the presence of an event-demultiplexing layer does not guarantee an efficient, yet simple demultiplexing and dispatching of events to event handlers. The four event-handling patterns in our pattern language for distributed computing help with filling this gap. They provide efficient, extensible, reusable, yet often surprisingly simple solutions to core event demultiplexing and dispatching problems in event-driven distributed systems:

The *Reactor* architectural pattern (55) [POSA2] allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

The *Proactor* architectural pattern (58) [POSA2] allows event-driven applications to efficiently demultiplex and dispatch service requests triggered by the completion of asynchronous operations, to achieve the performance benefits of concurrency without incurring certain of its liabilities.

The *Acceptor-Connector* design pattern (62) [POSA2] decouples the connection and initialization of cooperating peer services in a networked system from the processing performed by the peer services after they are connected and initialized.

The *Asynchronous Completion Token* design pattern (65) [POSA2] allows an application to demultiplex and process efficiently the responses of asynchronous operations it invokes on services.

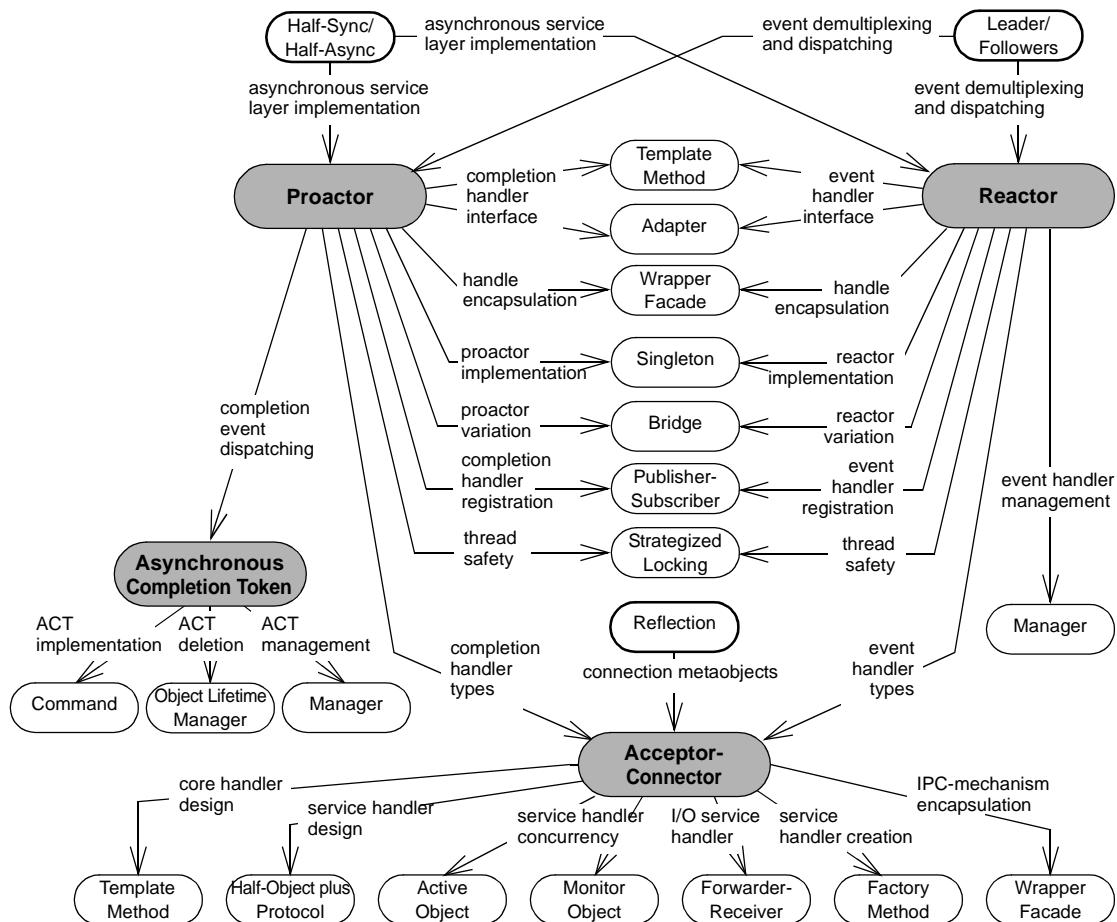
Note that the *Reactor* (55) and *Proactor* (58) patterns basically resolve the same problem in the same context: demultiplexing and dispatching events to application services within an event-driven application. Both also use almost the same set of patterns to implement their solutions. Yet the concrete event handling infrastructures that these two patterns introduce are fairly distinct, due to the orthogonal forces to which each pattern is exposed. *Reactor* focuses on simplifying application service programming. Therefore, it implements a reactive event handling model: application components wait until client events arrive and then react to process these

