Table of Contents

III The Language

"I wish life was not so short," he thought,
"Languages take such a time,
and so do all the things one wants to know about."

J.R.R. Tolkien, The Lost Road

In the third part of the book we present one possible pattern language for distributed computing. We distilled it from our own experiences in realizing distributed systems, as well as from the distribution patterns that skillful software architects, designers, and developers contributed to the software community. The language has been used to develop many real-world distributed object computing middleware and distributed applications. You can use it with your colleagues and project team-mates to guide the design of new distributed systems, and also to improve and refactor existing ones.

Over the past fifteen years we have participated in the development of many industrial networked, concurrent, and distributed systems, ranging from industrial process automation systems, medical imaging, and large-scale telecommunication systems, to high-performance communication middleware. The pattern language for distributed computing that we present in this part of the book distils this experience in a tangible, ready-to-use form. You can use it to build new distributed systems, to evolve, re-engineer, or refactor existing systems, or simply to understand the architectures of distributed software systems or middleware that you are using in your work.

Our pattern language for distributed computing includes 114 patterns, which are grouped into thirteen problem areas. A problem area addresses a specific technical topic related to building distributed systems, and comprises all those patterns in our language that address the challenges associated with that technical topic. The main intent of the problem areas is to make the language and its patterns more tangible and comprehensible: patterns that address related problems are presented and discussed within a common and clearly scoped context. The problem areas are presented in their approximate order of relevance and application when building distributed systems.

Each problem area and its constituent patterns forms a separate chapter in this part of the book:

- Chapter 9, *From Mud To Structure*, includes the ten root patterns of our pattern language for distributed computing. They help transform the 'mud' of requirements and constraints we usually start with into a coarse-grained software structure with clearly separated, tangible parts that make up the system being developed.

- Chapter 10, *Distribution Infrastructure*, describes twelve patterns pertaining to *middleware*, distribution infrastructure software that helps to simplify distributed computing applications.

- Chapter 11, *Event Demultiplexing and Dispatching*, comprises four patterns that provide efficient and flexible infrastructures for demultiplexing, dispatching, and responding to events received from the network.

- Chapter 12, *Interface Partitioning*, offers eleven patterns that help in the design and specification of meaningful component interfaces

that are easy to use for common component usage scenarios, but also allow for special-purpose and out-of-band scenarios.

- Chapter 13, *Component Partitioning*, includes six patterns for partitioning components. The focus of the patterns is on supporting visible component quality properties such as performance, scalability, and flexibility.
- Chapter 14, *Application Control*, addresses eight patterns that help in transforming user input for an application into concrete service requests to its functionality, executing those requests, and transforming any results back into an output meaningful for users—which can be a challenging task.
- Chapter 15, *Concurrency*, comprises four patterns for concurrency that help servers and server-side software to handle requests from multiple clients simultaneously.
- Chapter 16, *Synchronization*, describes nine patterns that help with synchronizing the access to shared components, objects, and resources, either by outlining efficient synchronization strategies, or by minimizing the need for synchronization.
- Chapter 17, *Object Interaction*, comprises eight patterns that support efficient collaboration and data exchange between interacting components and objects of an application.
- Chapter 18, *Adaptation and Extension*. describes thirteen patterns that help in preparing components and objects in long-lived systems, in particular distributed systems, for their own configuration, adaptation, and evolution.
- Chapter 19, *Modal Behavior*, offers three patterns for structuring components and objects that are inherently state-driven.
- Chapter 20, *Resource Management*. includes twenty-one patterns that help with explicit management of components and resources in a distributed system.
- Chapter 21, *Database Access*, 'closes' our pattern language by presenting five patterns for mapping an object-oriented application design to a relational database schema efficiently, *and* without introducing tight dependencies between the two worlds.

The main intent of our pattern language for distributed computing is to serve as an overview about, introduction to, guide through, and communication vehicle for the best practices and state-of-the-art in major areas of the construction of distributed software systems. It is not a tutorial for distributed computing in general, however, but has a clear focus on the *design* of distributed software systems. We therefore assume readers have some familiarity with core distributed computing concepts and mechanisms, as described in the body of the relevant literature [TaSte02] [Bir05].

9 From Mud To Structure

Kasbah Ait Benhaddou, Atlas Mountains, UNESCO world cultural heritage

This chapter presents the root and entry point to our pattern language for distributed computing. Its featured patterns help to transform the mud of requirements and constraints we usually start with into a coarse-grained software structure with clearly separated, tangible parts that make up the system being developed, and address several key concerns of sustainable software architectures: operational aspects such as performance and availability, as well as developmental qualities like extensibility and maintainability.

Large distributed systems tend to be complex. In the beginning, all we have is a set of requirements and constraints that must be transformed into a working software system. A naive approach to development is likely to result in a 'big ball of mud' [FoYo99], a software clump whose design and code is so messy that it is hard to see any coherent architecture in it. Such software is hard to understand, maintain, and evolve, and over time it also tends to suffer from poor stability, performance, scalability, and other essential operational architecture qualities [Bus03].

One of the keys to successful software development is *structure*. We need structure that can be understood by developers, structure that is resilient to the forces to which the system and its development are subjected, structure that favors the development process surrounding and creating it, structure that respects the business and individuals who will make it and shape it. In short, structure that provides a habitable environment for developers and other stakeholders of a software system. Without vision and a guiding hand, however, the structure of a software system is likely to be complicated rather than just complex, leading not only to the loss of the big picture, but the small picture as well—the code can become mired in accidental detail and assumptions.

In undertaking such software development, therefore, a coarse-grained conception of the system is needed that—with the help of abstraction and separation—omits unnecessary details and organizes the system's key concepts at a broader level.

First and foremost, a software architecture must be a meaningful expression of the system's application domain. Specifically, the functionality and features provided by the system must support a concrete business, otherwise it has no practical value for its users. If the system's software architecture does not scope and portray the application domain appropriately, however, it will be hard, if not impossible, to provide user-level services and features that correctly address the functional requirements of the system.

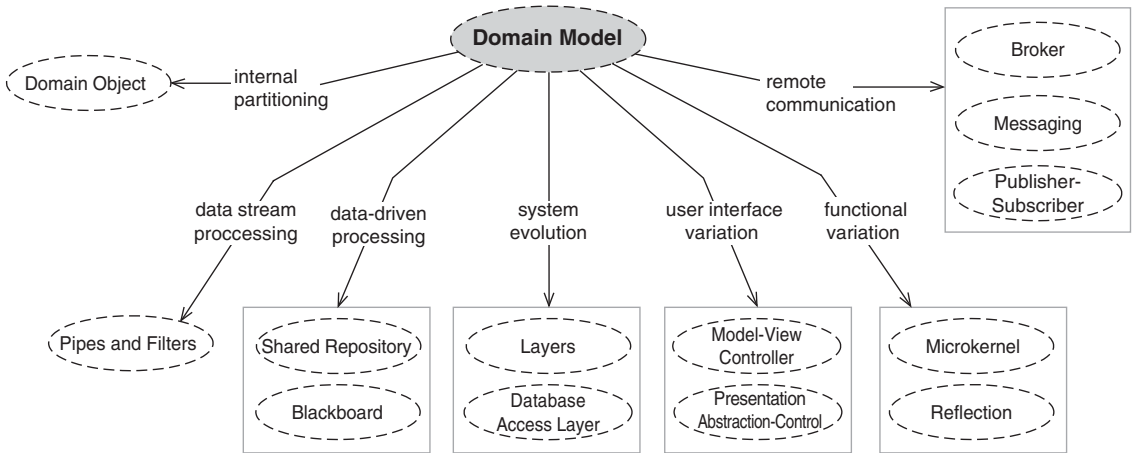
A further concern when modeling the functional architecture of a system is *variability*. Variations can arise in regard to different feature sets, alternatives in business processes, choices for concrete business algorithms, and options for the system's appearance to the user. Without a clear knowledge of what can vary in an application domain,

and also of what variations must be supported, it is hard to provide the right level and degree of flexibility in a software system or product.

The root pattern of our pattern language addresses the challenge of creating a model of the application domain that both reflects the functional responsibilities of a software system and can serve as a solid basis for the further elaboration of its technical architecture.

> The DOMAIN MODEL pattern (182) [Fow03a] defines a precise model for the structure and workflow of an application domain—including their variations. Model elements are abstractions meaningful in the application domain; their roles and interactions reflect domain workflow and map to system requirements.

The following diagram illustrates how DOMAIN MODEL connects to the body of our pattern language for distributed computing and orchestrates the patterns that help with its refinement.



DOMAIN OBJECT is also described in the collection of *Patterns Of Enterprise Application Architecture* [Fow03a], but its focus there is solely on technical aspects of the realization of a domain model, and it does not address the explicit modeling of the domain.

Dividing the core structure of a distributed software system purely along lines visible in the application domain, however, will not always help to define a feasible baseline architecture. On one hand, a software system needs to include many components and exhibit

many properties that are unrelated to its domain. For example, quality-of-service requirements, such as performance and predictable resource utilization, are cross-cutting issues and therefore cannot be addressed through component decomposition alone. Similarly, the need for responsive user interaction can conflict with the latency and partial-failure modes associated with networks. On other hand, we as developers want more than a system that simply meets the visible user requirements. User's indifference to developmental qualities such as portability, maintainability, comprehensibility, extensibility, testability, and so on should not be shared by developers.

Finding a suitable application partitioning depends on framing answers to several key questions and challenges:

- *How does the application interact with its environment*? Some systems interact with different types of human user, others with other systems as peers, and yet others are embedded within even more complex systems. Inevitably, there are also systems that have all of these interactions.
- *How is application processing organized*? Some applications receive requests from clients to which they react and respond. Other applications process streams of data. Some applications perform self-contained tasks without receiving stimuli from their environment. Indeed, for some applications, it may not even be possible to identify any concrete workflow and explicit cooperation among its components.
- *What variations must the application support*? Flexibility is a major concern in software development, especially when developing software products, or software product families, that are intended to serve a whole range of different customer needs. Some systems must support different feature sets, such as for small, medium, and large enterprises, to address different markets and customer groups. Other systems must support variations in business processes, so that each customer can model the workflow of its specific business appropriately. Yet other systems must support variations in algorithmic behavior and visual appearance to be attractive to a broad range of customers.

- *What is the life expectancy of the application*? Some systems are short-lived and thrown away when they are no longer used, such as an online trading program designed to exploit a transient market trend. Other systems will be in operation for thirty years or more and must respond to changing requirements, environments, and configurations, such as Telecommunication Management Network (TMN) system.

Our pattern language for distributed computing, therefore, includes nine strategic patterns that help in the transformation of a DOMAIN MODEL into a technical software architecture that can serve as the basis for further development. Each pattern provides its own answers to the questions raised above:

> The LAYERS pattern (185) [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, granularity, hardware-distance, or other partitioning criteria.
>
> The MODEL-VIEW-CONTROLLER pattern (MVC) (188) [POSA1] [Fow03a] divides an interactive application into three parts. 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 pattern (PAC) (191) [POSA1] defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Each 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 an agent from its functional core and its communication with other agents.
>
> The MICROKERNEL pattern (194) [POSA1] applies to software systems that must 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 REFLECTION pattern (197) [POSA1] provides a mechanism for changing the 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 base level includes the core application logic. Its runtime behavior is observed by a meta level that maintains information about selected system properties to make the software self-aware. Changes to information kept in the meta level thus affect subsequent base-level behavior.

The PPIPES AND FILTERS pattern (200) [POSA1] [HoWo03] provides a structure for systems that process data streams. Each processing step is encapsulated in a filter component. Pipes are used to pass data between adjacent filters.

The SHARED REPOSITORY pattern (202) [HoWo03] helps to structure applications whose functionality and collaboration is purely data-driven. A shared repository maintains the common data on which the application's components operate, the components themselves access and modify the data in the shared repository, and the state of that data in the shared repository instigates the control flow of specific components.

The BLACKBOARD pattern (205) [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.

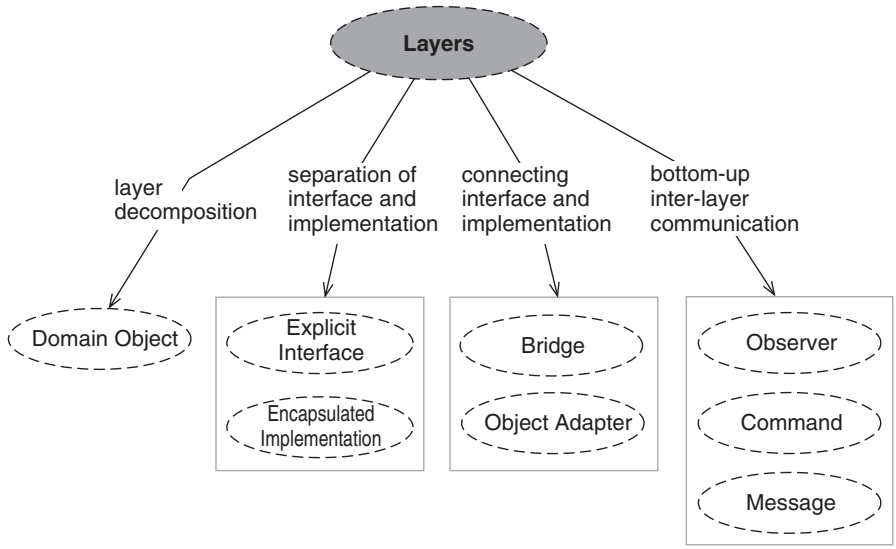
The DOMAIN OBJECT (208) pattern encapsulates a self-contained, coherent functional or infrastructural responsibility into a well-defined entity that offers its functionality via one or more explicit interfaces while hiding its inner structure and implementation.

The nine patterns above address a whole range of different concerns in refining DOMAIN MODEL towards a sustainable software baseline architecture.

LAYERS defines a general approach for partitioning the responsibilities of an application according to a (sub) system-wide property, such that each group of functionalities can be developed and evolved independently. The specific partitioning criteria can be defined along one or more dimensions, such as abstraction, granularity, hardware distance, and rate of change. LAYERS is probably the most fundamental

pattern for separating different, and grouping related, concerns in a software architecture.

LAYERS integrates with our pattern language for distributed computing as follows.

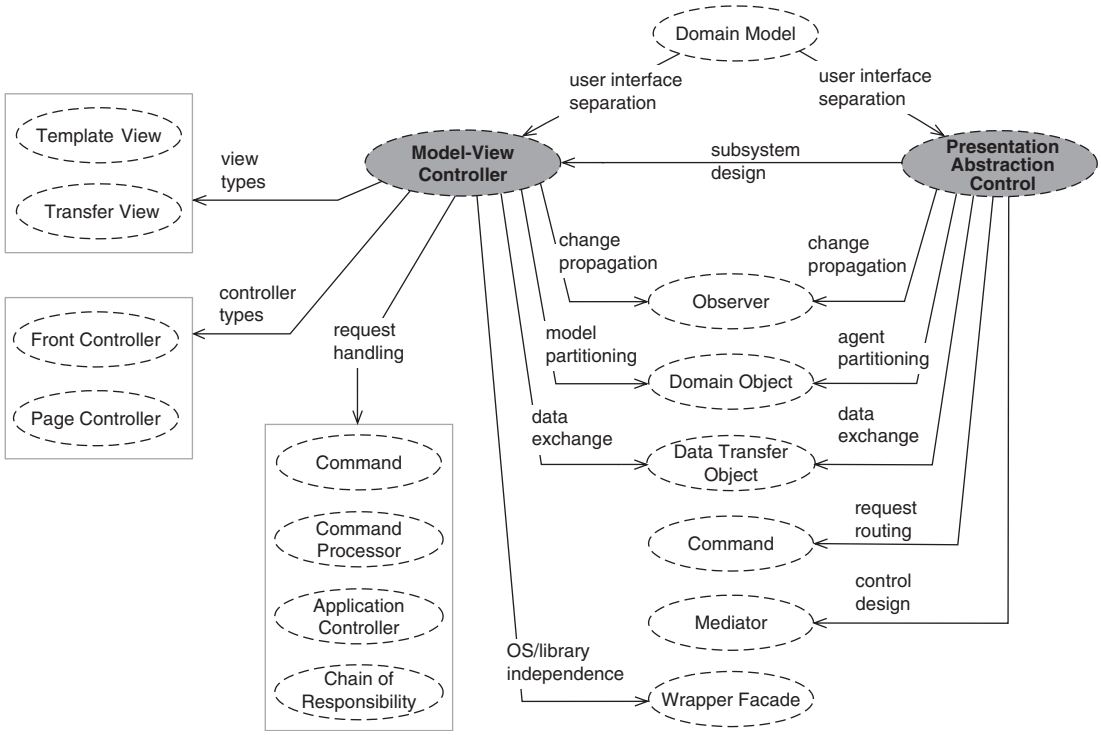


The next two patterns, MODEL-VIEW-CONTROLLER and PRESENTATION-ABSTRACTION-CONTROL, address the problem of support for variability in user interfaces. Though both patterns are related in several ways, they are not necessarily alternatives. In a nutshell, MODEL-VIEW-CONTROLLER supports variability within one specific user interface, while PRESENTATION-ABSTRACTION-CONTROL supports the use of multiple, distinct user interfaces and their independent variation. As most software systems need only one user interface paradigm, MODEL-VIEW-CONTROLLER should always be your first choice.

PRESENTATION-ABSTRACTION-CONTROL, in contrast, is (only) useful if a software system is partitioned into multiple, largely independent but sometimes cooperating subsystems, each of which suggests its own user interface paradigm. Examples for such systems include software that is used off-shore by users both on board a ship and underwater, robot control systems, or applications that are partly operated via virtual reality devices. Only few types of system fall into this

category, thus the applicability of PRESENTATION-ABSTRACTION-CONTROL is significantly narrower than that of MODEL-VIEW-VIEW-CONTROLLER.

The following diagram illustrates how these two patterns integrate with our pattern language for distributed computing.

