Architectural Patterns Revisited – A Pattern Language

Paris Avgeriou
Department of Mathematics and Computing Science
University of Groningen
Groningen, the Netherlands
paris@cs.rug.nl

Uwe Zdun
Department of Information Systems
Vienna University of Economics and BA
Vienna, Austria
zdun@acm.org

Architectural patterns are a key concept in the field of software architecture: they offer well-established solutions to architectural problems, help to document the architectural design decisions, facilitate communication between stakeholders through a common vocabulary, and describe the quality attributes of a software system as forces. Regrettably, finding and applying the appropriate architectural patterns in practice still remains largely ad-hoc and unsystematic. This is due to the lack of consensus in the community with respect to the "philosophy" and granularity of architectural patterns, as well as the lack of a coherent pattern language. In this paper we attempt to establish common ground in the architectural patterns community by proposing a pattern language that acts as a superset of the existing architectural pattern collections and categorizations. This language is particularly focused on establishing the relationships between the patterns and performs a categorization based on the concept of "architectural views".

1 Motivation

The software architecture community has had many debates on various aspects like which design and evaluation methods, Architecture Description Languages, and views are best for which case. Architectural patterns were one of the few points, where consensus was achieved in the field of software architecture: their significance is well-established and they are essential to an architecture description [SC97, SG96, BCK98, CBB$^+$02, BMR$^+$96, SSRB00]. Regrettably, describing, finding, and applying architectural patterns in practice still remains largely ad-hoc and unsystematic. This is due to several issues that are still not resolved: there is a semantic gap concerning what architectural patterns really represent, what is the philosophy behind them; moreover, there is much confusion with respect to what is the granularity of architectural patterns; finally there is no accepted classification or cataloging of patterns that can be used by architects. The next paragraphs elaborate on these issues.

There are two different "schools of thought" in the literature with respect to the nature of architectural patterns: one that uses the term "architectural pattern" (e.g. in [BMR$^+$96, SSRB00, VKZ04]) and another that uses the term "architectural style" (e.g. in [SG96, SC97, BCK98, CBB$^+$02]). Both terms refer to recurring solutions that solve problems at the architectural design level, and provide a common vocabulary in order to facilitate communication. They both also accept that patterns provide the means to reason for the quality attributes of a software system and help to document the design decisions taken by the architect. But they have some key differences in their underlying philosophy:

- In the architectural patterns perspective, patterns are considered as problem-solution pairs that occur in a given context and are affected by it. Furthermore a pattern does not only document “how” a solution solves a problem but also “why” it is solved, i.e. the rationale behind this specific solution. In particular, the description of the problem pays much attention to the forces that shape the problem, while the solution elaborates on how (and if) those forces are resolved. The description of architectural patterns is based on the context-problem-solution triplet and may be further elaborated with richer details, especially focusing on the rationale behind a solution. Moreover, patterns are meant to work synergistically in the context of a pattern language, and have numerous inter-dependencies among each other. Finally, there are a number of postulations for a solution to qualify as a pattern. For instance, the pattern must capture common, successful practice (e.g. have at least three known uses), and at the same time the solution of the pattern must be non-obvious (although this relates to experience and background). Patterns should provide aesthetic solutions, and in the pattern literature the human aspect of software is accentuated [Cop96].
- In the architectural styles perspective, the problem does not receive much attention nor does the rationale behind choosing a specific solution. In [SC97, BCK98] a style is looked upon in terms of components, connectors, and issues related to control and data flow. In [SG96] attention is drawn to architectural configurations, semantics of the styles, and potential architectural analysis that can be performed on the systems built on the styles. In [BCK98, CBB$^+$02] the concept of architectural styles is treated as a set of constraints on components, connectors, and their interactions. Similarly, in [MM03], an architectural style is represented by components, connectors, their configurations, and constraints upon all of them. In this viewpoint, patterns are not considered generic and “timeless” in the Alexandrian sense [AIS$^+$77, Ale79], but become much more concrete and focused. This also leads to multiple variations of the same pattern in order to solve specialized design problems [SC97]. Also, since the pattern’s implementation details can be pinpointed, Architecture Description Languages [MT00] can be designed in order to support individual architectural patterns.