---

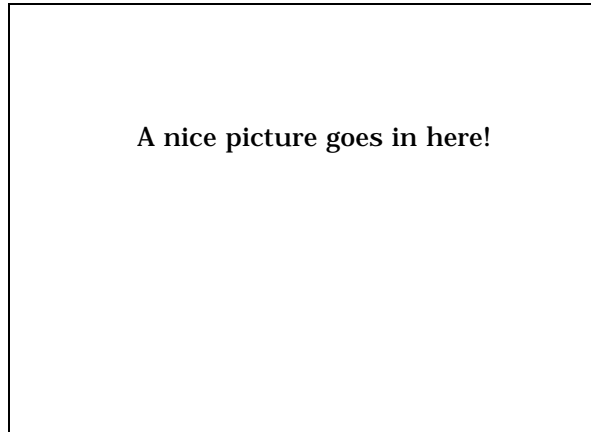
1. While reading the patterns in this book it is important to recognize that the terms 'client' and 'service' are not immutable properties of particular software or hardware components. Instead, they are *roles* [RG98] played during a particular request/response interaction. For example, in a symmetric peer-to-peer system  $PEER_1$  and  $PEER_2$  could play both the roles of client initiator and service provider at various times during their interactions.

events. *Proactor* intends to maximize the performance of an event-driven application. Thus, it implements an event handling model where application components proactively instigate service processing and then wait for particular client events to arrive so that they can continue or finish their computation.

The four event handling patterns are integrated into our pattern language as follows:



## Reactor \*\*



We are building an event-driven application. Or we are developing the asynchronous service layer in a *Half-Sync/Half-Async* (39) architecture or the event demultiplexing and dispatching infrastructure in a *Leader/Followers* (41) architecture.

\*\*\*

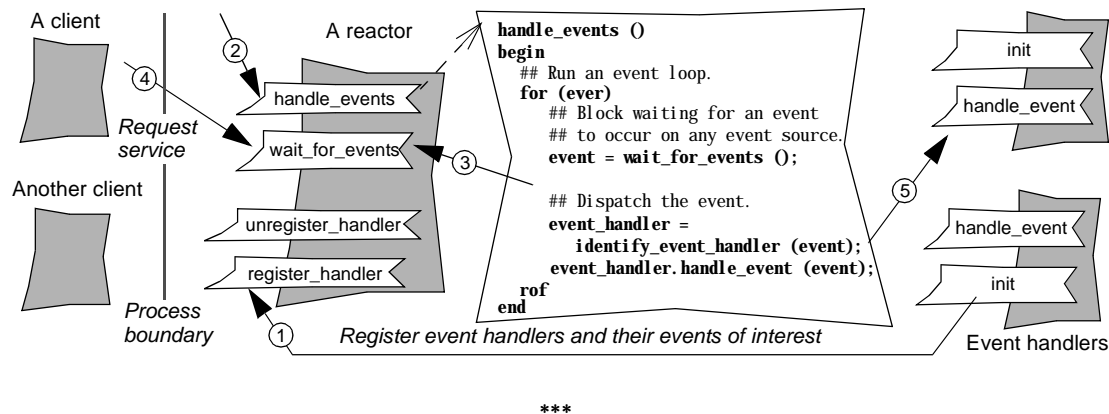
**An event-driven application typically receives multiple events from multiple sources. However, although an event indicates a request for a particular service, it does not itself specify the concrete service component and method that should process it. It is the application that must demultiplex incoming events and dispatch them to their corresponding service implementations.**

To simplify the implementation of services, the application should arrange that service request events are executed synchronously and serially, even though events from different sources can arrive simultaneously. Using multi-threading to resolve this problem may be undesirable due to the overhead of context switching, synchronization, and data movement. Likewise, blocking indefinitely on any single event source would exclude other event sources, degrading the application's quality of service to clients.

To support the evolution of an application, it should also be possible to integrate new or improved services into its event demultiplexing and dispatching infrastructure.