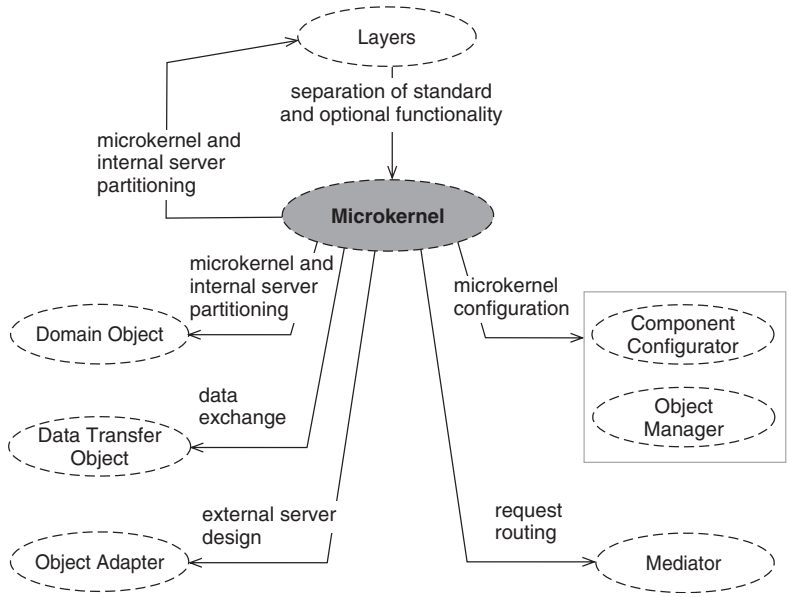


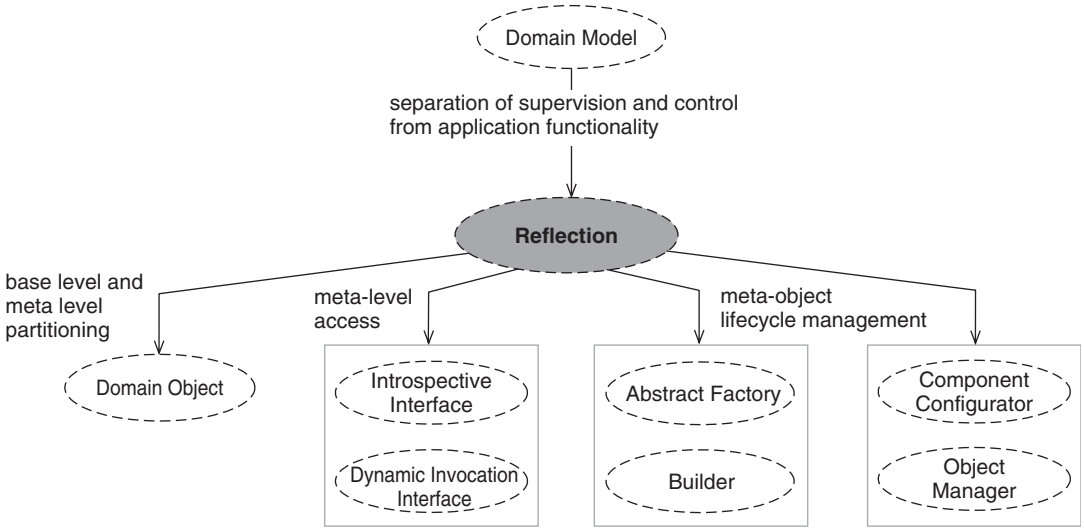
The MODEL-VIEW-CONTROLLER pattern is also described in *Patterns of Enterprise Application Architecture* [Fow03a], where it has the same intent, structure, and behavior as the POSA version. The MODEL-VIEW-PRESENTER pattern [Fow06] is also related to MODEL-VIEW-CONTROLLER, but works better for rich client development than MVC because it does not delegate all view behavior to the model. An intermediate presenter component receives all user actions, such as when clicking a checkbox, or if the involvement of the model is necessary, such as when clicking an 'Apply' button, and decides whether it can handle them without consulting the model. This improves composability of complex views and the testability of the different roles.

The next two patterns, MICROKERNEL and REFLECTION, both foster the construction of flexible software systems. Both patterns address different aspects of flexibility, however. MICROKERNEL, in general, provides a plug-in architecture that supports flexibility in terms of *what* functionality a system provides to its users. MICROKERNEL has thus evolved as a popular architecture for operating systems, middleware, and product-line architectures.

REFLECTION, in contrast, defines an architecture that objectifies specific aspects of a system's structure and behavior, which supports flexibility in terms of *how* its functionality executes and/or can be used by its clients. It is thus often used in the context of application and service integration scenarios, in which client applications must be able to use or control the functionality of other applications without having an explicit, built-in knowledge of their interfaces and internal behavior.

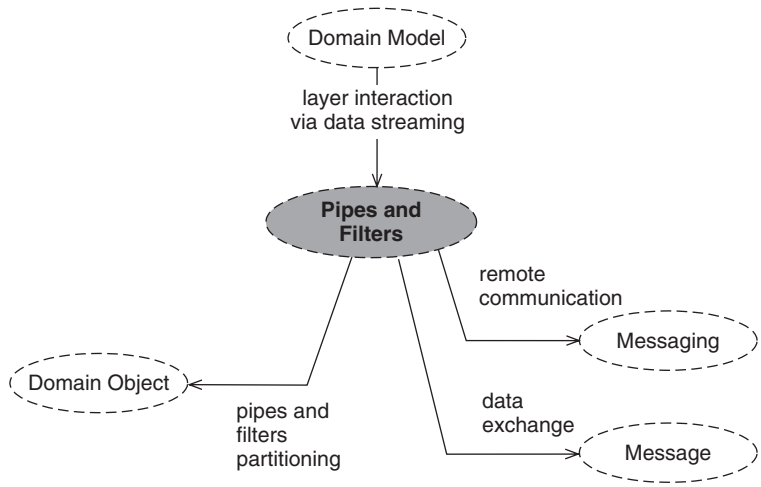
The following diagrams outline how MICROKERNEL and REFLECTION connect to the patterns of our language.





The PIPES AND FILTERS pattern is suited for applications that process data streams, or whose components communicate via the exchange of data streams. Image processing is a prime example of a domain that can best be modeled in software via a PIPES AND FILTERS architecture.

The integration of PIPES AND FILTERS into our pattern language is shown in the diagram below.

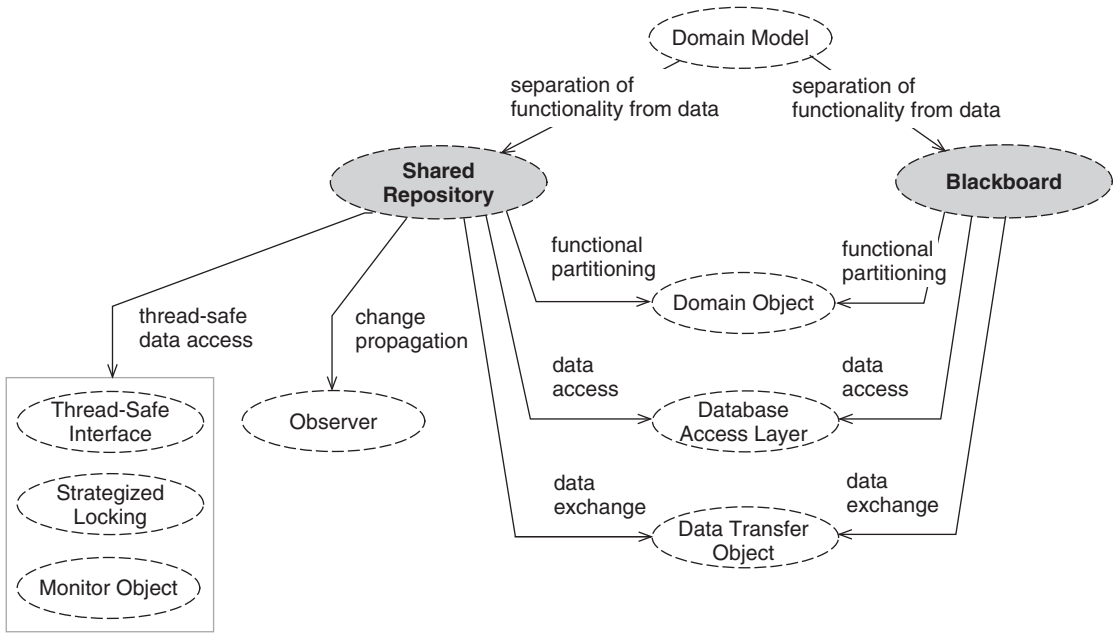


Some sources also see PIPES AND FILTERS as a fundamental style for interprocess communication [HoWo03] [VKZ04], but in the context of our pattern language for distributed computing we consider it more as an approach for *orchestrating the collaboration* of an application's services, rather than a style they can use to *exchange information*. In a PIPES AND FILTERS arrangement the latter is done with help of messaging, which, as we outline in Chapter 10, *Distribution Infrastructure*, we consider of equal importance to remote method invocation and publish/subscribe as one of three communication styles.

Another version of PIPES AND FILTERS is published in *Enterprise Integration Patterns* [HoWo03], where it is used to enable stepwise transformation of message formats and content in message-oriented middleware. The scope of PIPES AND FILTERS in POSA is broader, because there the pattern is used to structure entire data stream processing *applications*. The use of PIPES AND FILTERS in this pattern language follows the POSA scope.

The SHARED REPOSITORY and BLACKBOARD patterns help in designing applications whose components work largely on a common set of (structured) data. By separating the data from the functionality of a system, data exchange between components of the application is simplified, and coordination of the components via Change of Value (CoV) notifications in that data becomes possible.

Their integration with our pattern language for distributed computing is as follows:



The difference between the two patterns is in their computational approach. In a SHARED REPOSITORY architecture, application components realize a deterministic control flow and cooperate in an explicitly coded or configured manner. It is thus a suitable approach for applications in the network management and control system domains, which typically operate on a large amount of data from field devices, such as Telecommunication Management Network (TMN) systems or industrial process control systems.

A BLACKBOARD architecture, in contrast, implements a computational model based on heuristics, which is able to produce useful results even when no deterministic algorithms are known or feasible in an application domain, or if input data is fuzzy, inaccurate, or otherwise questionable in its quality. For example, BLACKBOARD is a fairly popular architectural approach for bio-information systems, which typically operate on large bases of fuzzy, incomplete, or partly erroneous

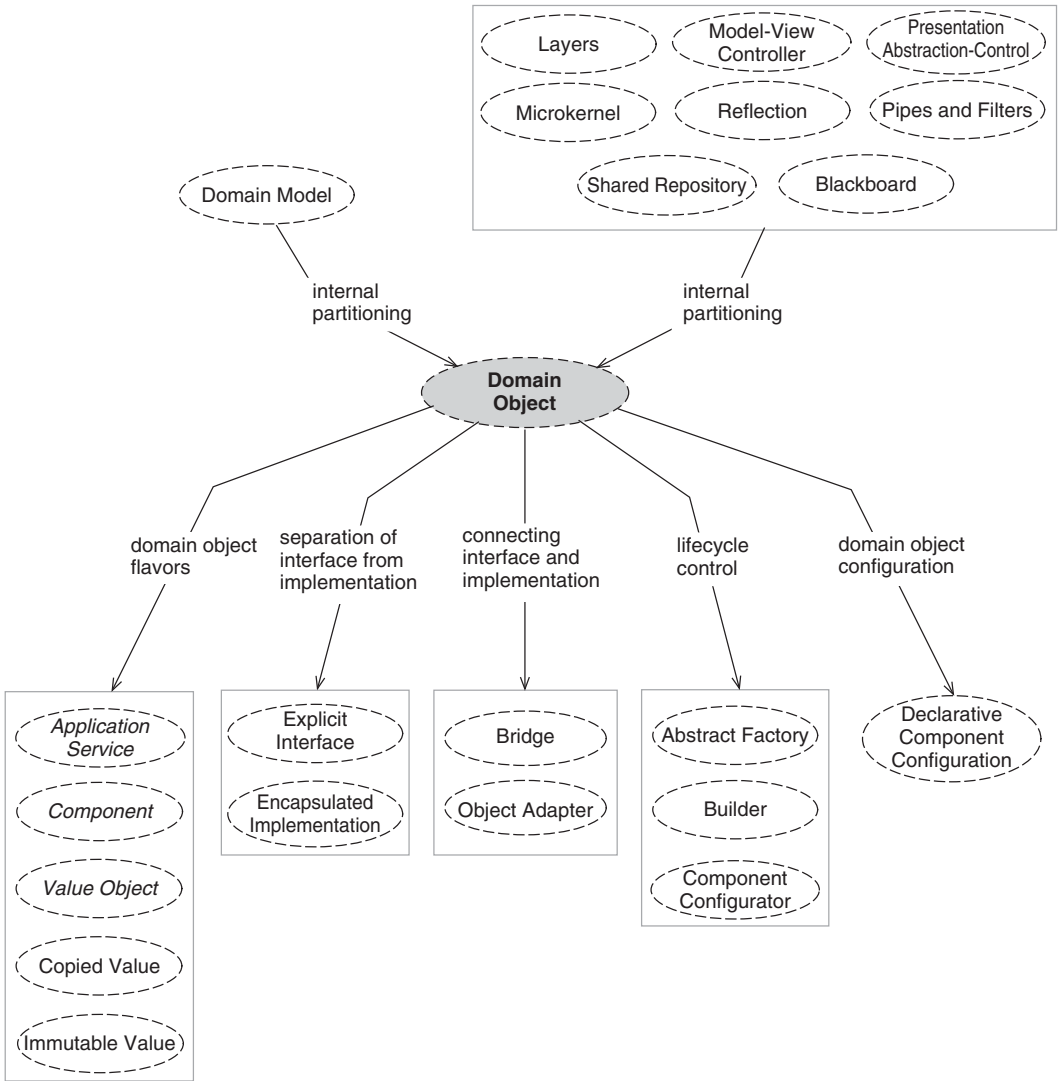
data. It was also popular in speech recognition applications until appropriate deterministic solutions were discovered.

From a general perspective, BLACKBOARD is a specialized variant of SHARED REPOSITORY, but one that addresses a different set of forces whose resolution requires a different computational approach. Although the applicability of BLACKBOARD is definitely narrower than that of SHARED REPOSITORY, these differences justify its description as a pattern in its own right.

The SHARED REPOSITORY pattern is also described in *Enterprise Integration Patterns* [HoWo03], under the name SHARED DATABASE. Its focus there is on (enterprise) application integration—in contrast to coordinating the control flow of a data-driven applications, which is the scope of the pattern in the context of our pattern language. As with PIPES AND FILTERS, some sources also see SHARED REPOSITORY as a fundamental style for interprocess communication [Fow03a] [HoWo03] [VKZ04]. We do not share this perspective in the context of our pattern language for distributed computing. and consider the pattern as an approach to partitioning the functionality of an application.

The final pattern we present in this chapter, DOMAIN OBJECT, supports the encapsulation of self-contained responsibilities in an application within a defined software realization. Such encapsulation allows us to address the specific functional, operational, and developmental requirements of this responsibility explicitly, directly, and independently of other DOMAIN OBJECT realizations.

The next diagram illustrates the integration of DOMAIN OBJECT into our pattern language for distributed computing.



Some of the patterns referenced are not described in this book, but are nevertheless included in our pattern language because they represent different DOMAIN OBJECT flavors. APPLICATION SERVICE is the key pattern in *Core J2EE Patterns* for partitioning the business logic

of an enterprise application [ACM01], and COMPONENT is one of the two root patterns of *Server Component Patterns* [VSW02]. The two patterns, therefore, connect our pattern language to all patterns in *Server Component Patterns* and *Core J2EE Patterns* of which they are composed. A VALUE OBJECT, finally, is a small object whose identity is based on its state rather than its type [PPR] [Fow03a]

Most real-world software systems cannot be formed reasonably from a single pattern from the above application partitioning patterns. Different patterns offer different architectural properties, and 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, such as adaptability of its user interface *and* portability to multiple platforms. For such systems you must combine several patterns to form an appropriate structure. When scaling to a distributed environment, these application infrastructure patterns must also be integrated with suitable distribution patterns.

However, the selection of an application partitioning pattern, or a combination of several, is just one step of many when designing a software system. A selection of partitioning patterns is not yet a complete software architecture in the sense of being whole and addressing all the significant decisions that characterize the design. Instead it remains a structural framework for a software system that must be further specified and refined. This process includes the task of expressing the application's concrete functionality within the framework, detailing its components and relationships. We support this with the patterns presented in the subsequent chapters of our pattern language.

Domain Model **

When starting to build a (distributed) application ...

... we need an initial structure for the software being developed.

♦♦♦

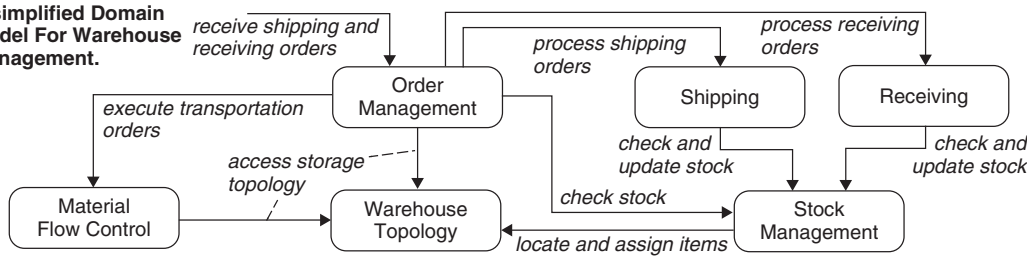
Requirements and constraints inform the functionality, quality of service, and deployment aspects of a software system, but do not themselves suggest a concrete structure to guide development. Without precise and reasoned insight into a system's scope and application domain, however, its realization is likely to be a 'big ball of mud' that is hard to understand and convey to customers, and a poor architectural basis on which to build.

A list of requirements shows the problem domain of an application, but not its solution domain. Yet requirements in a working system must be addressed by concrete software entities. If these entities and their interactions are unrelated to the application's core business, however, it will be hard to understand and communicate what the system actually does. Similarly, it will be hard to meet system quality of service requirements, since they cannot be mapped clearly to the software elements where they are relevant. Without a clear vision of a system's application domain, therefore, software architects cannot determine if their designs are correct, complete, coherent, and sufficiently bounded to serve as the basis for development.

Therefore:

Create a model that defines and scopes a system's business responsibilities and their variations: model elements are abstractions meaningful in the application domain, while their roles and interactions reflect the domain workflow.

A simplified Domain Model For Warehouse Management.



The domain model serves as the foundation for a system's software architecture, which becomes an expression of the model as it evolves.

♦♦♦

A DOMAIN MODEL provides the initial step in transforming requirements into a sustainable software architecture and implementation. Defining a precise model for the structure and workflow of an application domain, including their variations, helps to map requirements to concrete software entities and check whether requirements are complete and self-consistent. Missing requirements can be identified, fuzzy requirements clarified, and unnecessary requirements removed. As a result, the responsibilities and boundaries of a system's architecture are scoped properly. A well-formed domain model also makes it easier meet a system's quality of service requirements, because they can be assigned to the specific elements and workflows to which they apply in the model. A domain model also fosters communication between software professionals, domain experts, and customers, because its elements are based on the terminology used in the application domain.