As far as the granularity of architectural patterns is concerned, the boundaries between design patterns and architectural patterns (or potentially other classifications) are unclear. Is the architectural scope of a pattern a question of scale or nature? In particular, the category of “design patterns”, e.g. as described in [GHJV94], are often referred to, or used as architectural patterns. In general it is hard to draw the line between architectural patterns and design patterns. In fact, it depends heavily on the viewpoint of the designer or architect whether a specific pattern is categorized as an architectural pattern or a design pattern. Consider, for instance, a classical design pattern, the INTERPRETER pattern [GHJV94]. The description in [GHJV94] presents it as a concrete design guideline. Yet, instances of the pattern are often seen as central elements in the architecture of software systems, because an INTERPRETER is a central, externally visible component – i.e. the pattern is treated like an architectural pattern (see [SG96]).

Finally, there is no single catalog of architectural patterns for software architects to use. Instead there is a voluminous and heterogeneous literature of patterns, where the various patterns differ in their scope, context and way of description and are often not related in the context of a pattern language. To make matters worse, many architectural pattern languages have been developed since the earlier software patterns literature [SG96, SC97, BMR$^+$96, GHJV94] has been documented, but the former are not clearly related to the latter. Of course, there have been attempts to classify architectural patterns: in [BMR$^+$96] architectural patterns are categorized according to the global system properties that they support; in [SC97, BCK98] architectural patterns are classified with respect to a framework of features, like the types of components and connectors, and control and data issues; a more recent classification scheme that has been proposed is based on the concept of architectural views [CBB+02]. But again there is no consensus on these classifications that could possibly lead to a single scheme.

2 Putting the pieces together: a pattern language

2.1 The approach

In this paper we attempt to find *common ground* in the architectural patterns community by proposing a pattern language that acts as a superset of the existing architectural pattern collections and categorizations. This language, as a whole, is greater than the sum of its parts because it particularly focuses on establishing the relationships between the patterns in order to present the "big picture". In particular this pattern language tackles the aforementioned shortcomings in the following ways:

- We consider that both architectural patterns and architectural styles are in essence the same concepts and that they only differ in using different description forms. Therefore, we put both the "classical" architectural patterns such as those from POSA (see [BMR+96]) and the architectural styles from SEI (see [SC97, Sha96, SG96, BCK98, CBB+02]) in the same pattern language, paying special attention on relating them with each other. For the sake of simplicity, we shall use only the term "architectural pattern" for the rest of this paper. Note that we are aware that some patterns described below, e.g. EXPLICIT INVOCATION, would not qualify as patterns in a strict interpretation of the pattern definition, because they may be considered as stating really obvious solutions. Naturally, the obviousness of a solution is a matter of definition. We nevertheless include them in our pattern language because they are important pieces of the architectural patterns puzzle and substantially contribute in putting the individual architectural styles and patterns together into a coherent pattern language.
- We consider a pattern to be architectural, if it refers to a problem at the architectural level of abstraction; that is, if the pattern covers the overall system structure and not only a few individual subsystems. Thus we have not included the classical "design patterns" (e.g. [GHJV94]) in our pattern language except for one (INTERPRETER). However we emphasize that these design patterns can potentially be used as architectural patterns, if one applies them at the level and scope of a system's architecture. Therefore, the architectural scope of a pattern is not only a property of the pattern per se, but also of the pattern application.
- We propose a classification of the architectural patterns, based upon architectural views, that extends the one proposed in [CBB+02]. The next subsection elaborates on this classification scheme.

This pattern language, as aforementioned, contains patterns from existing collections of architectural patterns. The emphasis of this language is therefore not on describing the individual patterns; they have already been elaborately described before. Thus we do not repeat these descriptions here. Instead emphasis is given only on the related pattern sections that analytically describe the relationships between the patterns. Consequently, the intended audience for this pattern language is comprised of architects that have a sound knowledge of these patterns and can accordingly understand the "big picture" of the inter-related patterns.

2.2 Classification of architectural patterns according to views

The classification scheme for architectural patterns that we propose is based on the concept of architectural views. An *Architectural View*[1] is a representation of a system from the perspective of a related set of *concerns* [IEE00] (e.g. a concern in a distributed system is how the software components are allocated to network nodes). This representation is comprised of a set of system *elements* and the *relationships* associated with them [CBB+02]. An *Architectural Pattern*, on the other hand, defines *types* of elements and relationships that work together in order to solve a particular problem from some perspective.