Therefore:

**Decouple event demultiplexing and dispatching mechanisms from application services. Introduce a reactor component that waits synchronously for events to occur in a set of event sources. Allow this reactor to detect events that arrive on multiple event sources simultaneously. When events occur on these sources, the reactor demultiplexes and dispatches them serially to event handlers that implement the application's services. These handlers then 'react' and process the events. Register event handlers with the reactor for each event of interest so that the reactor can find and invoke the 'right' handler**



Implement event handlers as *Template Methods* [GoF95]: an abstract class defines a uniform interface that represents the set of operations available to process events. Concrete event handlers derive from this abstract class, each implements a particular application service. This design provides an extensible event dispatching infrastructure: the reactor is independent of any event handler logic, it only knows a polymorphic interface. Via *Adapters* [GoF95] we can also map the event handler interface to existing application service *functions*.

The public event handler interface often consists of a single method whose implementation within the concrete event handlers dispatches all events that the handlers can process to appropriate private service methods. By this we can extend concrete event handlers to handle new event types without changing their interface. Alternatively, we can provide a separate public event handling method for each event type. Such a multi-method interface avoids a handler-internal event dispatching and makes it easy to override event handling methods selectively for a particular concrete event handler.

The *Acceptor-Connector* pattern (62) suggests to group event handlers into three categories: service handlers implement application functionality; acceptors and connectors establish connections on behalf of these service handlers. This design allows to vary connection establishment strategies independent of service processing strategies. Configure each event handler such that it only processes events that arrive on a particular event source, for example, on a particular data-mode socket endpoint. Thus, every event handler instance is uniquely identified by its event source—or better, the handle that represents this source. To simplify the use of handles within their event handlers, wrap them into *Wrapper Facades* [POSA2].

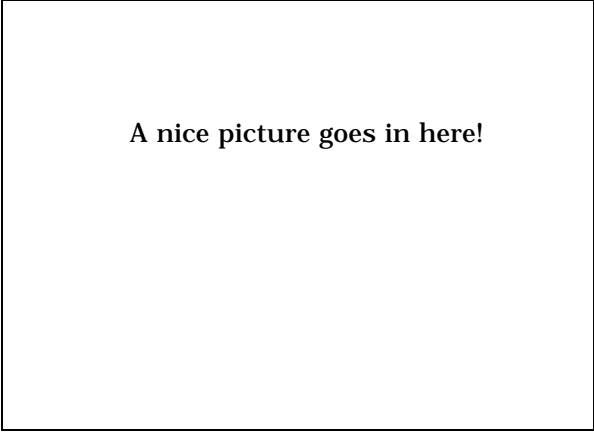
The reactor provides and encapsulates the event demultiplexing and dispatching infrastructure of an event-driven application. Most applications need only one reactor instance, therefore it is typically implemented as a *Singleton* [GoF95]. To be able to integrate new or improved event handlers, a reactor offers a *Publisher-Subscriber* [POSA1] interface for registering and unregistering event handlers—together with the events these handlers ‘want’ to process and the handles associated with them. Different platforms usually require different reactor



implementations, thus separate the reactor interface from its implementations using a *Bridge* [GoF95]. *Strategized Locking* [POSA2] helps making these implementations thread-safe.

The reactor provides an event loop that uses a synchronous event demultiplexer, such as `select()`, to wait for events to occur on the handles of its registered event handlers. Calling the synchronous event demultiplexer blocks the reactor until one or more events arrive on the handles and it is possible to process these events without blocking. The synchronous event demultiplexer then returns all handles to the reactor on which events occurred. The reactor demultiplexes and dispatches these events serially to the handlers that own the 'ready' handles. These handlers then 'react' and process the events. This demultiplexing and dispatching mechanism is typically implemented as a *Manager* [PloPD2] that maintains a table of *<handle, event handler, event tuple>* entries.

## Proactor



A nice picture goes in here!

We are building an event-driven application. Or we are developing the asynchronous service layer in a *Half-Sync/Half-Async* (39) architecture or the event demultiplexing and dispatching infrastructure in a *Leader/Followers* (41) architecture.