In general, a domain model is created using an appropriate method, such as *Domain-Driven Design* [Evans03] or *Domain Analysis* [CLF93]. Several specialized methods support the expression of variabilities in a domain, such as *Commonality/Variability Analysis* [Cope98] and *Feature Modeling* [CzEi02]. Domain-specific patterns can further support the creation of a domain model: they offer representations for recurring arrangements of common abstractions and workflows within a domain, including their potential variations. Domain-specific patterns are documented in a variety of domains, such as telecommunication, health care, and corporate finance [Fow97] [Ris01] [PLoPD1] [PLoPD2] [PLoPD3] [PLoPD4] [PLoPD5].

Once the domain model has matured to the point where it adequately portrays the functional responsibilities of an application, as well as their variations, the next step is to transform the model into a concrete architecture that expresses and supports this functionality, and which addresses a range of quality of service requirements such as performance, scalability, availability, adaptability, and extensibility.

Several patterns help to arrange and connect the elements of a domain model to support specific styles of computation. For example,

Pipes and Filters (200) is suitable for applications that process data streams, Shared Repository (202) helps to organize data-driven applications, and Blackboard (205) is appropriate for applications that operate on incomplete or fuzzy data, or for which no deterministic solution algorithm is known or feasible.

Other patterns help group and separate elements of a domain model to support specific aspects of system adaptation, extension, and evolution. For example, Layers (185) groups elements of the domain model that share similar responsibilities, properties, or granularity into separate layers, so that each layer can evolve independently. Model-View-Controller (188) and Presentation-Abstraction-Control (191) separate user interfaces from domain functionality, to support customer-specific interface adaptations without the need to change or modify the realization of business logic. Microkernel (194) partitions applications into core functionality, version-specific functionality, and version-specific APIs, to support different product variants. Reflection (197) objectifies specific aspects of a system's structure and behavior to supports runtime flexibility in terms of how its functionality executes and/or can be used by its clients. Finally, Database Access Layer (538) decouples application functionality from a relational database, to make it easy to replace the database.

In production systems, several of the patterns outlined above can be applied in combination to form a structural baseline architecture for an application. For example, a Model-View-Controller arrangement may be combined with Reflection and a Shared Repository-based computational model.

Typically, each self-contained and coherent entity or responsibility within the application's baseline is represented as a separate Domain Object, to provide a defined software realization that addresses its specific functional, operational, and developmental requirements.

In a distributed system, the domain objects in an application's baseline can communicate via middleware. For example, Broker (237) supports applications whose components communicate via remote method invocation, Messaging (221) supports systems in which components exchange asynchronous messages, and Publisher-Subscriber (234) mediates communication between components that coordinate their processing via notifications of changes to their state.

Layers **

When transforming a DOMAIN MODEL (182) into a technical software architecture, or when realizing BROKER (237), DATABASE ACCESS LAYER (438), MICROKERNEL (194), or HALF-SYNC/HALF-ASYNC (359) ...

... we must support the independent development and evolution of different system parts.

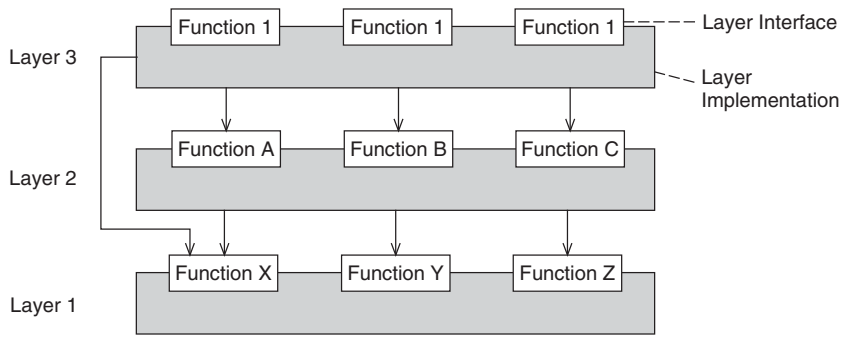
♦♦♦

Regardless of the interactions and coupling between different parts of a software system, there is a need to develop and evolve them independently, for example due to system size and time-to-market requirements. However, without a clear and reasoned separation of concerns in the system's software architecture, the interactions between the parts cannot be supported appropriately, nor can their independent development.

The challenge is to find a balance between a design that partitions the application into meaningful, tangible parts that can be developed and deployed independently, but does not lose itself in a myriad of detail so that the architecture vision is lost and operational issues such as performance and scalability are not addressed appropriately. An ad hoc, monolithic design is not a feasible way to resolve the challenge. Although it allows quality of service aspects to be addressed more directly, it is likely to result in a spaghetti structure that degrades developmental qualities such as comprehensibility and maintainability.

Therefore:

Define one or more layers for the software under development, with each layer having a distinct and specific responsibility.



Assign the functionality of the system to the respective layers, and let the functionality of a particular layer only build on the functionality offered by the same or lower layers. Provide all layers with an interface that is separate from their implementation, and within each layer program using these interfaces only when accessing other layers.

♦♦♦

A LAYERS architecture defines a horizontal partitioning of a software's functionality according to a (sub)system-wide property, such that each group of functionalities is clearly encapsulated and can evolve independently. The specific partitioning criteria can be defined along various dimensions, such as abstraction, granularity, hardware distance, and rate of change. For example, a layering that partitions an architecture into presentation, application logic, and persistent data follows the abstraction dimension. A layering that introduces a business object layer whose entities are used by a business process layer follows the granularity dimension, while one that suggests an operating system abstraction layer, a communication protocol layer, and a layer with application functionality follows the hardware distance dimension. Using rate of change as a layering criteria separates functionalities that evolve independently of one another.

In most applications we find multiple dimensions combined. For example, decomposing an application into presentation, application logic, and persistent data layers is a layering according to both levels of abstraction and rate of change. User interfaces tend to change at a higher rate than application logic, which evolves faster than data schemes such as tables in a relational database. Regardless of which layering dimensions an application follows, each layer uses the functionality offered by lower layers to realize its own functionality.

A key challenge is to find the 'right' number of layers. Too few layers may not separate sufficiently the different issues in the system that can evolve independently. Conversely, too many layers can fragment a software architecture into bits and pieces without a clear vision and scope, which makes it hard to evolve them at all. In addition, the more layers are defined, the more levels of indirection must cross in an end-to-end control flow, which can introduce performance penalties—especially when layers are remote.

Typically, each self-contained and coherent responsibility within a layer is realized as a separate Domain Object, to further partition the layer into tangible parts that can be developed and evolved independently.

Split each layer into an Explicit Interface (281) that publishes the interfaces of those domain objects whose functionality should be accessible by other layers, and connect it with an Encapsulated Implementation (313) that realizes this functionality. This separation of concerns minimizes inter-layer coupling: each layer only depends on layer interfaces, which makes it possible to evolve a layer implementation with minimal impact on other layers, and also to provide remote access to a layer. A Bridge (436) or an Object Adapter (438) supports the separation of the explicit interface of a layer from its encapsulated implementation.

Control and data can flow in both directions in layered systems. For example, data is exchanged between adjacent layers in layered protocol stacks such as TCP/IP or UDP/IP. However, Layers defines an acyclic downward dependency: lower layers must not depend on functions provided by higher layers. Such a design avoids accidental structural complexity, and supports the use of lower layers in other applications independently of the higher layers. Therefore, control flow that originates from the 'bottom' of the stack is often instigated via an Observer-based (399) callback infrastructure. Lower layers can pass data and service requests to higher layers via notifications realized as Commands (412) or Messages (420), without becoming dependent on specific functions in their interfaces.

Model-View-Controller **

When transforming a DOMAIN MODEL (182) into a technical software architecture, or specifying an agent in a PRESENTATION-ABSTRACTION-CONTROL (191) configuration ...

... we must consider that the user interface of an application changes more frequently than its domain functionality.

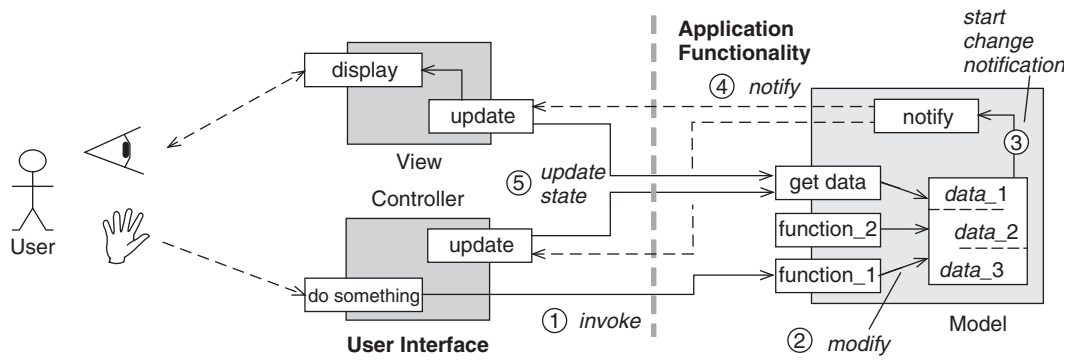
♦♦♦

User interfaces are prone to change requests: some must support multiple look-and-feel skins, others must address specific customer preferences. However, changes to a user interface must not affect an application's core functionality, which is generally independent of its presentation, and also changes less frequently.

Changes to a user interface should be both easy and local to the modified interface part. A changeable user interface must however not degrade the application's quality of service: at any time it must display the current state of computation, and respond to state changes immediately. To further complicate matters, in a system that supports multiple look-and-feel skins, each skin can change at a different rate, which requires additional decoupling of different user interface parts.

Therefore:

Divide the interactive application into three decoupled parts: processing, input, and output. Ensure the consistency of the three parts with the help of a change propagation mechanism.



Encapsulate the application's functional core inside a model whose implementation is independent of specific user interface look-and-feel and mechanics. For each aspect of the model to be presented in the application's user interface, introduce one or more self-contained views. Associate each view with a set of separate controllers that receive user input and translate this input into requests for either the model or the associated view. Let users interact with the application solely through the controllers.

Connect the model, view, and controller components via a change propagation mechanism: when the model changes its state, notify all views and controllers about this change so that they can update their state accordingly and immediately via the model's APIs.

♦♦♦

A MODEL-VIEW-CONTROLLER arrangement separates responsibilities of an application that tend to change at a different rate, to support their independent evolution.

The model defines the functional heart of the interactive application, thus its internal structure depends strongly on the application's specific domain responsibilities. Often the model is partitioned into one or more application DOMAIN OBJECTS (208), one for each self-contained responsibility. The implementation of the model should not rely on specific I/O data formats or view and controller APIs, to avoid having to change the model when the user interface changes.

Each coherent piece of information that is presented in the application's user interface is encapsulated within a self-contained view, together with functionality to retrieve the respective data from the model, transform this data into its output format, and display the output in the user interface. This self-containment allows views to evolve without affecting one another or the model. Two typical types of view are TEMPLATE VIEW (345) and TRANSFORM VIEW (347). A TEMPLATE VIEW renders model information into a predefined output format. A TRANSFORM VIEW creates its output by rendering each data element individually that it retrieves from the model.

Each view of the system is associated with one more controllers to manipulate the model's state. A controller receives input through an associated input device such as a keyboard or a mouse, and translates it into requests to its associated view or the model. There

are three common types of controller: a controller associated with a specific function in the application's user interface, a PAGE CONTROLLER (337) that handles all requests issued by a specific form or page in the user interface, and a FRONT CONTROLLER (339) that handles all requests on the model. A controller per function is most suitable if the model supports a wide range of functions. A PAGE CONTROLLER is appropriate for form-based or page-based user interfaces in which each form or page offers a set of related functions. A FRONT CONTROLLER is most usable if the application publishes functions to the user interface whose execution can differ for each specific request, such as the HTTP protocol of a Web application.

The requests issued by controllers may be encapsulated into COMMAND (412) objects that are passed to a dedicated COMMAND PROCESSOR (343) for execution. Such a design allows controllers to change transparently to both the views and the model. In addition, it supports the treatment of requests as first class objects, which in turn enables an application to offer 'house-keeping' services like undo/redo and request scheduling.

If a controller is in doubt over which concrete command to create, for example in a workflow-driven application, an APPLICATION CONTROLLER (341) helps to avoid dependencies on the model's internal state. In most applications, multiple controllers are active at the same time, but each user input can only be processed by one particular controller. A CHAIN OF RESPONSIBILITY (440) that connects all controllers simplifies the dispatching of the 'right' controller in response to a specific input.

Using WRAPPER FACADES (459) for accessing low-level device driver APIs and graphical libraries enables the views and controllers to be kept independent of the system's platform, as well as of its input and output devices. DATA TRANSFER OBJECTS (418) help to encapsulate the data that views and controllers retrieve from the model.

To support efficient collaboration between model, views, and controllers without breaking the model's independence of user interface aspects, connect them via an OBSERVER (405) arrangement. The model is a subject, while the views and controllers are its observers. When the model changes its state, it notifies all registered views and controllers, which in turn update their own state by retrieving the corresponding data from the model.

Presentation-Abstraction-Control

When transforming a DOMAIN MODEL (182) into a technical software architecture . . .

. . . we must at times consider that different functional responsibilities of an application can require different user interface paradigms.

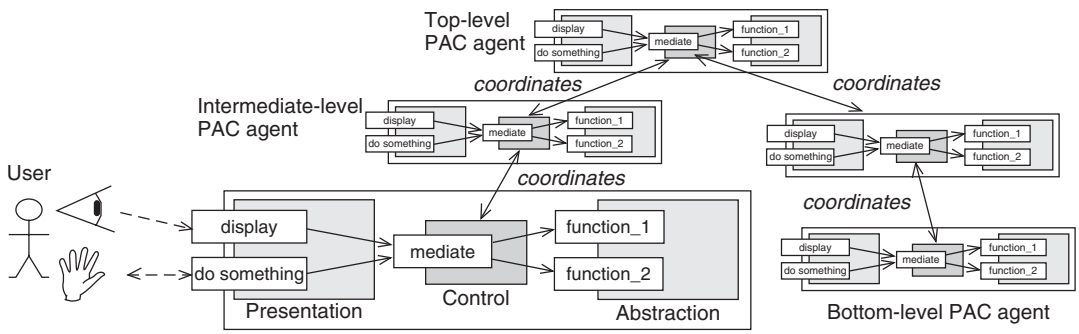
♦♦♦

A human-computer interface allows users to interact with an application via a specific 'paradigm,' such as forms or menus and dialogs. However, some applications are best operated via a distinct interface paradigm for each functionality type on offer.

For example, in a robot control system, the functionality for defining a mission requires a different user interface than the functionality for controlling a mobile robot during a mission. Yet we must ensure that all functions and their user interfaces form a coherent system. In addition, changes to any user interface should neither affect the implementation of its corresponding functionality, nor that of other functions and their associated user interfaces. Similarly, changes to the implementations of a distinct function should not affect user interfaces and implementations of other functions.

Therefore:

Structure the interactive application as a hierarchy of decoupled agents: one top-level agent, several intermediate-level agents, and many bottom-level agents. Each agent is responsible for a specific functionality of the application and provides a specialized user interface for it.



Bottom-level agents implement self-contained functionality with which users can interact, for example administration, error handling, and data manipulation. Mid-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.

Split each agent into three parts. A presentation part defines the agent's user interface. An abstraction part provides agent-specific domain functionality. A control part connects the presentation with the abstraction and allows the agent to communicate with other agents. Connect the agents in the hierarchy via their controls.

Users interact with an agent via its presentation. All user requests to the respective functionality in its abstraction are mediated by the agent's control. If a user action requires accessing or coordinating other agents, mediate this request to the controls of these agents, either up or down the hierarchy, and from there to their abstractions.

♦♦♦

A PRESENTATION-ABSTRACTION-CONTROL architecture helps to connect multiple self-contained subsystems, or even whole applications, with specialized human-computer interaction models to a coherent (distributed) system. The downside of such an arrangement is its complexity: multiple user interfaces must be provided, and actions instigated by a specific user interface must be coordinated carefully and explicitly if control flow spans multiple subsystems and causes reactions or view changes in their associated user interfaces. Consequently, a PRESENTATION-ABSTRACTION-CONTROL architecture only pays off if a software system cannot be implemented by a single user interface paradigm.

To specify a PRESENTATION-ABSTRACTION-CONTROL (PAC) architecture, identify all the self-contained responsibilities the application should offer to its users. Each responsibility is then encapsulated within a separate bottom-level agent. If several agents share functionality or need coordination, factor out this (coordination) functionality into an intermediate-level agent. There can be multiple levels of intermediate-level agents within a PAC architecture. Functionality shared by all agents is provided by the top-level agent. Such decoupling supports independent modification of agents without affecting other agents,

and allows each agent to provide its own user interface. Provide all agents with a MODEL-VIEW-CONTROLLER (188) architecture: the abstraction corresponds with the model and its partitioning into DOMAIN OBJECTS (208), and the presentation to the views and controllers. Changes to an agent's interface will therefore affect its realization.

Decouple an agent's abstraction from its presentation via a control component that is a MEDIATOR (410) with a twofold responsibility. First, it must route all user requests from the agent's presentation to the appropriate functionality in its abstraction. It must also 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 received by a particular agent cannot be handled by the agent alone, the control routes the request to the controls of appropriate higher- or lower-level agents, together with its associated input data. Results are returned in the same way, but in reverse. Similarly, the control of an agent can receive requests and data from the controls of other agents. The requests to be routed can be encapsulated inside COMMAND (412) objects, and the data inside DATA TRANSFER OBJECTS (418). Controls are the key to a loose coupling between agents: if an agent's abstraction changes, effects on other agents are limited to their controls.

To keep agents consistent with one another, connect them via an OBSERVER (405) arrangement. An agent that is dependent on the state of its associated higher- or lower-level agents registers its control as a subscriber of the other agents' controls, which play the role of subjects. Whenever one of these 'subject' agents changes its state, its control notifies the control of the 'observing' agents, which can then react appropriately to update their own state.

Microkernel **

When transforming a DOMAIN MODEL (182) into a technical software architecture . . .

. . . we must design support for functional scalability and adaptability in different deployment scenarios.