An essential issue however is how architectural views relate to architectural patterns. There have been two major approaches on this matter so far:

- The first considers that views are of large granularity in the sense that the elements and relationships are generically defined. Therefore, in such a coarse-grained view, multiple architectural patterns may be applied. For instance we can consider a *structural* view that describes how a system is structurally decomposed into components and connectors. In this view, we may apply the LAYERS and the PIPES AND FILTERS patterns. Some well-established examples of this approach are Kruchten's "4+1 views" [Kru95], the so-called "Siemens 4 views model" [HNS00], and the "Zachman framework" [Zac87]. The same thesis is supported in the IEEE 1471 Recommended Practice for Architectural Description. Even though it does not prescribe any specific set of views, it considers views of such granularity, e.g. structural and behavioral views.
- The second considers that each architectural pattern corresponds to a view in a one-to-one-mapping, and is mainly advocated in [CBB+02]. This notion of a view is of course far more fine-grained since it leads to elements and relationships of very specialized semantics, e.g. consider a pipe-and-filter view or a client-server view. These views are categorized into classes, called "view types", which are of the same granularity as the views of the first approach. For example, in [CBB+02] the Components and Connectors viewtype contains the pipe-and-filter and the client-server views.

We follow the middle path between the two aforementioned approaches. We assume that views should be more fine-grained than the first approach in order to be useful. Therefore, instead of generic structural or behavioral views, we consider views that show more specific aspects of the system like the flow of data or the interaction of components. On the other hand we consider views to be more coarse-grained than individual architectural patterns, since more than one pattern can be either complimentary or alternatively applied in a given view. For example in a view that shows how shared data is manipulated by a number of components, we could either apply SHARED REPOSITORY or ACTIVE REPOSITORY.

Our classification scheme for architectural patterns is organized around a number of the most common views that are used in practice. Therefore an architectural pattern is classified in a particular view if the application of the pattern in a system architecture is *visible* in this view. Emphasis is given on the multiple and complex inter-relationships between the patterns. In our classification scheme each pattern is assigned to one primary view, which is the most suitable. However there are some cases where a single pattern could be used in a second or third view. This can be derived as follows: when two patterns from different views are combined on the

[1]We use the term *view* to describe the different perspectives of the architecture like "structural view" or "behavioral view", instead of "structural architecture" or "behavioral architecture", in the spirit of [IEE00] which mandates that a system has *one* architecture, whose description can be organized in many views.

same system, then they can be seen in both views. For example, consider the case of a SHARED REPOSITORY which also implements a CLIENT-SERVER structure because the repository plays the role of a server satisfying the clients-accessors requests. In this case each of the two patterns is visible both in the data repository and the component interaction views.

The views that we have chosen for this classification scheme contain mainly two types of elements: *components* which are units of runtime computation or data-storage, and *connectors* which are the interaction mechanisms between components [PW92, CBB+02]. In this paper, we concentrate on patterns describing component and connector structures. There are, of course, other views that contain different kinds of elements. For instance, in [CBB+02] the "module" views deal with implementation modules (e.g. Java or C++ classes). The "allocation" views deal with how software elements are allocated to environment elements (e.g. code units are allocated to members of the development team). "Service" views abstract from the system in a service-oriented fashion. "Event handling" views explain the system in terms of events and reactions on these events. "Concurrency" views describe the concurrent behavior of system elements. These other views are as important for the architecture as the component and connector structure views, but because we want to focus on the "classical" architectural patterns and styles – which mainly describe structural properties of the system – we did not include the other views in our pattern language for the time being.

2.3 Overview of the pattern language

The following pattern language is comprised of component and connector views, and the patterns that are classified in each view. Note that we focus on "classical" architectural patterns from both POSA (see [BMR+96]) and SEI (see [SC97, Sha96, SG96, BCK98, CBB+02]), because these have been well-established in the software architecture community. We have also included a few patterns from other sources, in order to describe important links or gaps in the realm of the "classical" architectural patterns. We also discuss some links to related pattern languages inside the description of the patterns.