\*\*\*

**An event-driven application typically receives multiple events from multiple sources. However, although an event indicates a request for a particular service, it does not itself specify the concrete service component and method that should process it. It is the application that must demultiplex incoming events and dispatch them to their corresponding service implementations.**

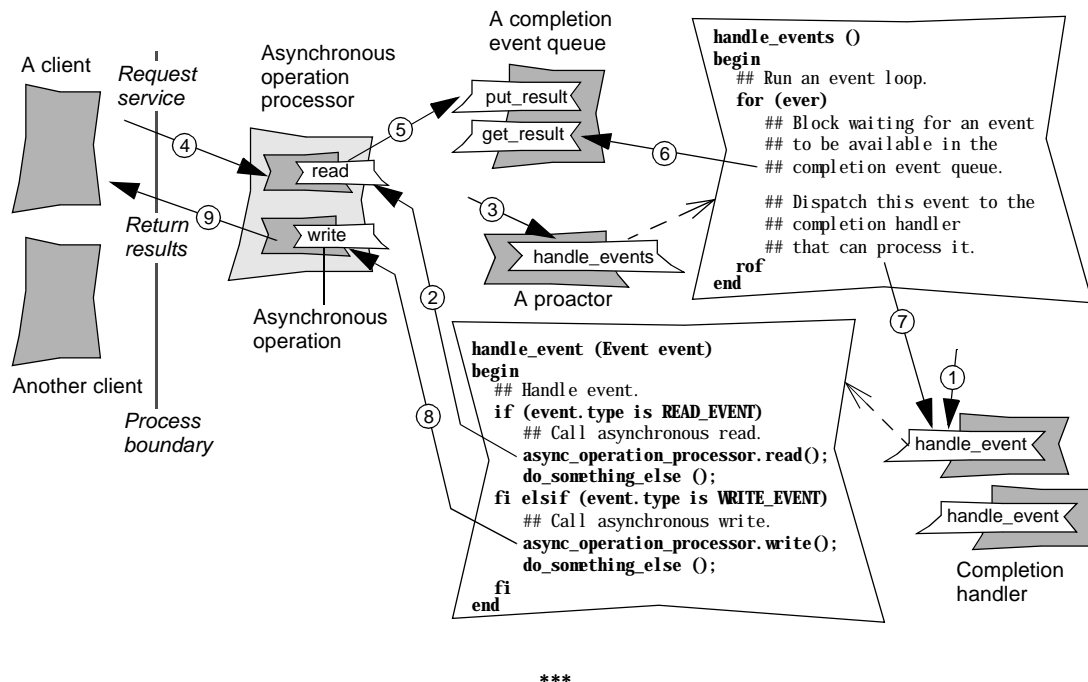
To maximize the performance and throughput of services, the application should arrange that multiple service request events are executed simultaneously. In particular, long-duration services should not delay other service processing unduly. Even multiple requests for the same service should execute in parallel. Using multi-threading to resolve this problem may be undesirable due to the overhead of context switching, synchronization, and data movement.

To support the evolution of an application, it should also be possible to integrate new or improved services into its event demultiplexing and dispatching infrastructure.

Therefore:

**Split application services into asynchronous operations that perform long-duration activities, in particular waiting for client request events and I/O, and completion handlers that use these asynchronous operations to implement application service logic. Provide an asynchronous operation processor to execute asynchronous operations without blocking their caller. This enables an application to process multiple requests simultaneously: every completion handler calls one or more asynchronous operations ‘proactively,’ and while these operations are executing, other handlers can service other client requests. Introduce a separate infrastructure to transfer the results of asynchronous**

operations back to their respective completion handlers: a completion event queue to buffer the completion events that asynchronous operations use to encapsulate their results and a proactor to demultiplex and dispatch these events serially to the handlers that process them. Start the application's proactive event processing mechanism by calling one or more completion handlers and the proactor.



The success of the *Proactor* pattern largely depends on a smart split of application services into asynchronous operations and completion handlers. Asynchronous operations should implement long-duration activities: awaiting events from remote clients, I/O, database access, processing-intensive algorithms, or operations that can block. Completion handlers, in contrast, should implement pure application service logic. If a completion handler must perform a long-duration activity, it calls the corresponding asynchronous operation instead of implementing this activity by itself.

The asynchronous operation processor executes asynchronous operations without blocking their caller. This enables an application to handle multiple requests simultaneously without using multi-threading. A handler that processes a particular client request will 'proactively' call an asynchronous operation at some point, for example to await a particular client event. While this operation is executing, another handler can process a second client request. Once the processing of this second request is also delegated to an asynchronous operation, yet another handler can process a third request; or the first handler can continue the processing of the first request if the corresponding asynchronous operation completed.