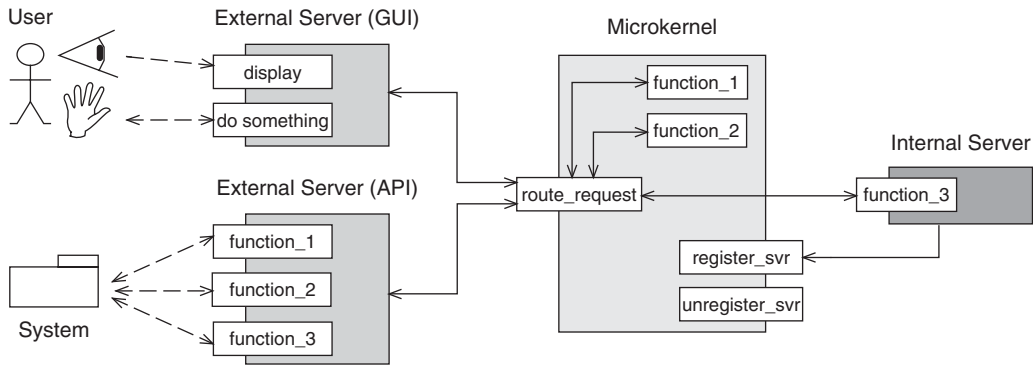
♦♦♦

Some applications exist in multiple versions. Each version offers a different set of functionality to its users, or differs from other versions in specific aspects, such as its user interface. Despite their differences, however, all versions of the application should be based on a common architecture and functional core.

The goal is to avoid architectural drift between the versions of the application and to minimize development and maintenance effort for shared functionality. In addition, upgrading one version of the application to another by adding and removing features, or by changing their implementation, should require no or only minimal modifications to the system. Similarly, it should be easy to provide a particular application version with different user interfaces, and also to run the version on different platforms, allowing clients to use it most appropriately within their specific environments.

Therefore:

Compose different versions of the application by extending a common but minimal core via a 'plug-and-play' infrastructure.



A microkernel implements the functionality shared by all application versions and provides the infrastructure for integrating version-specific functionality. Internal servers implement self-contained version-specific functionality, 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 function set and look-and-feel.

Clients, whether human or other software systems, access the microkernel's functionality solely via the interfaces or APIs provided by the external servers, which forward all requests they receive to the microkernel. If the microkernel implements the requested function itself, it executes the function, 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 can be tailored exactly for its purpose. Users or client systems only get the functionality and look-and-feel that they require, but do not have to incur the cost of anything they do not need. In general, evolving a particular version towards new or different functions and aspects 'only' requires reconfiguring it with appropriate internal and external servers: the microkernel itself is unaffected by such upgrades. Existing internal and external servers and other application versions are similarly unaffected. In addition, a MICROKERNEL architecture minimizes development and maintenance efforts for all members of the application family: each service, user interface, or API is implemented only once.

The internal structure of the microkernel is typically based on LAYERS (185). The bottommost layer abstracts from the underlying system platform, thereby supporting the portability of all higher levels. The second layer implements infrastructure functionality, such as resource management, on which the microkernel depends. The layer above hosts the domain 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.

Each specific and self-contained function and responsibility within the microkernel can be realized as a DOMAIN OBJECT (208), which supports its independent implementation and evolution. The routing functionality of the microkernel is often implemented as a MEDIATOR (410) that receives requests through a uniform interface and dispatches these requests onto corresponding domain functions in the microkernel or the internal servers. To minimize resource consumption, particularly memory, the routing layer can use a COMPONENT CONFIGURATOR (490) or an OBJECT MANAGER (492) to load internal servers on demand, unload them after use, and control their lifecycle. This design also supports the upgrade of a particular application version with new, different, or modified functionality dynamically at runtime.

Internal servers follow a similar LAYERS design as the microkernel, but do not usually 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 avoid implementing these services and abstractions themselves, and instead call back the corresponding layers in the microkernel. This keeps the server's footprint small, but at the expense of additional runtime overhead to perform the callbacks. To minimize network traffic in a distributed system, and to increase the performance of internal servers, therefore, it may be beneficial to provide them with all the system services and platform abstractions that they need.

The design of an external server strongly depends on its complexity and purpose. It can range from a simple OBJECT ADAPTER (438) that maps the application's published APIs onto its internal APIs, to a complex user interface.

The application-specific data exchanged between external servers, the microkernel, and its configured internal servers can be encapsulated inside DATA TRANSFER OBJECTS (418).

Reflection *

When transforming a DOMAIN MODEL (182) into a technical software architecture . . .

. . . we must sometimes provide a design that is prepared for evolution and integration of unanticipated changes.

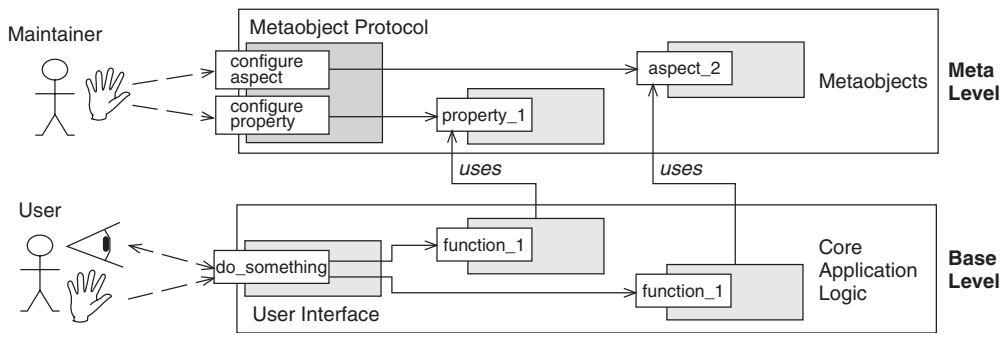
♦♦♦

Support for variation is the key to sustainable architectures for long-lived applications: over time they must respond to evolving and changing technologies, requirements, and platforms. However, it is hard to forecast what can vary in an application and when it must respond to a specific variation request.

To complicate matters, the need for variation can occur at any time, specifically while the application is in productive use. Variations can also be of any scale, ranging from local adjustments of an algorithm to fundamental modifications of distribution infrastructure. Yet, while the variation of the application should be possible at appropriate times, the complexity associated with particular variations should be hidden from maintainers, and there should be a uniform mechanism for supporting different types of variation.

Therefore:

Objectify information about properties and variant aspects of the application's structure, behavior, and state into a set of metaobjects. Separate the metaobjects from the core application logic via a two-layer architecture: a meta level contains the metaobjects, a base level the application logic.



Provide the meta level with a metaobject protocol, which is a specialized interface that administrators, maintainers, or even other systems can use to *dynamically* configure and modify all metaobjects under the supervising control of the application. 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.

♦♦♦

REFLECTION supports a high degree of runtime flexibility in a software architecture. Almost any information about a software system can be made accessible; and any aspect that can change can be made (ex)changeable. Some programming languages, therefore, support specific flavors of REFLECTION directly, such as `Java` with the `java.lang.reflect` package and C# with the `System.Reflection` namespace. Note, however, that the heavyweight measures of a REFLECTION architecture only pay off if there are similarly heavyweight flexibility requirements that justify these measures.

To realize a REFLECTION architecture, first specify a stable design for the application that does not consider flexibility at all: stability is the key to flexibility [Bus03]. Typically, each self-contained responsibility of the application is encapsulated within a DOMAIN OBJECT (208), which together form the base level of the REFLECTION architecture.

Using a suitable method, such as *Open Implementation Analysis and Design* [KLLM95], *Commonality/Variability Analysis* [Cope98], or *Feature Modeling* [CzEi02], identify all the structural and behavioral aspects of the application that can vary. Variant behavior often includes algorithms for application functionality, lifecycle control of domain objects, transaction protocols, IPC mechanisms, and policies for security and failure handling. There may even be the need to add completely new behavior to the system or to remove existing behavior.

Structural aspects that can vary include the application's thread or process model, the deployment of domain objects to processes and threads, or even the system's type structure. In addition, determine all system-wide information, properties, and global state that can influence the behavior of the application, such as runtime type information about what interfaces domain objects offer, what their inner structure is, or whether they are persistent.

Realize each variant behavioral and structural aspect, system property, and state identified in the analysis as a separate metaobject, and assign all metaobjects to the meta level of the REFLECTION architecture. Such a strict encapsulation makes the aspects explicitly accessible, and thus (ex)changeable at any time. Changes to metaobjects also cannot ripple through to the implementation of the application's base level.

Open the implementation of each DOMAIN OBJECT at the base level such that it consults an appropriate metaobject for each aspect encapsulated in the meta level. Changes to the metaobjects thus immediately impact the base level's subsequent behavior.

To support the creation, configuration, exchange, and disposal of metaobjects at runtime, introduce a metaobject protocol that serves as the sole interface to manage the meta level. The metaobject lifecycle infrastructure that is necessary for these activities can be realized with help of ABSTRACT FACTORIES (525) and BUILDERS (527) to create and dispose of metaobjects, and a COMPONENT CONFIGURATOR (490) or an OBJECT MANAGER (492) to control the execution of specific metaobject lifecycle steps. Such a design also enables the integration of metaobjects that were developed after the reflective application went live with the meta level. An INTROSPECTIVE INTERFACE (286) and a DYNAMIC INVOCATION INTERFACE (288) support application-external clients such as test frameworks or object browsers, to obtain information about base-level domain objects without becoming dependent on their internal structure, as well as invoking methods on them without the need to use their functional interfaces.

The metaobject protocol in conjunction with the two-layer structure of a REFLECTION architecture is a prime example of how the open/close principle [Mey97] can be realized. The metaobject protocol hides the complexity of software evolution behind a 'simpler' interface, making it easy, uniform, and dynamic, but allows the reflective application to supervise its own evolution so that uncontrolled changes are minimized. The separation of a REFLECTION architecture into a base level and a meta level strictly separates variant from invariant aspects in an application: metaobjects can be managed without implications for the internal design and implementation of the domain objects at the base level.

Pipes and Filters **

When transforming a DOMAIN MODEL (182) into a technical software architecture . . .

. . . we must sometimes provide a design that is suitable for processing data streams.

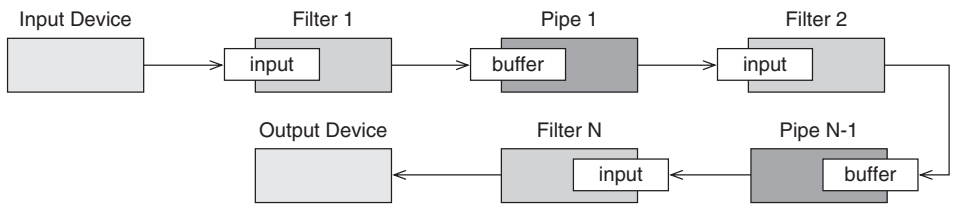
♦♦♦

Some applications process streams of data: input data streams are transformed stepwise into output data streams. However, using common and familiar request/response semantics for structuring such types of application is typically impractical. Instead we must specify an appropriate data flow model for them.

Modeling a data-flow-driven application raises some non-trivial developmental and operational challenges. First, the parts of the application should correspond to discrete and distinguishable actions on the data flow. Second, some usage scenarios require explicit access to intermediate yet meaningful results. Third, the chosen data flow model should allow applications to read, process, and write data streams incrementally rather than wholesale and sequentially so that throughput is maximized. Last but not least, long-duration processing activities must not become a performance bottleneck.

Therefore:

Divide the application's task into several self-contained data processing steps and connect these steps to a data processing pipeline via intermediate data buffers.



Implement each processing step as a separate filter component that consumes and delivers data incrementally, and chain the filters such

that they model the application's main data flow. In the data processing pipeline, data that is produced by one filter is consumed by its subsequent filters. Adjacent filters are decoupled using pipes that buffer data exchanged between the filters.

♦♦♦

A PIPES AND FILTERS architecture decouples different data processing steps so that they can evolve independently of one another and support an incremental data processing approach.

Within a PIPES AND FILTERS architecture, filters are the units of domain-specific computation. Each filter can be implemented as a DOMAIN OBJECT (208) that represents a specific, self-contained data processing step. Filters with a concurrent DOMAIN OBJECT implementation enable incremental and concurrent data processing, which increases the performance and throughput of a PIPES AND FILTERS arrangement. If a filter performs a long-duration activity, consider integrating multiple parallel instances of the filter into the processing chain. Such a configuration can further increase system performance and throughput, as some filter instances can start processing new data streams while others are processing previous data streams.

Pipes are the medium of data exchange and coordination within a PIPES AND FILTERS architecture. Each pipe is a DOMAIN OBJECT that implements a policy for buffering and passing data along the filter chain: data producing filters write data into a pipe, while data consuming filters receive their input from a pipe. The integration of pipes decouples adjacent filters so that the filters can operate independently of one another, which maximizes their individual operational performance.

In a single-process PIPES AND FILTERS arrangement, pipes are typically implemented as queues. Pipes with a concurrent DOMAIN OBJECT implementation enable incremental and concurrent data processing, as do concurrent filters. In a distributed arrangement, pipes are realized as some form of MESSAGING (221) infrastructure that passes data streams between remote filters. Pipes that are implemented as a DOMAIN OBJECT shield filters from a knowledge of their specific implementation, which also allows transparent swapping of implementation forms. Such a design supports a flexible (re-)deployment of filters in a distributed PIPES AND FILTERS arrangement. MESSAGES (420) help to encapsulate the data streams that are passed along the pipes.

Shared Repository **

When transforming a DOMAIN MODEL (182) into a technical software architecture . . .

. . . we must sometimes provide a design for applications whose parts operate on, and coordinate their cooperation via, a set of shared data.

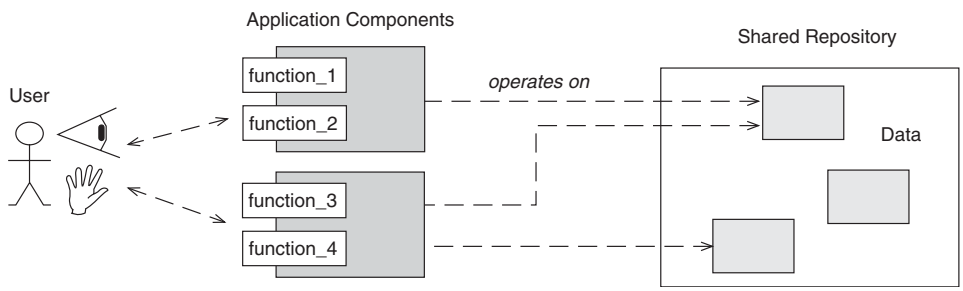
♦♦♦

Some applications are inherently data-driven: interactions between components do not follow a specific business process, but depend on the data on which they operate. However, despite the lack of a functional means to connect the components of such applications, they must still interact in a controlled manner.

One example of a data-driven system is a network management and control application, such as a Telecommunication Management Network (TMN) system. Such systems operate on massive amounts of data provided by field devices. Core responsibilities like monitoring and control, alarming, and reporting are largely independent of one another, and it is the state of the data that determines the control flow and collaboration of these tasks. Connecting the tasks directly would hard-code a specific business process into the application, which may be inappropriate if specific data is unavailable, not of the required quality, or in a specific state. However, we need a coherent computational state across the entire application.

Therefore:

Maintain all data in a central repository shared by all functional components of the data-driven application and let the availability, quality, and state of that data trigger and coordinate the control flow of the application logic.



Components work directly on the data maintained by the shared repository, so that other components can react if this data changes. If a component creates new data, or if the application receives new data from its environment, is also stored in the shared repository, to make it accessible to other components.

♦♦♦

A SHARED REPOSITORY architecture allows integration of application functionality with a data-driven control flow to form coherent software systems. It also supports coherent integration of applications that operate on the same data, but neither share nor participate in common business processes. Coordinating components via the state of shared data can introduce performance and scalability bottlenecks, however, if many concurrent components need access to the same data exclusively and are thus serialized.

The shared repository is the central control coordination entity and data access point of a data-driven application. It can be as simple as an in-memory data collection, or as complex as an external data repository that is accessed via a DATABASE ACCESS LAYER (438). If the shared repository is implemented as a DOMAIN OBJECT (208), its concrete implementation is hidden from the application's components and can be swapped or modified transparently. DATA TRANSFER OBJECTS (418) help to encapsulate the data passed between the shared repository and the components of the application.

The data maintained by the shared repository is often encapsulated inside managed objects: DOMAIN OBJECTS that hide the details of concrete data structures and offer meaningful operations for their access and modification. Managed objects allow the application's components to use specific data without becoming dependent on its concrete representation, and support the modification of data representations without effects on the components that use the data. Managed objects can also indicate the quality of the represented data via a corresponding quality attribute, for example that the data is up-to-date, out-of-date, uncertain, or corrupted. Components can use this information to control the specific treatment of that data.

In general, access to the shared repository and its managed objects must be synchronized, because multiple components of the application can access it concurrently. In most configurations, this synchronization happens at the level of managed objects, which

maximizes the potential concurrency within the data-driven application. Providing a managed object with a Thread-Safe Interface (406) enforces synchronization at the interface of the managed object. If only small portions of its methods are critical sections, synchronization via Strategized Locking (388) is a possible alternative. Realizing a managed object as a Monitor Object (390) supports cooperative concurrency control of multiple components that access the managed object simultaneously.

Many shared repositories offer a mechanism for notifying application components about data changes within the repository. For example, new data may have been inserted, existing data modified, or data may have been dropped. Components can therefore react immediately to changes to the data in the repository. In most cases, the change notification mechanism is realized by an Observer (405) arrangement: the shared repository is the subject, the application's components are its observers. Similarly, managed objects can also offer an Observer-based change propagation mechanism, which allows them to notify components about specific value changes.

Which of the two options best suits a data-driven application depends on its concrete responsibilities. The trade-off to consider is simplicity versus granularity: change notification at the level of the shared repository is simple to implement, but could cause overhead due to notification of components that are not interested in the changes reported. Vice versa, a mechanism implemented at the level of managed objects avoids unnecessary notifications and data transfer, but is of higher complexity. The more components of an application that operate on the entire data maintained by the shared repository, the more feasible a notification mechanism at the repository level becomes, while the more selectively components access managed objects, the more notification at the level of managed objects is the best fit.

The data-driven service components of the application are typically implemented as Domain Objects that realize a specific responsibility by accessing and manipulating data in the shared repository. Cooperation between the components happens purely at the data level, by notifying other components when a specific managed object changes its state, or when data is inserted into, or deleted from, the shared repository.

Blackboard

When transforming a DOMAIN MODEL (182) into a technical software architecture . . .

. . . we must sometimes provide a design suitable for applications that resolve tasks for which no deterministic solution strategy is known.