- The Layered Decomposition View shows how the system is horizontally partitioned into interacting parts with dependencies between them.
  - LAYERS [SC97, Sha96, SG96, BCK98, CBB+02, BMR+96]
  - INDIRECTION LAYER [Zdu04, Zdu03] (a variant of this pattern is called "virtual machine" in [CBB+02])
- The Data Flow and Transformation View shows how streams of data flow in the system and get successively processed or transformed by components.
  - BATCH SEQUENTIAL [SG96, SC97, BCK98]
  - PIPES AND FILTERS [SC97, Sha96, SG96, BCK98, CBB+02, BMR+96]
- The Data Repository View shows how multiple components access a central repository of data.
  - SHARED REPOSITORY [VKZ04] (called "repository" in [SG96, Sha96, BCK98])
  - ACTIVE REPOSITORY (called "blackboard" in [SC97, BCK98])
  - BLACKBOARD [BMR+96, SG96]

- The Adaptation Infrastructure View shows the infrastructure that a system uses to adapt itself during evolution.
  - MICROKERNEL [BMR$^+$96]
  - REFLECTION [BMR$^+$96]
  - PLUGIN [Fow02]
  - INTERCEPTOR [SSRB00]
- The Language Infrastructure View shows infrastructures for extending a system with languages, such as domain-specific languages.
  - INTERPRETER [SG96, Sha96, BCK98, GHJV94]
  - VIRTUAL MACHINE [GMSM00]
  - RULE-BASED SYSTEM [SG96, BCK98]
- The Interaction Decoupling View shows how the interactions in a system can be decoupled, for instance, how to decouple user interface logic from application logic and data.
  - MODEL-VIEW-CONTROLLER [KP88, BMR$^+$96]
  - PRESENTATION-ABSTRACTION-CONTROL [Cou87, BMR$^+$96]
  - C2 [TMA$^+$96]
- The Component Interaction View shows how individual components interact with each other but retain their autonomy.
  - EXPLICIT INVOCATION (called "communicating processes" in [SC97, Sha96, BCK98, CBB$^+$02])
  - IMPLICIT INVOCATION [Sha96, BCK98, SG96] (also called "event systems" in [SC97, SG96, BCK98])
  - CLIENT-SERVER [SC97, SG96, BCK98, CBB$^+$02]
  - PEER-TO-PEER [CBB$^+$02]
  - PUBLISH-SUBSCRIBE [BCK98, CBB$^+$02] (called "publisher-subscriber" in [BMR$^+$96].
- The Distributed Communication View shows how distributed components in a networked environment can communicate with each other.
  - BROKER [BMR$^+$96, VKZ04]
  - REMOTE PROCEDURE CALLS [VKZ04] (called "distributed objects" in [SC97])
  - MESSAGE QUEUING (called "messaging" in [HW03, VKZ04])

Figure 1 illustrates the above views, the patterns classified in each view, and the most significant relationships between the patterns. These relationships will be elaborated during the presentation of the individual views, in pattern overview diagrams.

The next sections elaborate on the various views and the patterns assigned to each view. The description of each view is presented informally with respect to the types of elements and relationships, as well as the concerns addressed by the view. Each pattern is described using a brief summary, and a more elaborate discussion of its relationships to other patterns.

This list of Component and Connector views is not exhaustive but merely contains commonly used patterns from POSA (see [BMR$^+$96]) and SEI (see [SC97, Sha96, SG96, BCK98, CBB$^+$02]). There are numerous more patterns that could be classified in more views such as:

- The Concurrency Infrastructure View that would include the ACTIVE OBJECT, MONITOR OBJECT, HALF-SYNC/HALF-ASYNC, LEADER/FOLLOWERS, and THREAD-SPECIFIC STORAGE patterns from [SSRB00].
- The Event Handling View that would include the REACTOR, PROACTOR, ASYNCHRONOUS COMPLETION TOKEN, and ACCEPTOR-CONNECTOR patterns from [SSRB00].
- The Synchronization Infrastructure View that would include the SCOPED LOCKING, STRATEGIZED LOCKING, THREAD-SAFE INTERFACE, DOUBLE-CHECKED LOCKING OPTIMIZATION.
- The Security Infrastructure view that would include the patterns from [SFHB05].