For efficiency reasons, base all communication between completion handlers and asynchronous operations on *Asynchronous Completion Tokens* (ACTs) (65). When a handler calls the asynchronous operation processor to execute a particular asynchronous operation, it also passes along its own identification. The asynchronous operation processor creates an ACT that serves as a placeholder for the results of the requested operation and configures this ACT with the completion handler identifier it received. Then the asynchronous operation processor invokes the asynchronous operation, passing along the ACT as an additional parameter. When the asynchronous operation completes, it fills the ACT with its results. The asynchronous operation processor then generates a completion event that contains this ACT and inserts this event into the completion event queue, which is observed by the proactor. The proactor removes the event and uses this event's ACT to demultiplex and dispatch the particular completion handler that called the asynchronous operation originally.

Implement completion handlers as *Template Methods* [GoF95]: an abstract class defines a uniform interface that represents the set of operations available to process completion events that are generated when asynchronous operations terminate. Concrete completion handlers derive from this abstract class, each implements the logic of a particular application service. This design provides an extensible event dispatching infrastructure: the proactor is independent of any completion handler logic, it only knows a polymorphic interface. Via *Adapter* [GoF95] subclasses we can also map the completion handler interface to existing application service *functions*.

The public completion handler interface often consists of a single method whose implementation within the concrete completion handlers dispatches all events that the handlers can process to appropriate private service methods. By this we can extend concrete completion handlers to handle new completion event types without changing their interface. Alternatively, we can provide a separate public event handling method for each event type. Such a multi-method interface avoids a handler-internal event dispatching and makes it easy to override event handling methods selectively for a particular concrete completion handler.

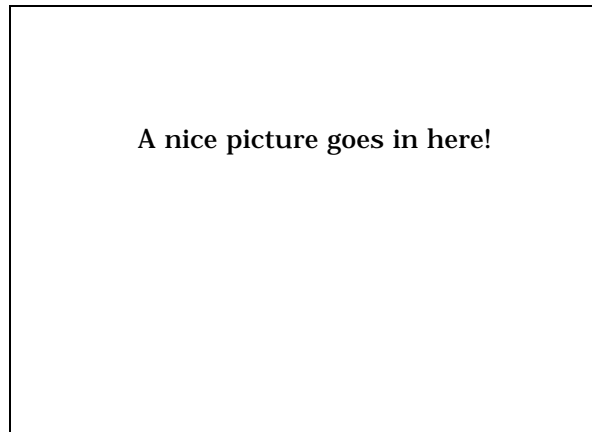
The *Acceptor-Connector* pattern (62) suggests to group completion handlers into three categories: service handlers implement application functionality; acceptors and connectors establish connections on behalf of these service handlers. This allows to vary connection establishment strategies independent of service processing strategies. Configure each completion handler such that it only processes completion events that arrive on a particular event source, for example, on a particular data-mode socket endpoint. Thus, every completion handler instance is uniquely identified by its event source—or better, the handle that represents this source. To simplify the use of handles within their completion handlers, wrap them into *Wrapper Facades* [POSA2].

Implementing the asynchronous operation processor is often straight forward: many operating systems offer efficient mechanisms for asynchronous operation execution as well as for completion event generation and queueing. Examples include Real-time POSIX and Windows 2000. In this case, introduce a *Wrapper Facade* [POSA2] that offers a portable asynchronous operation processor interface with an appropriate signature for each asynchronous operation. Make sure that all signatures include a parameter for identifying the completion handler that should process the results of the respective operations. Within the *Wrapper Facade* implementation, provide functionality for creating and mainting *Asynchronous Completion Tokens* and for associating the handles of completion handlers with the proactor's event demultiplexing mechanism. Map the signatures for asynchronous operations and the completion event generation and queueing functionality onto available operating system APIs.

The proactor provides and encapsulates the completion event demultiplexing and dispatching infrastructure of an event-driven application. Most applications need only one proactor instance, therefore it is typically implemented as a *Singleton* [GoF95]. If there are multiple proactors, the signatures of asynchronous operations within the asynchronous operation processor must also allow to specify the proactor that should dispatch this operation's completion event. To allow the proactor to demultiplex completion events to a particular completion handler, it must know the handle on which these events arrive. Therefore, a proactor offers a *Publisher-Subscriber* [POSA1] interface for registering the handle of a completion handler with its demultiplexing infrastructure. As described above, this registration is performed within the asynchronous operation processor—after a completion handler requested a particular asynchronous operation. Different platforms usually require different proactor implementations, thus separate the proactor interface from its implementations using a *Bridge* [GoF95]. *Strategized Locking* [POSA2] helps making these implementations thread-safe.