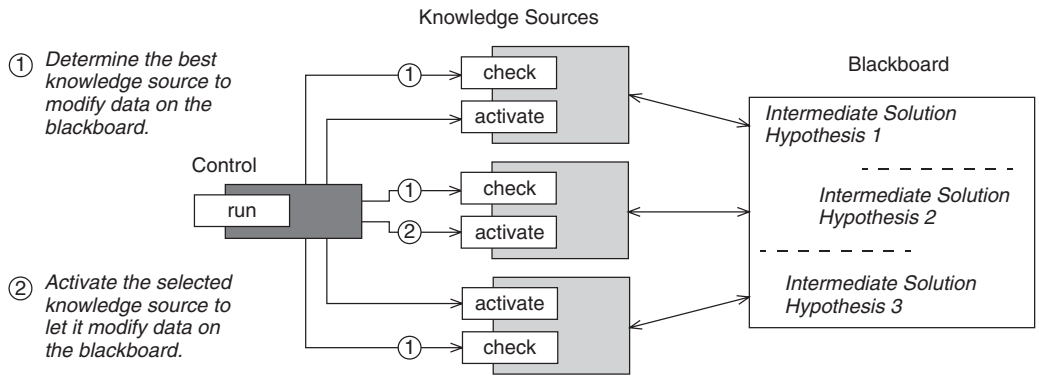
♦♦♦

For some tasks no deterministic solution algorithms are known, only approximate or uncertain knowledge is accessible. However, despite this lack of proper algorithmic support, trial-and-error techniques can be sufficiently successful and it is necessary to develop productive applications for these types of task.

Examples of such systems include speech recognition, submarine detection based on sonar signals, and the inference of protein molecule structures from X-ray data. Such tasks must resolve several hard challenges: input data is often fuzzy or inaccurate, the path towards a solution must be explored, every processing step can generate alternative results, and often no optimal solution is known. Nevertheless, it is important to compute valuable solutions in a reasonable amount of time.

Therefore:

Use heuristic computation to resolve the task via multiple smaller components with deterministic solution algorithms that gradually improve an 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 knowledge source. To allow the knowledge sources to execute independently of one another and 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 input data is available for them, process this input, and deliver their results, which may then form the input for any other knowledge source in the system.

Coordinate the computation with a control component that uses an opportunistic heuristic to select and activate adequate 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 algorithm.

♦♦♦

A BLACKBOARD architecture helps in the construction of software systems that must resolve tasks on the basis of uncertain, hypothetical, or incomplete knowledge and data. It also helps to discover and optimize strongly deterministic solutions for tasks that lack such solutions. On the other hand, there is no guarantee that a BLACKBOARD-based system actually produces a useful result. In addition, a computational approach based on heuristics is often not feasible for systems that demand a predictability in terms of result quality and the time in which a result is produced.

To implement a BLACKBOARD system, first decompose the task that it must resolve. What input does the system receive? What form of output should it produce? What potential solution paths and intermediate results towards a solution are known? What are the well-known algorithms that can contribute to the solution (path)? What input or intermediate results can each algorithm process? What intermediate or final results can each algorithm deliver? On the basis of this analysis, define self-contained and independently executable knowledge sources for every algorithm that is involved in the tasks's solution. Such independence allows the execution order of knowledge sources to be *arbitrary*—a necessary precondition for a heuristic solution strategy. To allow a heuristic to determine a *particular* execution

order, split each knowledge source into two separate parts. A condition part examines whether 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 inputs from the blackboard, processes it, and writes one or more outputs back to the blackboard. Alternatively, the action part could erase data from the blackboard because it identifies the data as not contributing to the overall task's solution. Typically, a knowledge source is implemented as a DOMAIN OBJECT (208), which supports its independent evolution and optimization when more knowledge about the application's overall task becomes available.

The blackboard is a data repository that maintains all partial and final results that the knowledge sources produce. It can be designed as an in-memory data collection, or as an external data repository that is accessed via a DATABASE ACCESS LAYER (538). If the blackboard is implemented as a DOMAIN OBJECT, its concrete implementation is hidden from the knowledge sources and can be swapped or modified transparently. DATA TRANSFER OBJECTS (418) help to encapsulate the data passed between the blackboard and the knowledge sources.

A 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 whether 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 final step invokes the action part of the selected knowledge source, which then modifies the blackboard's content. Once this knowledge source finishes its execution, the loop starts over again.

The loop ends if the blackboard contains a valid final result, or if none of the knowledge sources can improve the quality of any intermediate solution hypothesis on the blackboard. Implementing the control component as a DOMAIN OBJECT allows the chosen heuristics to be modified and evolved transparently for the knowledge sources and the blackboard of a concrete BLACKBOARD arrangement.

Domain Object **

When realizing a DOMAIN MODEL (182), or its technical architecture in terms of LAYERS (185), MODEL-VIEW-CONTROLLER (188), PRESENTATION-ABSTRACTION-CONTROL (191), MICROKERNEL (194), REFLECTION (197), PIPES AND FILTERS (200), SHARED REPOSITORY (202), or BLACKBOARD (205) . . .

. . . a key concern of all design work is to decouple self-contained and coherent application responsibilities from one another.

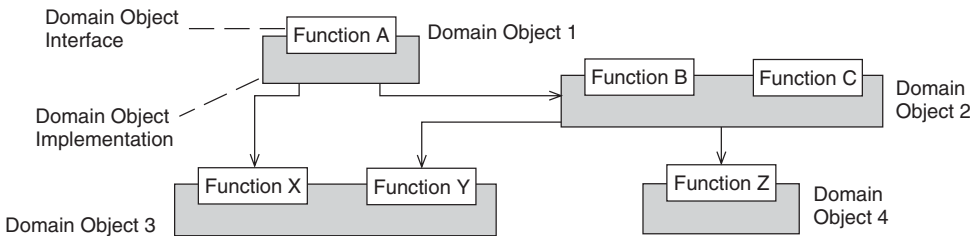
♦♦♦

The parts that make up a software system often expose manifold collaboration and containment relationships to one another. However, implementing such interrelated functionality without care can result in a design with a high structural complexity.

Separation of concerns is a key property of well-designed software. The more decoupled are the different parts of a software system, the better they can be developed and evolved independently. The fewer relationships the parts have to one another, the smaller the structural complexity of the software architecture. The looser the parts are coupled, the better they can be deployed in a computer network or composed into larger applications. In other words, a proper partitioning of a software system avoids architectural fragmentation, and developers can better maintain, evolve and reason about it. Yet despite the need for clear separation of concerns, the implementation of and collaboration between different parts in a software system must be effective and efficient for key operational qualities, such as performance, error handling, and security.

Therefore:

Encapsulate each distinct functionality of an application in a self-contained building-block—a domain object.



Provide all domain objects with an interface that is separate from their implementation, and within each domain object program only using these interfaces when accessing other domain objects.

♦♦♦

DOMAIN OBJECT separates different *functional* responsibilities within an application such that each functionality is well encapsulated and can evolve independently. The specific partitioning of an application's responsibilities into domain objects is based on one or more granularity criteria. An APPLICATION SERVICE [ACM01] is a domain object that encapsulates a self-contained and complete business feature or infrastructure aspect of an application, such as a banking, flight booking, or logging service [Kaye03]. A COMPONENT [VSW02] is a domain object that either encapsulates a functional building block such as an income tax calculation or a currency conversion, or a domain entity such as a bank account or a user. A VALUE OBJECT [PPR] [Fow03a], a COPIED VALUE (394), and an IMMUTABLE VALUE (396) are small domain objects whose identity is based on their state rather than their type, such as a date, a currency exchange rate, or an amount of money. A domain object can also aggregate other domain objects of the same or smaller granularity. For example, services are often created from components that use value objects.

Split each domain object into an EXPLICIT INTERFACE (281) that exports its functionality and an ENCAPSULATED IMPLEMENTATION (313) that realizes the functionality. This separation of interface and implementation minimizes inter-domain-object coupling: each domain object only depends on domain object interfaces, but not on domain object implementations. It is thus possible to realize and evolve a domain object implementation with minimal effect on other domain objects. The explicit interface of a domain object defines a contract for key operational properties, such as error behavior and security aspects, on which other domain objects can rely.

There are several options for connecting the explicit interface of a domain object with its encapsulated implementation. For example, Java and C# support the concept of explicit interface in the core language, and classes (encapsulated implementations) can implement them directly. In other statically typed languages such as C++, an explicit interface can be expressed as an abstract base class from which the explicit implementation derives.

A Bridge (436) or an Object Adapter (438) explicitly decouples the explicit interface of a domain object from its encapsulated implementation so that the two can vary independently. The degree of decoupling between the explicit interface of a domain object and its encapsulated implementation depends on its granularity and likelihood of change. The smaller the domain objects, for example when realizing a Value Object or an Immutable Value, the less beneficial strict decoupling becomes. Similarly, the more often an encapsulated implementation evolves, the more strongly the explicit interface should be decoupled.

Explicit interfaces also enable remote access to domain objects. Note, however, that remoting is generally feasible only for 'larger' domain objects such as services and coarse-grained components, but not for 'small' domain objects like a value object. The smaller are the domain objects, the more adverse is the ratio of networking overhead versus computation time inside the domain object, with corresponding penalties on operational quality factors such as performance, availability, and scalability.

Domain objects are often associated with an Abstract Factory (525) or Builder (527) that allows clients to obtain access to their explicit interface and to manage their lifetime transparently. On platforms like CCM [OMG02], EJB [MaHa99], and. NET [Ram02], domain objects are controlled by a Declarative Component Configuration (461) that specifies how their lifecycle, resource management, and other technical concerns like transactions and logging should be handled by their hosting environment. A Component Configurator (490) helps with loading, replacing, (re)configuring, and unloading domain objects at runtime.

10 Distribution Infrastructure

Zhaoqing power converter station for high-voltage direct-current transmission line, Guangdong province, China Siemens press picture, © Siemens AG

It is hard to meet complex distributed system requirements such as scalability and dependability if only the application, host operating system, and network perspectives are considered. The application should focus on 'business logic' rather than 'plumbing,' and the operating system and network should focus on endsystem resource management and communication protocol processing respectively. To address other key perspectives, this chapter describes twelve patterns pertaining to *middleware*, which is distribution infrastructure software that shields applications from many inherent and accidental complexities of operating systems and networks.

Several trends influence the way we conceive and construct distributed systems [ScSc01]:

- Information technology is becoming commoditized: hardware and software are generally getting more powerful, cheaper, and better at a relatively predictable rate. The commoditization of hardware, such as CPUs and storage devices, and networking elements such as IP routers and WiFi devices, has been underway for decades. More recently, software is being commoditized due to the maturation of object-oriented languages such as Java, C#, and C++, and commercial-of-the-shelf operating environments, such as Linux, Windows, and Java virtual machines.
- There is a growing acceptance of the service-oriented software paradigm, in which distributed systems with a range of requirements are constructed by integrating separate services that are connected by various forms of network protocols. The nature of these interconnections can range from very small and tightly coupled applications, such as anti-lock braking systems, to very large and loosely coupled systems, such as the Internet and World Wide Web.

The interplay of these two trends has yielded new architectural concepts and services embodying layers of *middleware*, such as MQ Series, CORBA, Enterprise Java Beans, DDS, and Web Services. Middleware is distribution infrastructure software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure, to coordinate how parts of applications are connected and how they inter-operate. Middleware also enables and simplifies the integration of components developed by different technology suppliers.

When implemented and applied properly, middleware addresses many of the challenges described in Chapter 2. For example, it shields developers from many low-level platform details, such as socket-level network programming, thereby enabling them to focus on their

application's business logic requirements. Middleware can also amortize software lifecycle costs by leveraging previous development expertise via reusable frameworks and services that are needed to operate effectively in a networked environment, rather than handcrafting them for each use.

Developing communication middleware that masters the challenges outlined above and in Chapter 2 is complex and time-consuming. Fortunately, there is rarely a need to design and implement your own approaches. A wide range of communication middleware standards and commercial off-the-shelf platforms are now available, such as CORBA [OMG04a], .NET Remoting [Ram02], the Microsoft Communication Framework [Pal05], and JMS [HBS+02], which are used successfully in many distributed systems.

The drawback of having so many different standards and products, of course, is that you now have to consider more options for your projects and systems. Selecting the 'best' communication paradigm, middleware standard, and product depends on many factors, including price, support, quality, and the requirements of the systems being developed. Rarely does one middleware solution work optimally for all applications in a distributed system.

To enhance productivity in a given distributed application, the selected middleware must also be used correctly. Several projects have failed [Bus03] due to insufficient understanding of the communication paradigm and a lack of knowledge about the key structure and behavior of chosen middleware. Selecting and using specific communication middleware therefore requires thoughtful consideration and explicit decisions.

The main intent for including this chapter in our pattern language for distributed computing is to help you understand different communication middleware approaches and their internal designs. Armed with a knowledge of the fundamental properties of each approach and their benefits and liabilities, you can choose the right communication paradigm and middleware for applications in your distributed system.

Despite their detailed differences, middleware technologies typically follow one or more of three different communication styles: *messaging*, *publish/subscribe*, and *remote method invocation*, which are reflected by the following three entry-point patterns in this chapter:

> The MESSAGING pattern (221) [HoWo03] structures distributed software systems whose services interact by exchanging messages. A set of interconnected message channels and message routers manages the exchange of messages between services across the network, including passing request and reply messages that contain information, metadata, and error information.
>
> The PUBLISHER-SUBSCRIBER pattern (234) structures distributed software systems whose services or components interact by exchanging events asynchronously in a one-to-many configuration. Publishers and subscribers of events are generally unaware of one another. Subscribers are interested in consuming events, not in knowing their publishers. Similarly, publishers just supply events, and are not interested in who subscribes to them.
>
> The BROKER pattern (237) [POSA1] [VKZ04] structures distributed software systems whose components interact by remote method invocations. A federation of brokers manages key aspects of interprocess communication between components, ranging from forwarding requests to transmitting results and exceptions.

Several criteria distinguish these patterns from one another, including their communication models—such as many-to-one versus one-to-many—and the degree of coupling between an application's components. In particular:

- In BROKER, many clients can make remote method invocations on specific remote component objects hosted by a server. Clients thus communicate with the server objects in a many-to-one fashion, and their functional interfaces are often statically typed. The remote method invocation style of communication provided by BROKER is best suited for systems that try to hide the presence of the network.
- MESSAGING relaxes this coupling and typing: clients send dynamically typed messages to specific remote services that reside at communication endpoints, not (necessarily) to specific methods. MESSAGING thus enables many-to-one communication without statically predefining the interface dependencies of clients to services.

- PUBLISHER-SUBSCRIBER decouples an application's components even more: they can exchange events in a one-to-many manner without knowing one another's identity explicitly, and without having to make a request each time new events are available. PUBLISHER-SUBSCRIBER middleware is therefore responsible for tracking which subscribers receive specific events sent asynchronously by publishers. Subscribers react when receiving an event by performing some action, but publishers do not directly initiate the execution of a specific method on the subscribers.

The following table summarizes these differences:

Pattern	Communication Style	Communication Relationships	Component Dependencies
Broker	Remote Method Invocation	One-to-one	Component interfaces
Messaging	Message	Many-to-one	Communication endpoints Message formats
Publisher-Subscriber	Events	One-to-many	Event formats

Though achieving completely location-transparent communication in a distributed system is infeasible [WWWK96], BROKER makes invocations on remote component objects look and act as much as possible like invocations on component objects in the same address space as their clients. MESSAGING and PUBLISHER-SUBSCRIBER are most appropriate for integration scenarios in which multiple, independently developed and self-contained services or applications must collaborate and form a coherent software system. MESSAGING still allows services to exchange requests and responses, whereas the entire collaboration between components in PUBLISHER-SUBSCRIBER is coordinated by notifying subscribers about state changes and other events of interest.

In practice, middleware platforms and products often implement one or more of these patterns. For example, Web Services implements PUBLISHER-SUBSCRIBER via WS-NOTIFICATION and MESSAGING via SOAP, whereas CORBA implements PUBLISHER-SUBSCRIBER via the

Notification Service and BROKER via the ORB itself. Some CORBA BROKER implementations, such as BEA's Web Logic Enterprise, are even implemented on top of the Tuxedo MESSAGING middleware. In general, the CORBA ORB Core can be viewed as the MESSAGING layer of the CORBA BROKER architecture.

Other distributed computing literature lists additional communication styles, such as shared databases, data streaming, file transfer, and peer-to-peer [Fow03a] [HoWo03] [VKZ04]. We consider these more as approaches for *orchestrating the collaboration* of an application's services, however, rather than a style they use to *exchange information*. This distinction explains why we present patterns like PIPES AND FILTERS (200) and SHARED REPOSITORY (202) in Chapter 9 rather than in this chapter. The data streaming style, for example, can be realized via a PIPES AND FILTERS design with the pipes being middleware based on MESSAGING or BROKER. Similarly, the shared repository style can be realized with a SHARED REPOSITORY design using middleware based on BROKER or PUBLISHER-SUBSCRIBER.

MESSAGING, BROKER, and PUBLISHER-SUBSCRIBER are just the entry points into the patterns in this chapter. To be usable for a distributed application, each type of middleware must address many different issues, each representing its own problems and offering a coherent solution space. It is therefore natural to document these problems and their solutions as separate patterns—in fact, the architecture of many middleware platforms today are guided and documented by many such patterns [SC99] [ACM01] [VSW02] [VKZ04] [MS03].

Middleware based on MESSAGING can be refined by the following four distribution infrastructure patterns:

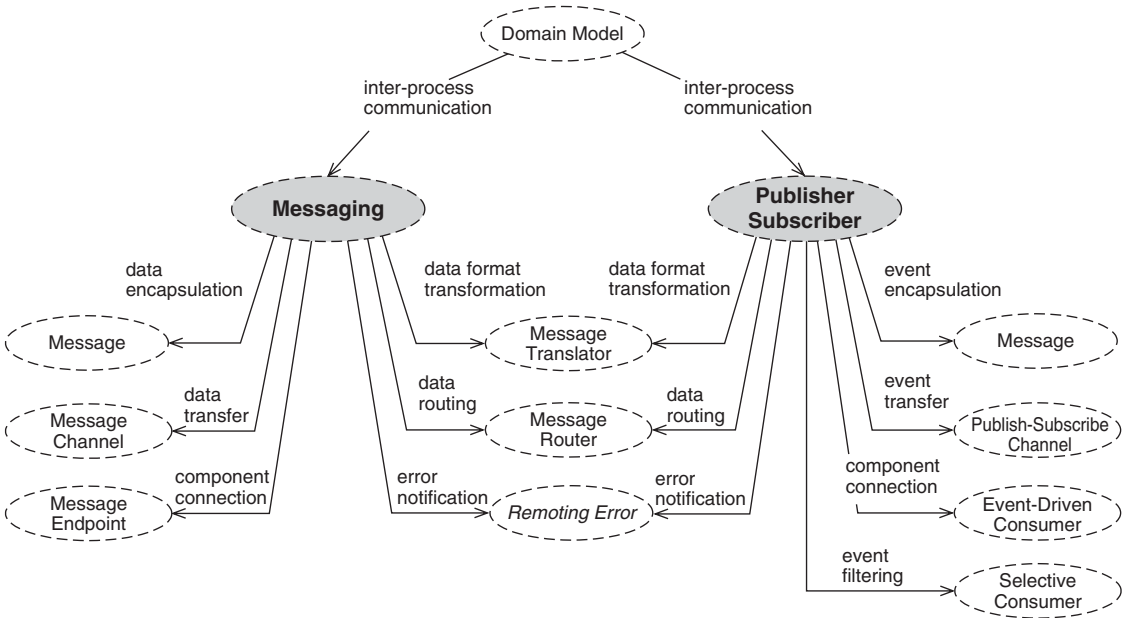
> The MESSAGE CHANNEL pattern (224) [HoWo03] connects application services that interact by exchanging messages. One service writes information to the channel and the other reads that information from the channel.
>
> The MESSAGE ROUTER pattern (231) [HoWo03] allows a client to send messages to other services of an application depending on a set of conditions.