The proactor provides an event loop that uses an asynchronous event demultiplexer, such as `GetQueuedCompletionStatus()`, that blocks the proactor until completion events for registered handles are inserted into the completion event queue. The asynchronous event demultiplexer removes any such event from the queue and returns it to the proactor. The proactor then 'executes' the event's *Asynchronous Completion Token*, which automatically dispatches the completion event serially to the particular completion handler that is responsible for processing it. The handler then processes this event, often by 'proactively' initiating further asynchronous operations.

## Acceptor-Connector \*\*



We are implementing peer components in a distributed system, or the event handlers in a *Reactor* (55) architecture, or the completion handlers in a *Proactor* (58) architecture, or special-purpose connection metaobjects in a *Reflection* (30) architecture.

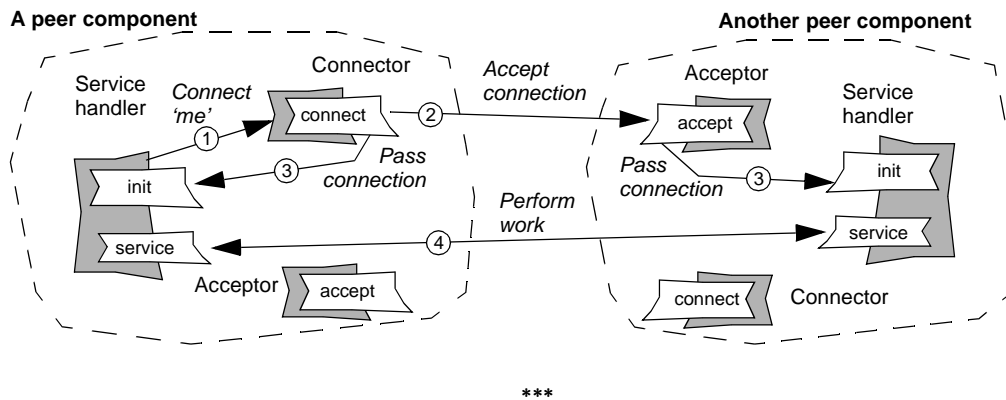
\*\*\*

**Peer components in a networked system execute their functionality in cooperation with other peer components to which they first must connect. However, the connection establishment code of a peer component is largely independent of the functionality that it performs once it is connected.**

To complicate matters, the application functionality of a peer component usually changes more frequently than its connection strategy. Yet a peer component may change its connection *role* dynamically: in one scenario it actively initiates a connection to a remote peer; in another scenario it passively accepts a connection request from a remote peer.

Therefore:

**Decouple the connection of peer components in a networked system from the processing these peers perform after they are connected. Encapsulate the application functionality of each peer component within a separate service handler. Connect peer service handlers using two factories: a connector factory actively initiates new connections to remote peers, an acceptor factory passively accepts connection requests from remote peers. Once the two factories established a connection between two peers, pass this connection to their associated service handlers so that these can execute their application functionality cooperatively.**



Decomposing peer components into service handlers, connectors, and acceptors allows these peers to vary their connection establishment functionality independent of their application functionality. In addition, the association of peer components with a connector *and* an acceptor allows them to vary their connection roles dependent on the needs of the application services they perform.

To provide an extensible connection establishment and service processing infrastructure, implement service handlers, connectors, and acceptors as *Template Methods* [GoF95]. An abstract service handler declares an interface for executing application services and for receiving an established connection. An abstract connector declares an interface for initiating connections actively and an abstract acceptor an interface for accepting connection requests passively. Let these abstract classes also provide the general processing skeletons for their respective operations. Within these skeletons, delegate the execution of peer-specific actions to corresponding hook methods.

Concrete service handlers typically derive from the abstract service handler because their hook method designs and implementations often widely differ—each concrete service handler implements the application functionality of a different peer component. Concrete connectors and acceptors, in contrast, usually differ only in very few aspects of their respective template methods: the IPC mechanism that they use to establish connections and the concrete service handlers on which behalf they establish these connections. Thus, concrete connectors and acceptors are often created by parameterizing the abstract connector and acceptor with IPC mechanism and service handler types, rather than by inheritance.