The MESSAGE TRANSLATOR pattern (229) [HoWo03] supports the translation of a message into another form if the sender of a message and its reader expect different message formats.

The MESSAGE ENDPOINT pattern (227) [HoWo03] helps application services connect with the messaging infrastructure by encapsulating and implementing the necessary adaptation code.

MESSAGING middleware is defined by other distribution infrastructure patterns beyond MESSAGE, MESSAGE CHANNEL, MESSAGE ROUTER, MESSAGE TRANSLATOR, and MESSAGE ENDPOINT. Each of these four patterns references other finer-grained patterns that assist their further decomposition and implementation. We do not cover these patterns in detail, but instead refer to their original source, *Enterprise Integration Patterns* [HoWo03]. Nonetheless, all these patterns form an integral part of our pattern language for distributed computing.

The following diagram outlines how the MESSAGING and PUBLISHER-SUBSCRIBER patterns integrate with our pattern language.



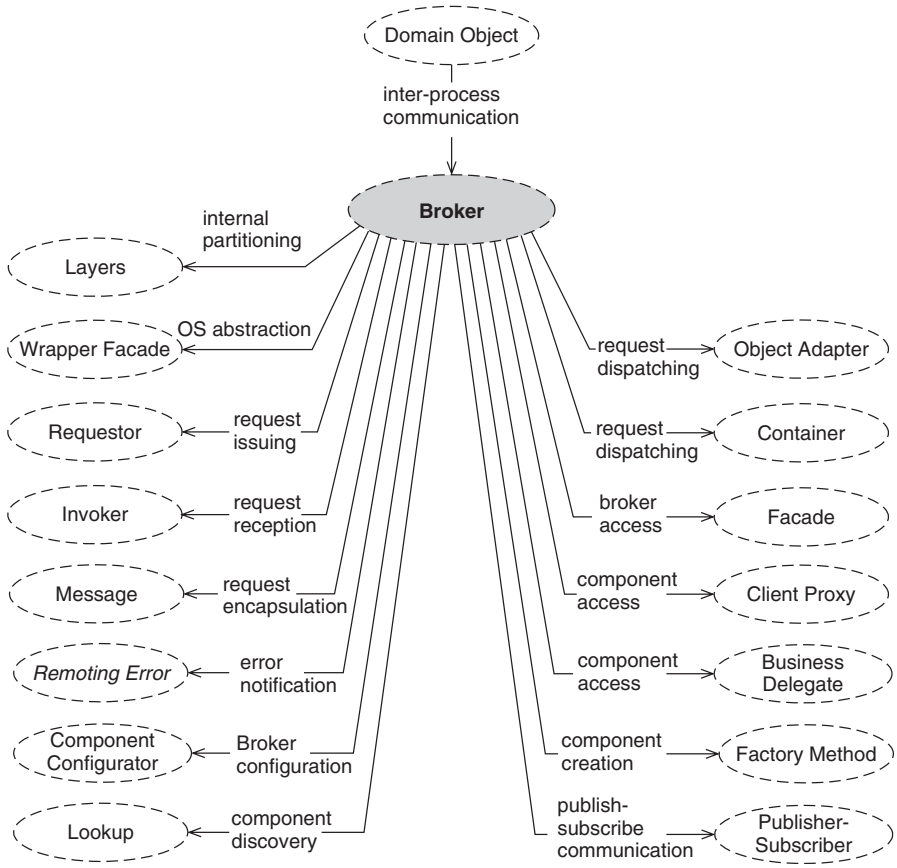
The responsibilities of BROKER middleware can be decomposed into the following five distribution infrastructure patterns, which are described in the order they are applied from client to server:

> The CLIENT PROXY pattern (240) [VKZ04] offers clients a local interface as a remote component with which they interact. Clients can access the remote component in a location-independent manner, as if it were collocated with the client.
>
> The REQUESTOR pattern (242) [VKZ04] encapsulates the details of client-side remote communication, such as marshaling and sending a request across the network, and allows clients to access remote components in a location-independent manner.
>
> The CLIENT REQUEST HANDLER pattern (246) [VKZ04] encapsulates the details of client-side interprocess communication behind a uniform interface.
>
> The SERVER REQUEST HANDLER pattern (249) [VKZ04] encapsulates the details of server-side interprocess communication behind a uniform interface.
>
> The INVOKER pattern (244) [VKZ04] shields a server component application from dealing with networking issues, such as receiving, demarshaling, and dispatching requests, when a request arrives from a remote client.

CLIENT PROXY, REQUESTOR, CLIENT REQUEST HANDLER, SERVER REQUEST HANDLER, and INVOKER are themselves refined by several other distribution infrastructure patterns. Again, we do not describe these other patterns in detail, but refer you to their original source, *Remoting Patterns* [VKZ04].

Readers familiar with the first volume of the POSA series, *A System of Patterns*, might notice that the CLIENT-DISPATCHER-SERVER and FORWARDER-RECEIVER patterns [POSA1] are missing in the list of patterns above. We omit these two patterns because their responsibilities are better covered by other patterns in our language. Our experience also revealed that these patterns were too broad in scope, which suggested refactoring them into multiple smaller, more focused patterns. The responsibilities of CLIENT-DISPATCHER-SERVER are thus addressed by a BROKER configuration that uses a LOOKUP (495) service, and a CLIENT REQUEST HANDLER (246) and SERVER REQUEST HANDLER (249) association forms a FORWARDER-RECEIVER arrangement.

The diagram below illustrates how the BROKER pattern connects with other patterns in our pattern language for distributed computing



The diagram above and the earlier one on page 217 show the relationship of PUBLISHER-SUBSCRIBER with MESSAGING and BROKER. PUBLISHER-SUBSCRIBER can be viewed as a specialized form of these two patterns, since its anonymous and asynchronous group communication model can be implemented using either MESSAGING or BROKER. We present it as a separate pattern in our pattern language because it addresses a different set of forces than MESSAGING or BROKER. These forces result in more loosely coupled—and often more scalable—communication between distributed application components. Describing PUBLISHER-SUBSCRIBER as a separate pattern enables us to discuss these forces

and consequences more prominently and explicitly than if it were presented as a special case of other patterns.

We recognize that our distribution infrastructure patterns do not enable fully location-transparent communication in a distributed system [WWWK96]. While middleware platforms based on Broker, Messaging, and/or Publisher-Subscriber can off-load many tedious and error-prone network programming tasks from applications, they are ultimately just connectors between components of a distributed application. These components must therefore be prepared to handle certain challenges themselves.

For example, a distributed application must be resilient against cases of non-maskable network failures or server crashes. Likewise, a client that invokes services on a remote component must consider the latency and jitter that the network adds to remote communication. In addition, distribution infrastructure alone cannot solve problems resulting from a suboptimal deployment of application components across a network, or from inappropriate orchestration of their cooperation. The particular deployment of the components of a distributed system, as well as the way these components handle network failure, latency, and jitter, thus has a significant impact on this system's stability, performance, and scalability [DBOSG05].

In other words, middleware platforms based on Messaging, Broker, or Publisher-Subscriber are an important part of a distributed system, but cannot handle responsibilities that are application-specific and thus out of scope. Components of a distributed system must be specified thoughtfully—always keeping in mind the properties of the network—even if Messaging, Broker, or Publisher-Subscriber middleware allows them to be independent of the specific location of other components.

Messaging **

When deploying a DOMAIN MODEL (182), or a PIPES AND FILTERS (200) arrangement, to multiple processors or network nodes . . .

. . . we often need a communication infrastructure that integrates independently developed services into a coherent system.

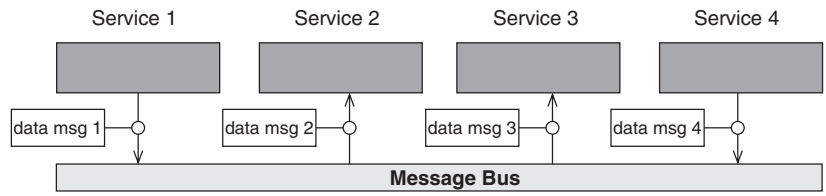
♦♦♦

Some distributed systems are composed of services that were developed independently. To form a coherent system, however, these services must interact reliably, but without incurring overly tight dependencies on one another.

Application integration is a key technique for composing solutions—often at the enterprise level—from existing, self-contained, special-purpose services. Each service provides its own business logic and value, but together they can provide the business processes and value chain of an entire enterprise. Integrating independent services into a coherent application naturally requires reliable collaboration with one another. Since services are developed independently, however, they are generally unaware of each other's specific functional interfaces. Each service may also participate in multiple integration contexts, so using it in a specific context should not preclude its use in other contexts.

Therefore:

Connect the services via a message bus that allows them to transfer data messages asynchronously. Encode the messages so that senders and receivers can communicate reliably without having to know all the data type information statically.



The services that form the distributed system connect with the message bus to exchange data messages with other services. Clients can initiate collaborations with remote services by sending them data messages asynchronously. The remote services process the received messages and return their responses—if there are any—asynchronously to the clients via messages containing the processing results. The messages are often *self-describing*: they contain both metadata that describes the message schema, and the values corresponding to the schema.

♦♦♦

Middleware based on MESSAGING enables services in a distributed application to interact without having to deal with remoting concerns by themselves, and without depending on statically defined service interfaces and data structures. In addition, the asynchronous nature of MESSAGING communication allows distributed application services to handle multiple requests simultaneously without blocking, as well as participate in multiple application integration and usage contexts. MESSAGING thus enables loose coupling, which is a key to *Enterprise Application Integration* (EAI) [Lin03] and *Service-Oriented Architectures* (SOA) [Kaye03]. The primary drawbacks of MESSAGING are its lack of statically typed interfaces, which makes it hard to validate system behavior prior to runtime, and the potential for high time and space overhead necessary to process self-describing messages [Bell06].

Data exchanged between application services are often encapsulated inside MESSAGES (420). A message hides the concrete data format of its contents from both the sender and the receiver, as well as from the MESSAGING middleware itself, which enables the transparent modification of its format. XML is a popular format for representing both the metadata and data values of self-describing messages. It enables clients to generate and send messages whose form and content need not be fixed statically.

MESSAGING clients only know the endpoints of the services they use, not their specific interfaces. Consequently the form and content of a message cannot be checked statically on the client before sending it to a specific service. Instead, the service that *receives* a message is responsible for understanding the message's form and content. This process typically involves parsing the message dynamically to validate

and extract its contents. If the service does not understand some of the message fields, it can simply ignore them, thereby simplifying the integration of services whose message formats were not originally designed to work together.

A concrete MESSAGING arrangement typically consists of several specialized parts. MESSAGE CHANNELS (224) support point-to-point communication between interacting remote services and enable reliable message exchange. MESSAGE ENDPOINTS (227) connect application services with the MESSAGING middleware: these can send and receive messages without depending on concrete messaging APIs, which enables the transparent exchange and evolution of the underlying MESSAGING infrastructure.

If the sender and the receiver of a message do not share a common message format, a MESSAGE TRANSLATOR (229) can convert messages issued by the sender into a format understood by the receiver. If the sender does not know where to address a message, a MESSAGE ROUTER (231) can help to direct it to its intended receiver. A communication failure that cannot be handled internally by the MESSAGING middleware can be returned as a REMOTING ERROR [VKZ04] to the client that sent the message.

Message Channel **

When developing a MESSAGING (221) infrastructure or a CLIENT REQUEST HANDLER (246) and SERVER REQUEST HANDLER (249) in a BROKER (237) arrangement . . .

. . . we must provide a means to connect a set of clients and services that communicate by sending and receiving messages.

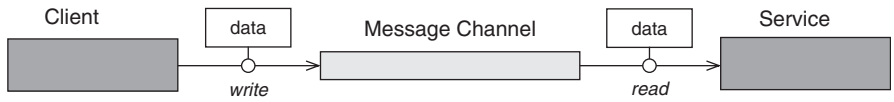
♦♦♦

Message-based communication supports loose coupling between services in a distributed system. Messages only contain the data to be exchanged between a set of clients and services, however, so they do not know who is interested in them.

Loose coupling makes it easier to integrate diverse information systems, but somehow the loose ends must tie back together. It is not sufficient for a client to send messages randomly while other services randomly receive whatever messages they come across. A client that sends out messages knows what sort of information these messages contain and often also knows who it wants to receive the messages. Similarly, services that receive messages look for particular messages they can process, and often for messages from specific senders. In other words, clients and services need to exchange messages in predictable and reliable ways.

Therefore:

Connect the collaborating clients and services using a message channel that allows them to exchange messages.



When a client has a message to communicate, it writes that message to the message channel. Services interested in the message can pick it up from there and process it.

♦♦♦

A message channel connects a set of interacting clients and services, thereby allowing them to exchange messages in a well-defined and reliable manner. Clients that write messages to the channel can be sure that the services reading the messages from the channel are interested in the information they contain, while services that read messages are sure they have received information that they can use and process.

A message channel is thus a logical address to which clients and services can write messages and/or from which they can receive messages. Several types of message channels are common. A Point-To-Point Channel [HoWo03] connects exactly one client and one service and ensures that only they can read the messages written to it. In contrast, a Publish-Subscribe Channel [HoWo03] enables publishers to broadcast messages to multiple subscribers using the Publisher-Subscriber pattern (234).

An Invalid Message Channel [HoWo03] decouples the handling of erroneous messages separately from the rest of the application logic, whereas a Dead Letter Channel [HoWo03] handles messages that were sent successfully but which could not be delivered. Finally, a Datatype Channel [HoWo03], ensures that all messages on a channel are of the same type, which helps to reduce message validation overhead in the intended receiver.

A message channel is shared by at least two concurrent entitles: a client that sends messages to the channel, and a service that obtains messages from the channel. Depending on a channel's implementation and use, therefore, it may require synchronization. A Thread-Safe Interface (384) enforces synchronization at the channel's interface, and a realization as a Monitor Object (368) supports cooperative concurrency control for simultaneous access to the channel.

In general, the operational requirements of a distributed application determine which specific message channel configuration is most appropriate. For example, information assurance requirements may dictate separate Secure Channels [SFHBS06] for selected security-sensitive collaborations. Performance and scalability requirements may equally dictate separate message channels for each type of message, or even for each use case.

A message channel does not come without cost, however, since it needs memory, networking resources, and persistent storage to support Guaranteed Delivery [HoWo03]. Developers must therefore plan and configure the number and types of message channels explicitly and thoughtfully to ensure the desired quality of service in a given system deployment. A well-designed set of message channels forms a Message Bus [HoWo03] that acts like a messaging API for the clients and services in the distributed system.

Clients and services that are not designed to use a message channel or message bus can connect to it via a Channel Adapter [HoWo03]. A Messaging Bridge [HoWo03] helps to connect clients and services that are designed to use different channel or bus implementations.

Message Endpoint **

When developing a MESSAGING (221) infrastructure . . .

. . . we must enable clients and services in an application to send and receive messages.

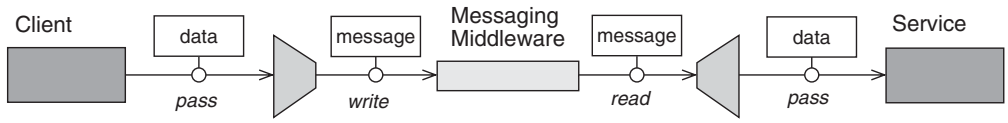
♦♦♦

Clients and services in a stand-alone application usually collaborate by passing data to one another. When clients and services are connected by a messaging infrastructure, however, such direct collaboration is impossible: data must be transformed into messages and vice versa.

Performing the data-to-message transformation directly within the applications would tightly couple them with the specific message format required by the messaging middleware. It would therefore be hard to use the services in other applications, and the mix of domain-specific code with infrastructure code would complicate their evolution and maintenance. Even if messaging is incorporated as a fundamental part of the application, replacing the underlying messaging infrastructure is time-consuming, tedious, and error-prone.

Therefore:

Connect the clients and services of an application to the messaging infrastructure using specialized message endpoints that allow clients and services to exchange messages.



When a client has data to communicate, it passes this data to its associated message endpoint, which first converts the data into a message understood by the messaging middleware, then sends that message to an endpoint representing the message's receiver. This endpoint converts the message into data that is understood by the receiver service and passes the data to that service in an appropriate format.

♦♦♦

Message Endpoints encapsulate the messaging middleware from the application clients and services and customize the middleware's general messaging API for them. Modifications to the messaging API, and even an exchange of the entire messaging infrastructure, can therefore be transparent to applications. In addition, all necessary changes are localized within the endpoints.

In general, a messaging endpoint should be designed as a Messaging Gateway [HoWo03], to encapsulate the messaging-specific code and expose a domain-specific interface to the service it represents. Internally, the endpoint can deploy a Messaging Mapper [HoWo03] to transfer data between the service and the messages. To provide asynchronous access to a synchronous method, a message endpoint can be structured as a Service Activator [HoWo03]. A Transactional Client [HoWo03] allows a message endpoint to control transactions explicitly in the messaging middleware.

Message endpoints can select among several different approaches for receiving messages. A Polling Consumer [HoWo03] provides a *proactive* message reception strategy that reads messages only when the represented service is ready to consume them. In contrast, an Event-Driven Consumer [HoWo03] supports a *reactive* message reception strategy that processes a message immediately upon arrival. If a service implements stateless functionality, the message endpoint can be a Competing Consumer [HoWo03], to allow multiple service instances to process messages concurrently. A Message Dispatcher [HoWo03] helps to dispatch incoming messages to the 'right' recipient if several services share the same message endpoint.