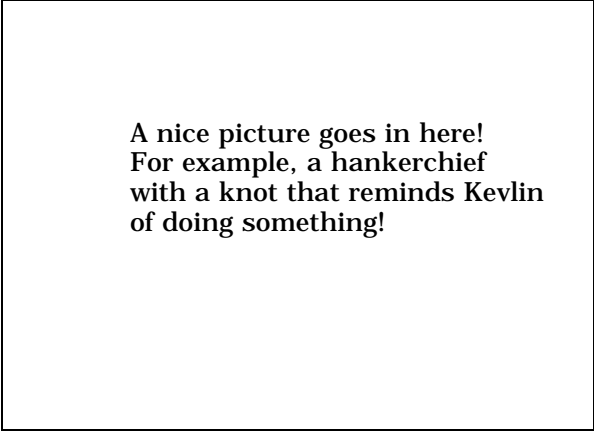
Each concrete service handler is a *Half-Object plus Protocol* [PLoPD1]: a half of a particular end-to-end service component in a networked system that performs its application functionality by exchanging data and messages with its peer service handler. Encapsulate the IPC mechanism used for this data and message exchange into a *Wrapper Facade* [POSA2]. This ensures its correct use within the peer service handlers. The performance of a networked application can often benefit from concurrent service handlers, which are implemented either as *Active Objects* (44) or as *Monitor Objects* (47). Service handlers that perform network I/O on behalf of particular application components are implemented as forwarders and receivers, according to the *Forwarder-Receiver* [POSA1] pattern.

Divide connectors into two methods. A connection initiation method is called by service handlers to establish a new connection actively. A connection completion method passes the established connection to the service handler that requested it. This decomposition into two methods allows connectors to support synchronous *and* asynchronous connection establishment transparently [POSA2]. As within service handlers, the IPC mechanisms used by connectors are typically encapsulated into *Wrapper Facades* [POSA2].

Let acceptors also offer two methods. A connection initialization method listens for the arrival of connection requests. If such a request arrives, a connection completion method accepts the connection request passively, establishes the connection, and passes the established connection to its associated service handler. The connection completion method is often implemented as a *Factory Method* [GoF95]. It creates a new concrete service handler instance whenever a new connection request arrives. This allows to handle all service requests from a particular remote peer via a designated service handler. As within service handlers and connectors, the IPC mechanisms used by acceptors are typically encapsulated into *Wrapper Facades* [POSA2].



## Asynchronous Completion Token \*\*



A nice picture goes in here!  
For example, a hankerchief  
with a knot that reminds Kevlin  
of doing something!

We invoke operations asynchronously. Or we are developing the event demultiplexing and dispatching infrastructure of a *Proactor* (58).

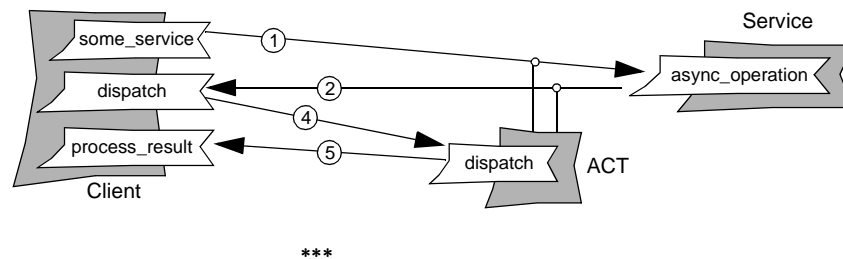
\*\*\*

**An operation that clients invoke asynchronously returns its response via a completion event. This response can then be demultiplexed, dispatched, and processed within the client. However, the client does not block after calling the operation. Thus, the state is has when the completion event arrives can differ from the state it had when the operation was called. Yet the client must process the result of the operation in its appropriate context in order to behave correctly.**

To not degrade performance, the client should also spend as little time as possible to identify how responses of asynchronous operations must be processed. In addition, if the client calls multiple asynchronous operations, the order in which responses arrive may not be identical to the order in which these operations were called.