Designing a message endpoint as a Selective Consumer [HoWo03] enables the filtering of incoming messages: a service only processes messages that comply to the filter's criteria. A message endpoint can also be a Durable Subscriber [HoWo03], so that messages received while the represented service is unavailable are not lost. Finally, an endpoint realized as an Idempotent Receiver [HoWo03] can handle messages that were accidentally received multiple times.

The type and functionality of the represented service generally dictates which of the message reception strategies outlined above are most appropriate for a specific endpoint. The development of the message endpoint can be customized for 'its' service.

Message Translator **

When developing a MESSAGING (221) infrastructure . . .

. . . we often must transform messages from the format delivered by the client to the format understood by the service that receives them.

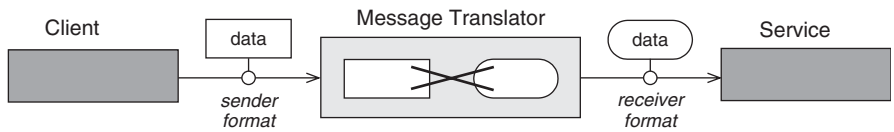
♦♦♦

Messages enable a loosely coupled style of communication between an application's clients and services. As a consequence of this decoupling, however, the client that sends a message cannot assume that the services that receive it understand the same message format.

In complex integration scenarios in which existing and independently developed components are composed into new applications, it is likely that many services will require a specific message format. Resolving such a 'Tower of Babel' confusion of 'languages' inside the services would introduce explicit and mutual dependencies between them, which contradicts the idea of loose coupling and degrades the benefits of message-based communication. Unifying the message formats across all services is often infeasible, however, because it can degrade their usability in other applications and integration scenarios.

Therefore:

Introduce message translators between clients and services of an application that convert messages from one format into another.



A message translator provides a bidirectional translation of message formats. In a specific collaboration, clients can send messages in any format they use. The message translator ensures that services get these messages in the formats they understand.

♦♦♦

A Message Translator maintains the loosely coupled style of communication introduced by Messaging even if the clients and services of an application do not share a common message format. In addition, all message transformation code is localized within a dedicated entity. This design supports evolution that is independent of, and transparent to, the clients and services that exchange messages via the translator.

In many integration scenarios, message exchange can be supported by placing specific requirements on the format and contents of a message header. An Envelope Wrapper [HoWo03] helps encapsulate the message payload so that it complies with the format required by the messaging infrastructure. When the message arrives at its destination the payload can be unwrapped. A Content Enricher [HoWo03] is needed if the target service requires data fields in a message that the originating client cannot supply: it has the ability to locate or compute the missing information from the available data. The opposite action—removing unneeded data from a message—is supported by a Content Filter [HoWo03]. A Claim Check [HoWo03] is similar to a Content Filter, but stores the removed data for later retrieval. A Normalizer [HoWo03] helps convert multiple different message format into one common format, and a Canonical Data Format [HoWo03] that is independent of any specific service can be used inside the messaging middleware to minimize the message transformations within an application.

Message Router **

When developing a MESSAGING (221) infrastructure . . .

. . . we must select a route to propagate messages through a system from their source to their destination.

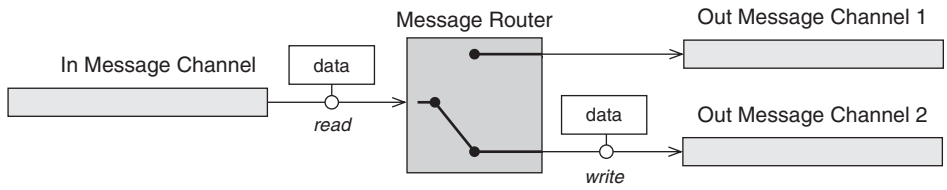
♦♦♦

Messages exchanged between collaborating clients and services must be routed through the messaging infrastructure. None of these entities, however, should have knowledge about the routing path to choose.

Making application clients and services responsible for determining the paths that messages should take through the system is not an effective solution to the routing problem, nor should they have to redirect messages they receive that were not intended for them. Such designs would tightly couple application code with infrastructure code, with clients and services depending on the internal structure and configuration of the messaging infrastructure, thereby causing maintenance problems when changes occurred. Similarly, making messages responsible for their own routing introduces the same problems for the data exchanged between collaborating components.

Therefore:

Provide message routers that consume messages from one message channel and reinsert them into different message channels, depending on a set of conditions.



A message router connects a set of message channels to a message channel network. Messages it reads from one channel are routed to a different channel that directs them to their intended receiver.

♦♦♦

The key benefit of a MESSAGE ROUTER is that it maintains the decision criteria for the destination of messages in a designated location, separate from application clients, services, and the data they exchange. If new message types are defined, the routing criteria within the message router can be modified easily and locally. If necessary, new message routers can be inserted into the messaging middleware. A message router thus increases the options developers have to send messages between application clients and services, as well as to change routing strategies independently of, and transparently to, applications.

The downside of MESSAGE ROUTER is that it adds extra processing steps to an application, which may degrade its performance. A message router also must know all the message channels it connects, which may become a maintenance problem if configurations change frequently. Moreover, the more message routers a system contains, the harder it is to analyze and understand the overall flow of messages through the system without additional tools.

Developers must therefore plan and configure the number and types of message routers carefully to meet quality of service requirements in a given system deployment. The more message routers a system configuration contains, the more flexibly messages can be routed between the components of the application, but the less efficient the message exchange becomes. In general, therefore, select the minimal set of message routers that meet application requirements to balance the needs for simplicity, flexibility, and quality of service.

There are many types of message routers. A CONTEXT-BASED ROUTER [HoWo03] bases its routing decisions on environmental conditions, such as system load, failover scenarios, or the need for a system monitoring DETOUR [HoWo03]. A CONTENT-BASED ROUTER [HoWo03], in contrast, determines a message's destination using specific message properties such as their type or content. A MESSAGE FILTER [HoWo03] assists a CONTENT-BASED ROUTER by discarding messages that do not match the routing criteria, and a RECIPIENT LIST [HoWo03] determines a list of recipients from the messages it receives. A PROCESS MANAGER [HoWo03] routes messages based on intermediate results it receives in response to previously routed messages. A MESSAGE BROKER [HoWo03] provides a central hub-and-spoke architecture for routing messages throughout an application.

Additional message routers help in managing messages exchanged between components. A SPLITTER [HoWo03] converts a single large message into several smaller messages that can be routed individually. An AGGREGATOR [HoWo03] provides the opposite functionality, by integrating multiple messages into a single message. A RESEQUENCER [HoWo03] helps to collect and reorder out-of-sequence messages so that they can be republished in the correct order. A ROUTING SLIP [HoWo03] adds explicit routing information to a message before it is sent to its receiver. A COMPOSED MESSAGE PROCESSOR [HoWo03] splits a message into multiple parts using SPLITTER, performs some processing on each message part, and reassembles the parts into a single message via AGGREGATOR before directing it to an output channel. Finally, a SCATTER-GATHER [HoWo03] broadcasts a message to multiple recipients and creates a single aggregated response message from the individual responses of each recipient.

There are two general options for implementing the control logic of a message router: it may be either statically or dynamically configurable. Statically configured message routers have less runtime overhead but are less flexible; dynamically configured message routers have the inverse properties. Dynamic configuration can be realized with help of a central CONTROL BUS [HoWo03], or by implementing the router as a DYNAMIC ROUTER [HoWo03] that configures itself based on control messages from potential message recipients.

A message router that receives messages from multiple input channels must be synchronized. A THREAD-SAFE INTERFACE (384) enforces synchronization at the router's interface, while realization as a MONITOR OBJECT (368) supports cooperative concurrency control of all message channels, enabling the router to receive messages simultaneously.

Publisher-Subscriber **

When deploying a DOMAIN MODEL (182) to multiple processors or network nodes . . .

. . . we often need an infrastructure that allows application components to notify each other about events of interest.

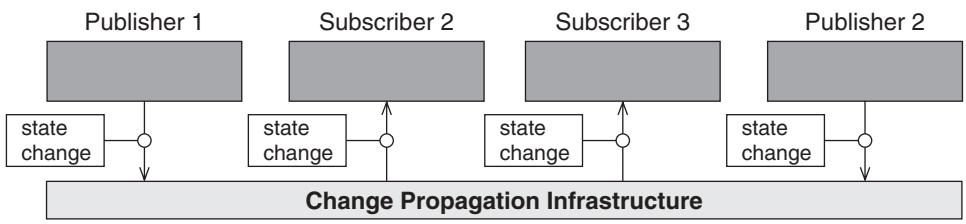
♦♦♦

Components in some distributed applications are loosely coupled and operate largely independently. If such applications need to propagate information to some or all of their components, however, a notification mechanism is needed to inform the components about state changes or other interesting events that affect or coordinate their own computation.

Nevertheless, this notification mechanism should not couple application components too tightly, or they will lose their independence. Such application components only want to know that another component in the system is in a specific state, not which specific component is involved. Similarly, components that disseminate events often do not care which other components want to receive the information. In addition, components should not depend on how other components can be reached, or on their specific location in the system.

Therefore:

Define a change propagation infrastructure that allows publishers in a distributed application to disseminate events that convey information that may be of interest to others. Notify subscribers interested in those events whenever such information is published.



Publishers register with the change propagation infrastructure to inform it about what types of events they can publish. Similarly, subscribers register with the infrastructure to inform it about what types of events they want to receive. The infrastructure uses this registration information to route events from their publishers through the network to interested subscribers. Subscribers receiving events from the infrastructure can use information in the events to guide or coordinate their own computation.

♦♦♦

Like MESSAGING, PUBLISHER-SUBSCRIBER supports *asynchronous communication*, in which publishers transmit events to subscribers without blocking to wait for a response. Asynchrony decouples publishers and subscribers so that they can be active and available at different points in time, and also leverages the parallelism inherent in a distributed system. In addition, PUBLISHER-SUBSCRIBER allows components in an application to coordinate their computation *anonymously* without introducing explicit dependencies to one another: they are unaware and independent of each other's location and identity, since they only send and receive events about changes of their state and/or the changed state itself.

Only the infrastructure has the knowledge of how the components connect, where they are located, and how events is routed through the system. PUBLISHER-SUBSCRIBER also supports *group communication*, in which publishers of events need not inform each subscriber explicitly, and the infrastructure forwards the events to all interested subscribers.

A drawback of anonymous communication is that it can cause unnecessary overhead if subscribers are interested in a specific type of event, and will only react if the event's content meets specific criteria. One way to address this problem is by filtering based on the type or content of events. Filtering can incur other costs, however. For example, filtering inside PUBLISHER-SUBSCRIBER middleware decreases its throughput, filtering within the subscribers can result in unnecessary notifications, and filtering inside publishers can break the anonymous communication model.

The information exchanged between the components connected by PUBLISHER-SUBSCRIBER middleware is encapsulated inside events, which

are realized as Messages (420). An event hides its concrete message format from both the publisher and subscriber(s), as well as from the Publisher-Subscriber middleware itself, which enables transparent modification of the message's format.

Publisher-Subscriber middleware can be implemented in various ways. One approach involves the reuse of Messaging and Broker middleware. For example, Publisher-Subscriber middleware has been implemented on top of Messaging and Broker middleware, as is the case with many WS-NOTIFICATION [OASIS06c] [OASIS06c] and CORBA Notification Service [OMG04c] products, respectively. Another approach is to implement Publisher-Subscriber using fundamental concurrency and network programming patterns [POSA2], as is the case with many DDS [OMG05b] products. In general, the former approach simplifies the efforts of the middleware developers, whereas the latter approach yields better performance.

To support anonymous and asynchronous group communication, a set of Publish-Subscribe Channels [HoWo03] or Event Channels [HV99] can broadcast or multicast event messages from publishers to subscribers. Components can inform a specific channel about which events they publish and which events they would like to receive. Event-Driven Consumers [HoWo03] support the transparent adaptation of a consumer to a specific notification publish/subscribe APIs. Such a design enables transparent exchange and evolution of the underlying Publisher-Subscriber infrastructure. Designing subscribers as Selective Consumers [HoWo03] enables the filtering of incoming event messages: a subscriber only processes events whose content complies with the filter's criteria.

If the publisher and subscriber of an event do not share a common message format, a Message Translator (229) can convert events issued by a publisher into the format understood by its subscribers. Message Routers (231) help maintain information about how to route events through the middleware to their registered subscribers. A communication failure that cannot be handled internally by the Publisher-Subscriber middleware can be returned as a Remoting Error [VKZ04] to the publisher that sent the event.

Broker **

When deploying a DOMAIN MODEL (182) to multiple processors or network nodes . . .

. . . we often need a communication infrastructure that shields applications from the complexities of component location and IPC.

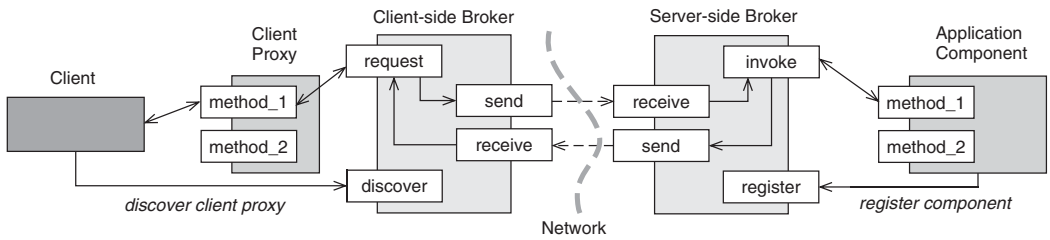
♦♦♦

Distributed systems face many challenges that do not arise in single-process systems. Application code, however, should not need to address these challenges directly. Moreover, applications should be simplified by using a modular programming model that shields them from the details of networking and location.

Sending requests to services in distributed systems is hard. One source of complexity arises when porting services written in different languages onto different operating system platforms. If services are tightly coupled to a particular context, it is time-consuming and costly to port them to another distribution environment or reuse them in other distributed applications. Another source of complexity arises from the effort required to determine where and how to deploy service implementations in a distributed system. Ideally, services should interact by calling methods on one another in a common, location-independent manner, regardless of whether the services are local or remote.

Therefore:

Use a federation of brokers to separate and encapsulate the details of the communication infrastructure in a distributed system from its application functionality. Define a component-based programming model so that clients can invoke methods on remote services as if they were local.



At least one broker instance is defined per participating network node. Component interfaces and locations are registered with their local broker to gain visibility within the distributed system. To invoke functionality on a component, clients ask their local broker for a proxy, which acts as the remote component's surrogate. A client calls a method on the proxy to initiate a request to a component. The proxy collaborates with the client and server-side brokers to deliver the request to the component and receive any results.

♦♦♦

A BROKER enables components of a distributed application to interact without handling remoting concerns by themselves. It can also optimize communication mechanisms, such as using remote method invocation versus collocated method calls, depending on the location of client and server components. For example, if the client and component are in the same address space, the broker can optimize the communication path to alleviate unnecessary overhead.

Most BROKER realizations are based on a LAYERS (185) architecture to manage complexity, such as CORBA [OMG04a] and Microsoft's .NET Remoting [Ram02]. These layers are further decomposed into 'special-purpose' components for specific networking and communication tasks. We illustrate this partitioning using the CORBA layering [SC99]—other layering schemes and middleware may involve different assignments [VKZ04].

An OS adaptation layer shields a broker from its underlying execution platform. In languages that use virtual machines, such as Java, this layer *is* the virtual machine. In other languages, such as C++, it usually contains a set of WRAPPER FACADES (459) that provide a uniform interface to specific OS APIs.

An ORB core layer forms the heart of a BROKER arrangement. In general it is a *messaging infrastructure* consisting of two components. A REQUESTOR (242) forwards request MESSAGES (420) from a client to the local broker of the invoked remote component, while an INVOKER (244) encapsulates the functionality for receiving request messages sent by a client-side broker and dispatching these requests to the addressed remote components. If a communication failure cannot be resolved internally by a BROKER infrastructure, a REMOTING ERROR [VKZ04] is signaled to the client that issued the failed request. A

COMPONENT CONFIGURATOR (490) can configure REQUESTOR and INVOKER implementations with specific communication strategies and protocols. This supports the transparent exchange and evolution of functionality within a BROKER, as well as a protocol-level integration of heterogeneous or legacy components into a distributed system.

Additional LOOKUP (495) functionality allows components to register their interfaces and location with a BROKER infrastructure. Clients can similarly use LOOKUP to find these components and the access to them. Using LOOKUP, clients need not know the concrete location of components, but can connect to them at runtime. LOOKUP also enables a flexible deployment of components, which supports both an optimal utilization of network resources and different application deployment scenarios. To complement remote procedure call invocation, a BROKER may interact with an *event channel* to support event-based notifications, which is in essence a PUBLISHER-SUBSCRIBER (234) service.

An ORB adapter typically provides a CONTAINER (488) that manages the technical environment of remote components. It interacts with a set of skeletons, which are OBJECT ADAPTERS (438) that map between the generic messaging infrastructure of the broker and the specific interfaces of remote components. A FACADE (294) presents a simple interface that components can use to access their local broker.

A CLIENT PROXY (240) represents a component in the client's address space. The proxy offers an identical interface that maps specific method invocations on the component onto the broker's message-oriented communication functionality. Proxies allow clients to access remote component functionality as if they were collocated, and can be used to implement collocation optimizations transparently [ScVi99].

Client proxies increase location-independent communication in a distributed system, but do not achieve full transparency. Before clients can use a client proxy they must obtain it from their local broker, for example via a LOOKUP. This activity is not necessary for local components unless they use a FACTORY METHOD (529) to access the components they use. In addition, client proxies may be unable to handle all REMOTING ERRORS transparently to their clients. A BUSINESS DELEGATE (292) can encapsulate such 'infrastructural concerns,' and thus help to provide more complete location-independent communication among the components of a distributed system.

Client Proxy **

When constructing a client-side BROKER (237) infrastructure, realizing PROXY-based (290) interfaces for distributed components, or implementing a BUSINESS DELEGATE (292) for a remote component ...

... we must provide an abstraction that allows clients to access remote components using remote method invocation.

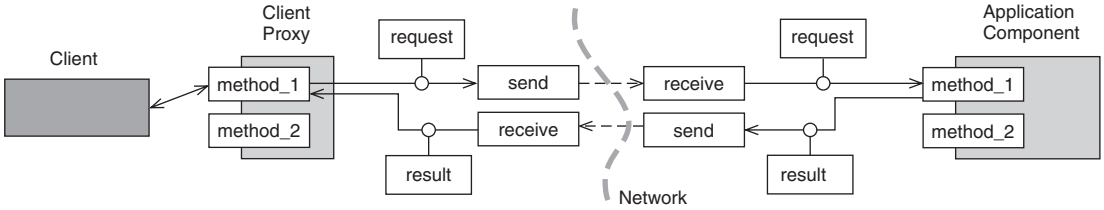
♦♦♦

Accessing the services of a remote component requires the client side to use a specific data format and networking protocol. Hard-coding the format and protocol directly into the client application, however, makes it dependent on the remoteness of its collaboration partner, because invocations on remote components will differ from invocations on local components.

Ideally, access to a component should be location-independent. There should be no functional difference between a method invocation on a local component and a method invocation on a remote component.

Therefore:

Provide a client proxy in the client's address space that is a surrogate for the remote component. The proxy provides the same interface as the remote component, and maps client invocations to the specific message format and protocol used to send these invocations across the network.



Ensure client applications issue requests to the remote component only via its client proxy. The proxy then transforms the concrete method invocation and its parameters into the data format understood by the network, and uses an IPC mechanism to send the data to the remote component. The client proxy also transforms results

returned from the component represented by the proxy back into the format understood by its clients.

♦♦♦

Client proxies support a remote method invocation style of IPC. As a result there is no API difference between a call to a local or a remote component, which enhances location-independent communication within a distributed application. In addition, a client proxy can shield its clients from changes in the represented component's 'real' interfaces, which avoids rippling effects in the case of component evolution. If the proxy interface is designed in a network-unaware style, however, it may incur excessive overhead. For example, it could offer many fine-grained methods, such as accessors for each visible attribute of the represented component. Accessing all attributes would require a client to invoke many proxy calls, each of which incurs network overhead.

In addition, client proxies can only support, but not fully achieve, location-independent communication. Before clients can use a client proxy they must obtain it from their local broker—thus clients are aware of the potential remoteness of the represented component. Moreover, a client proxy may not be able to handle all errors returned by the network transparently for its clients.

A client proxy can use a RESOURCE CACHE (505) to maintain immutable data and state of the represented remote component, once this data and state is first accessed and transferred. Caching avoids unnecessary performance penalties and network traffic for subsequent accesses to the data and state. If the immutable state and data is encapsulated within IMMUTABLE VALUES (396), the client proxy can pass it directly to clients. A client proxy can also use AUTHORIZATION (351) to enforce access rights to the remote component on the client side, which helps minimize unnecessary performance penalties and network traffic if access is denied.

Designing the client proxy as a REMOTE FACADE [Fow03a] helps to address performance problems by coalescing related fine-grained methods into a single coarser-grained method, such as a method that returns all visible attributes of the represented component in response to a single call.

Requestor **

When constructing a client-side BROKER (237) infrastructure . . .

. . . we must provide a means for sending method invocations over the network to remote components.

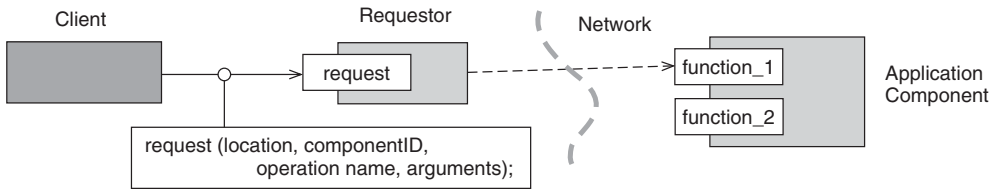
♦♦♦

The client-side invocation of a method on a remote component involves many administrative and infrastructure tasks. Implementing these tasks repeatedly within each client is tedious, error-prone, and pollutes application code with infrastructure code that may be non-portable.

Invoking a method on a remote component requires the client side to marshal invocation information, manage network connections, transmit the invocation over the network, and handle invocation results and errors. These activities are unnecessary for local method invocations. If client applications handle these issues directly, they can become dependent on specific networking protocols and IPC mechanisms, thereby decreasing their portability and reusability in other deployment scenarios and applications. Moreover, client developers would be distracted from their primary tasks: implementing application functionality correctly and efficiently.

Therefore:

Create a requestor that encapsulates the creation, handling, and sending of request messages to remote components.



Clients that want to access a remote component supply the requestor with information about the component's location, a reference to the component, the operation to be invoked, and its arguments. The requestor uses the information to construct a corresponding request message and send it over the network to the remote component.

♦♦♦

A requestor shields application logic in a distributed system from the details of client-side networking and IPC activities and tasks.

A requestor can delegate some of its sub-activities to other components. A MARSHALER [VKZ04] serializes concrete service requests into request MESSAGES (420) and de-serializes corresponding result messages into concrete responses. A CLIENT REQUEST HANDLER (246) manages connections and encapsulates specific IPC mechanisms, thereby simplifying sending request messages across a network and receiving result messages.

Some applications require additional requestor activities, such as adding a security token, which an INTERCEPTOR (467) can encapsulate via a uniform interface. In general, a MARSHALER, CLIENT REQUEST HANDLER, and INTERCEPTOR manipulate client-side request processing aspects without affecting the requestor's core algorithm for creating, handling, and sending requests. An ABSOLUTE OBJECT REFERENCE [VKZ04] can encapsulate information about the location and identity of the remote component that is the target of the request. Similarly, a REMOTING ERROR [VKZ04] is returned to the client when a failure cannot be handled transparently by the requestor.

In general there are three deployment options for a requestor [SMFG00]. The simplest option is to deploy one requestor for all client threads or processes on a node. The more clients access the requestor, however, the more it becomes a throughput and scalability bottleneck. To alleviate this drawback, there could be a separate requestor per client, or several clients could share a requestor. A requestor that is shared by multiple concurrent clients must be synchronized. Providing the requestor with a THREAD-SAFE INTERFACE (384) is a simple, coarse-grained synchronization option, because it enforces synchronization at the interface of the requestor, even if only small portions of its methods are critical sections. In this case, synchronization via STRATEGIZED LOCKING (388) is an alternative. Realizing a requestor as a MONITOR OBJECT (368) supports cooperative concurrency control of multiple clients that access the requestor simultaneously. If there are multiple requestors per client application, they must synchronize their internal use of shared resources, such as connections or cached request objects.

Invoker **

When constructing a server-side BROKER (237) infrastructure . . .

. . . we must provide a means for receiving method invocations from the network and dispatching them to remote components.

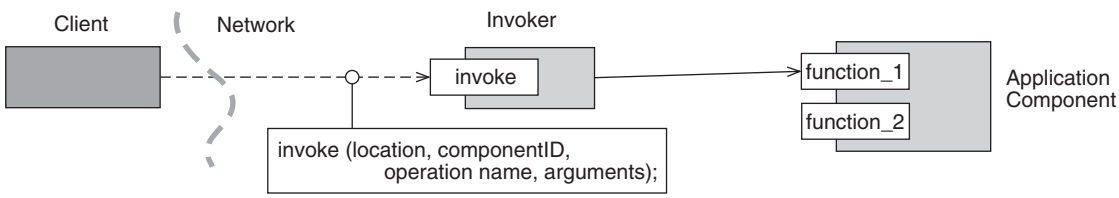
♦♦♦

The server side must perform many administrative and infrastructure tasks to transform data received from clients into an invocation on a specific method of a remote component. Implementing these tasks repeatedly within each component implementation is tedious, error-prone, and pollutes application code with infrastructure code that is often non-portable.

Invoking a specific method of a component implementation in response to a client request requires a server to manage network connections, receive data on the connections, demarshal that data to receive the associated invocation information, identify the intended component implementation, invoke the appropriate method on that component, and return its results or errors. If remote components handled these concerns directly, they would be tightly coupled to specific networking protocols and IPC mechanisms, thereby decreasing their portability and reusability in other deployment scenarios and applications. Moreover, server developers would be distracted from their primary tasks: implementing application functionality correctly and efficiently.

Therefore:

Create an invoker that encapsulates the reception and dispatch of request messages from remote clients in a specific method of a component implementation.



An invoker listens to network connections for request messages to arrive from remote clients, receives the request messages when they arrive, demarshals the received information to determine which method and parameters to invoke on which component implementation, and dispatches that method on the identified component.

♦♦♦

An invoker shields the application logic of a distributed system from the details of server-side networking and IPC tasks and activities.

An invoker can delegate several sub-activities of its responsibility to other components. A MARSHALER [VKZ04] can de-serialize request MESSAGES (420) into a concrete service and serialize corresponding services results into result messages. A SERVER REQUEST HANDLER (249) manages connections and encapsulates a specific IPC mechanism, thereby simplifying the process of receiving requests and sending results across a network.

Some applications also require additional invocation activities, such as interpreting an embedded security token, which an INTERCEPTOR (444) can encapsulate via a uniform interface. In general, a MARSHALER, SERVER REQUEST HANDLER, and INTERCEPTOR manipulate server-side request processing aspects without affecting the invoker's core algorithm for the reception and dispatch of client requests on component implementations.

An ABSOLUTE OBJECT REFERENCE [VKZ04] encapsulates the identity of a specific component implementation. A REMOTING ERROR [VKZ04] is returned to the client that issued a request if failures occur that cannot be handled by the invoker. The invoker can also delegate its invocation to a LOCATION FORWARDER [VKZ04] if it cannot find the component implementation, for example if the component was redeployed, or is temporarily unavailable due to high machine load.

An invoker has several deployment options, the simplest of which is to deploy one invoker for all components implementations in a server application. The more components are accessed via one invoker, however, the more it becomes a throughput and scalability bottleneck. To alleviate this drawback, several remote components could share an invoker, or each component could have its own invoker.

Client Request Handler **

When developing a REQUESTOR (242) . . .

. . . we must send requests to, and receive replies from, the network.

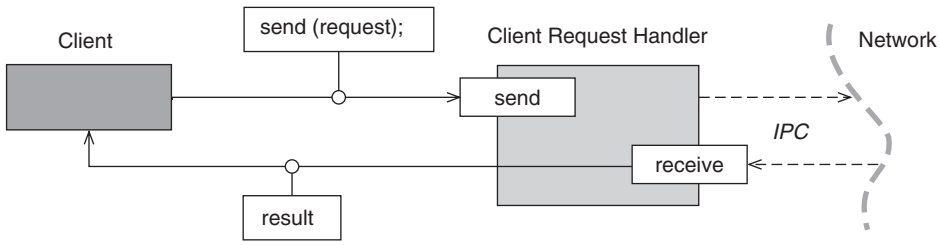
♦♦♦

Sending client requests to and receiving replies from the network involves various low-level IPC tasks, such as connection management, time-out handling, and error detection. Writing and performing these tasks separately for each client uses networking and endsystem resources ineffectively.

The more clients access the network, and the more requests and replies must be handled simultaneously, the more efficiently network resources must be managed to achieve appropriate quality of service in a distributed application. Network connections and bandwidth, for example, are limited resources and must be shared and used judiciously by all clients to ensure acceptable latency and jitter. In addition, writing error detection and time-out handling tasks separately for each client duplicates code and pollutes the application with non-portable networking code.

Therefore:

Provide a specialized client request handler that encapsulates and performs all IPC tasks on behalf of client components that send requests to and receive replies from the network.



The functional responsibilities of a client request handler include connection establishment, request sending and result dispatching, and time-out and error handling, which it performs with help of specific IPC mechanisms. In addition, the client request handler is

responsible for efficient management and utilization of networking and computing resources, such as network connections, memory, and threads.

♦♦♦

The centralized execution and management of all client-side networking activities within a CLIENT REQUEST HANDLER can improve distributed application quality of service, such as latency, throughput, scalability, and resource utilization. The encapsulation of specific IPC mechanisms makes communication transparent for clients that issue requests to remote components.

To establish concrete connections to remote components, the client request handler implements the connector role of ACCEPTOR-CONNECTOR (265), which supports the evolution of network connection establishment strategies independently of other client request handler responsibilities. If the client request handler is shared by multiple concurrent clients, the connector must be synchronized. Providing the connector with a THREAD-SAFE INTERFACE (384) is a simple but coarse-grained synchronization option, because it enforces synchronization at the interface of the connector, even if only small portions of its methods are critical sections. In this case, consider using STRATEGIZED LOCKING (388) to parameterize the synchronization mechanisms. Realizing a connector as a MONITOR OBJECT (368) supports cooperative concurrency control when multiple clients access the connector simultaneously.

A connection created by a connector can be encapsulated within a connection handler that plays the service handler role in ACCEPTOR-CONNECTOR. This design treats a connection as a first-class entity, which supports efficient maintenance of connection-specific state, as well as the handling of REMOTING ERRORS [VKZ04] that occur on the connection. Scalability is also supported, since each connection handler can run in its own thread, thereby processing requests from and replies to multiple clients simultaneously.

A connection handler can implement a synchronous or asynchronous communication strategy. Synchronous communication can simplify the client programming model, but reduces performance and throughput, whereas asynchronous communication has the inverse properties. Four patterns—FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT,

and Result Callback [VKZ04]—help realize an asynchronous communication model. These four patterns provide different strategies for addressing the following three aspects: whether or not a result is sent to the client, whether or not the client receives an acknowledgement, and, if a result is sent to the client, whether it is the client's responsibility to obtain the result or whether it is informed using a callback.

If a client expects a result or an acknowledgement, a connection handler can use time-outs to detect potential failures of asynchronous communication. Depending on whether a connection handler processes data serially or is interrupt-driven, it can register with a Reactor (259) or Proactor (262), respectively, which will notify it when a specific result arrives.

The specific IPC mechanism used by the connector and the connection handlers of a client request handler can be encapsulated by a Protocol Plug-in [VKZ04] or a set of Wrapper Facades (459). Both patterns hide IPC mechanism details behind uniform, platform-independent interfaces. A Protocol Plug-in allows runtime (re)configuration of IPC mechanisms, but incurs some runtime overhead. Wrapper Facade, in contrast, avoids runtime overhead, but only supports compile-time configuration. The requests are encapsulated within Messages (420) and sent over the network via a Message Channel (224). If security is required, the IPC mechanism should use a Secure Channel [SFHBS06] to transmit requests.

An Object Manager (492) can enhance client request handler performance via caching. For example, if a specific connection is no longer needed, it need not be destroyed, but can instead be kept 'alive' for a predetermined time and reused for another collaboration between the client and server it connects.

Results of specific invocations, as well as any Remoting Errors [VKZ04] that cannot be resolved by the client request handler, are returned to the client that issued the corresponding request.

Server Request Handler **

When developing an INVOKER (244) . . .

. . . we must receive requests send replies across the network.

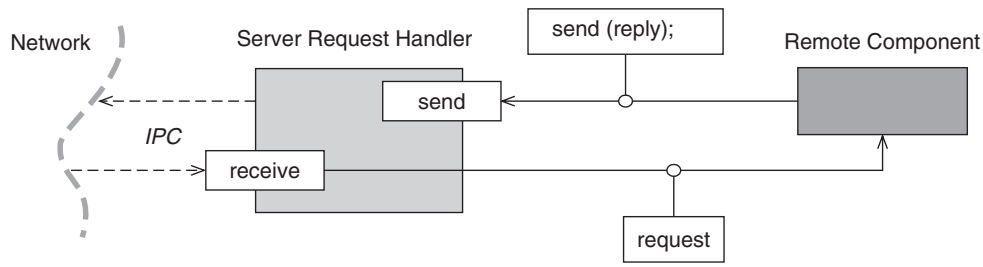
♦♦♦

Receiving client requests from and sending replies to the network involves several low-level IPC tasks, including connection management, time-out handling, and error detection. Writing and performing these tasks separately for each client uses networking and endsystem resources ineffectively.

The more remote requests and replies must be handled simultaneously by a server part of a distributed application, the more efficiently network resources must be managed and utilized to achieve an appropriate quality of service. Network connections and bandwidth, for example, are limited resources, and must be shared and used effectively by all remote components to provide an appropriate performance, throughput, and scalability on the server-side of a distributed system. In addition, writing error detection and time-out handling tasks separately for each component duplicates code and pollutes the application with non-portable networking code.

Therefore:

Provide a specialized server request handler that encapsulates and performs all IPC tasks on behalf of remote components that receive requests from and send replies to the network.



The functional responsibilities of a server request handler include connection establishment, request reception and dispatching, result

sending, and error handling, which it performs with the help of specific IPC mechanisms. In addition, the server request handler is responsible for efficient management and utilization of networking and computing resources such as network connections, memory, and threads.

♦♦♦

The centralized execution and management of all server-side networking activities within a SERVER REQUEST HANDLER can improve the distributed application's quality of service, in particular, latency, throughput, scalability, and resource utilization. The encapsulation of specific IPC mechanisms makes communication transparent for application components that receive requests from and send responses back to remote clients.

The client request handler needs an event-handling infrastructure that listens on the network for connection requests to arrive, establishes the requested connections, and dispatches service requests to methods on the appropriate application components. This infrastructure must allow multiple connection and service requests to be received and processed simultaneously to achieve appropriate latency, throughput, and scalability. Its core can be realized by a REACTOR (259) or PROACTOR (262), depending on whether the processing of received events is handled serially, or driven by interrupts, respectively.

The event handlers of a REACTOR or PROACTOR can be realized as an ACCEPTOR-CONNECTOR (265) to separate connection establishment from request and reply handling. A dedicated acceptor listens on the network for connection requests to occur, accepts these requests, and creates a connection handler that encapsulates the newly established connection. The connection handler then performs the IPC on behalf of the application component accessed via that connection. This design enables connection establishment and data transfer strategies to evolve independently of each other. Connections are also treated as first class entities, which supports efficient maintenance of connection-specific state and handling of any REMOTING ERRORS [VKZ04] that occur on the connection. In addition, scalability is supported: each connection handler can run in its own thread, which allows a server request handler to handle requests from, and replies for, multiple clients concurrently.

A connection handler can implement a synchronous or asynchronous communication strategy. Synchronous communication can simplify the client programming model but reduces performance, whereas asynchronous communication has the inverse properties. The SYNC WITH SERVER pattern [VKZ04] can help to return acknowledgements for an asynchronous communication model.

The specific IPC mechanism used by the acceptor and the connection handlers of a server request handler can be encapsulated by a PROTOCOL PLUG-IN [VKZ04] or a set of WRAPPER FACADES (459). Both patterns hide IPC mechanism details behind uniform and platform-independent interfaces. A PROTOCOL PLUG-IN allows runtime (re)configuration of IPC mechanisms, but incurs some runtime overhead. WRAPPER FACADE, in contrast, enhances performance, but only supports compile-time configuration. The requests are encapsulated within MESSAGES (420) and sent over the network via a MESSAGE CHANNEL (224). If security is required, the IPC mechanism should use a SECURE CHANNEL [SFHBS06] to transmit requests.

An OBJECT MANAGER (492) can enhance client request handler performance via caching. For example, if a specific connection is no longer needed, it need not be destroyed, but can instead be kept 'alive' for a predetermined time and reused for another collaboration between the client and server it connects.