Therefore:

**Along with each call that a client issues on an asynchronous operation, transmit information that identifies how the client should process the operation's response. This associates an individual invocation of an asynchronous operation with the specific behavior in the client to be executed when the operation finished. Hold the information—call it asynchronous completion token (ACT)—within the operation while it executes, but never modify it! Return the ACT to the client when the operation finishes, together with any results. The ACT thus unambiguously indicates to the client which behavior to execute in response to the finished operation. In particular, let the client use the ACT to demultiplex and dispatch control flow to an object or method that is responsible for processing the operation's result.**



Design ACTs as *Commands* [GoF95]. Just before the client invokes an asynchronous operation, create an ACT instance to represent this call and configure it with a particular handler inside the client that should process this invocation's result. Once the client received the completion event for this operation—which also contains its result and associated ACT—the client 'just' needs to execute the returned ACT to pass the result to its intended receiver and get it processed correctly. Thus, via ACTs the client can demultiplex and dispatch in constant  $O(1)$  time the results of *individual* asynchronous operation calls to the client-internal handlers that are responsible for processing these results—*independent* of its current state.

Keep the structure of ACTs opaque to asynchronous operations so they cannot modify the ACTs they receive. In particular, when calling an asynchronous operation, do not pass along the ACT itself, but an identifier for that ACT, such as a pointer to its local address, its object reference, or an index into a table. Keep the 'real' ACT within the client, for example, within a *Manager* [PloPD3]. When the ACT's identifier is returned, use it to retrieve the 'real' ACT.

Delete ACTs once the results of their corresponding asynchronous operations are processed. To be able to reclaim ACTs robustly even in case asynchronous operations fail, let an *Object Lifetime Manager* [PloPD4] control the lifetime of all ACTs within the client.

## References

[Bl91]

U.D. Black: *OSI: A Model for Computer Communications Standards*, Prentice Hall, 1991

[Cope98]

J.O. Coplien: *Multi-Paradigm Design with C++*, Addison-Wesley, 1998

[GoF95]

E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[JK00]

P. Jain, M. Kircher: *The Lookup Design Pattern*, Proceedings of the 5<sup>th</sup> European Conference on Pattern Languages of Programming, EuroPLoP 2000, July 2000

[Hen02]

K. Henney: *Disposal Method*, JavaSpektrum, July/August 2002, 2002

B. Johnson: *Frameworks = Patterns + Components*, Communications of the ACM, M. Fayad, D. C. Schmidt (eds.), vol. 40, no. 10, October 1997

[KLLM95]

G. Kiczales, R. DeLine, A. Lee, C. Maeda: *Open Implementation – Analysis and Design™ of Substrate Software*, Tutorial #21 of OOPSLA '95, October 1995

[Lea99]

D. Lea: *Concurrent Programming in Java, Design Principles and Patterns*, 2<sup>nd</sup> edition, Addison-Wesley, 1999

[Lew95]

B. Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, 1995

[PLoPD1]

J.O. Coplien, D.C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)

[PLoPD2]

J.O. Coplien, N. Kerth, J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)

[PLoPD3]

R.C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA '96, San Jose, California, USA, 1996)

[PLoPD4]

N. Harrison, B. Foote, H. Rohnert (eds.): *Pattern Languages of Program Design 4*, Addison-Wesley, 1999 (a book publishing selected papers from the Fourth and Fifth International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1997 and 1998, and the Second and Third European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1997 and 1998)

[POSA1]

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture—A System of Patterns*, John Wiley and Sons, 1996

[POSA2]

D.C Schmidt, M. Stal, H. Rohnert, F. Buschmann.: *Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000

[Sol98]

D.A. Solomon: *Inside Windows NT*, 2<sup>nd</sup> edition, Microsoft Press, 1998

[Ste98]

W.R. Stevens: *Unix Network Programming, Volume 1: Networking APIs: Sockets and XTI*, 2<sup>nd</sup> edition, Prentice Hall, 1998

[Ste99]

W.R. Stevens: *Unix Network Programming, Volume 2: Interprocess Communications*, 2<sup>nd</sup> edition, Prentice Hall, 1999

[Sun88]

Sun Microsystems: *Remote Procedure Call Protocol Specification*, Sun Microsystems, Inc., RFC-1057, June 1988

[Tan92]

A.S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, 1992

[VSW02]

M. Völter, A. Schmid, E. Wolff: *Server Component Patterns — Component Infrastructures illustrated with EJB*, John Wiley and Sons, 2002