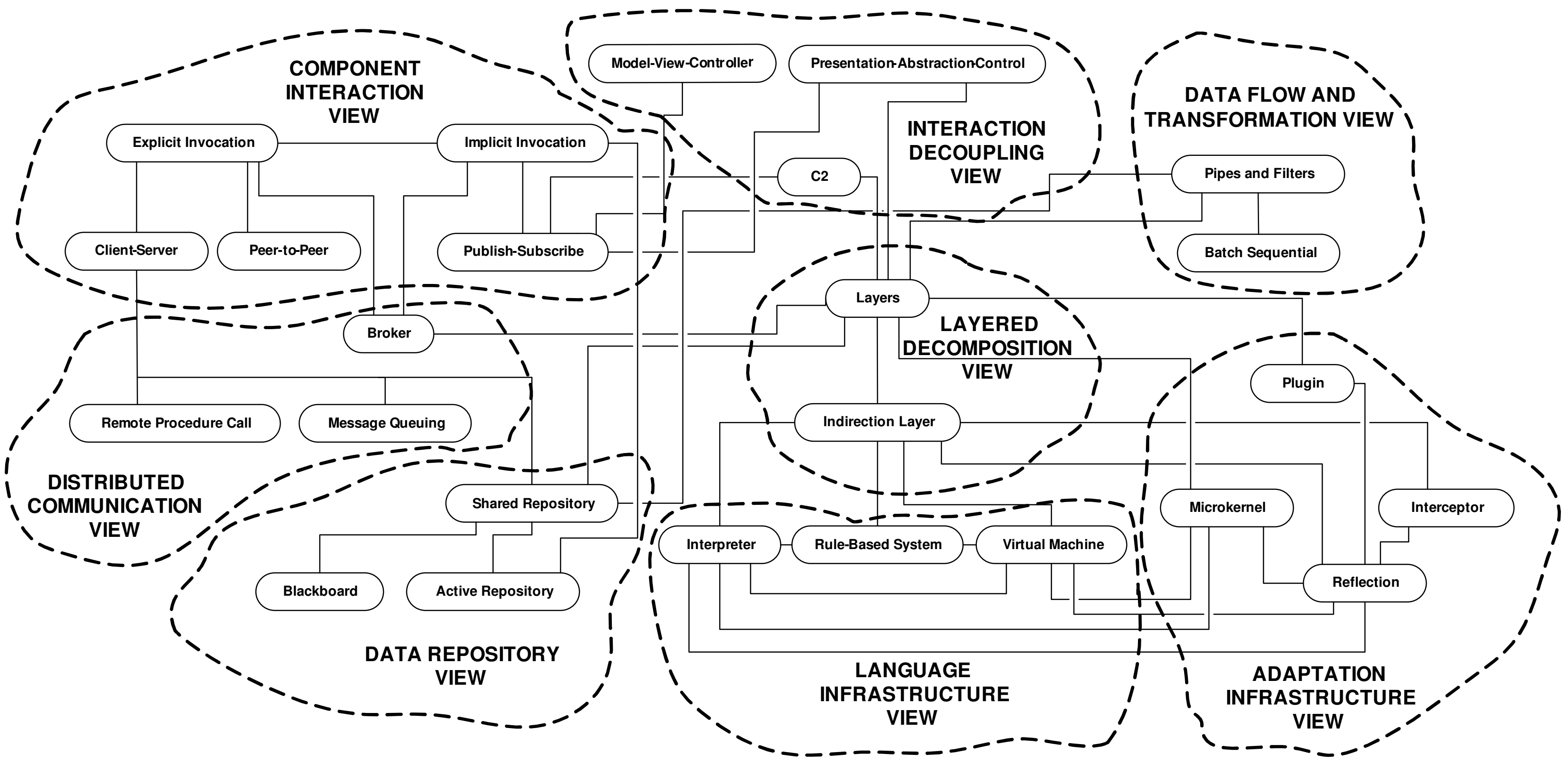

Figure 1: The patterns of the different views and the most significant pattern relationships

3 Layered Decomposition View

The Layered Decomposition View shows how the system is horizontally partitioned into interacting parts. Therefore a major concern in this view is to find the essential parts that make up the whole system, as well as the major interactions between these parts. The view shows how the parts perform their functionality and still remain decoupled from each other – in a primarily horizontal partitioning of these functionalities. The Layered Decomposition View should also explain what the criteria for separating the parts are.

The Layered Decomposition is related to a number of quality attributes. Modifiability should be considered in order to see whether a particular decomposition supports design for change. Usually it is difficult to establish the "optimal" boundaries for the horizontal partitioning. A trade-off must be achieved between the Reusability, Exchangeability, Portability, and Integrability of the parts of the system. Compared to monolithic architectures, a partitioned architecture might have a lower Efficiency due to indirections. Thus Efficiency might also be a criterion governing a partitioning.

The individual parts of the system are components that are decoupled as much as possible from one another. The interaction mechanisms between the components are implemented through connectors that include appropriate interfaces, states, and interaction protocols. In many cases, there is an overall control mechanism that maintains an overall organization scheme by orchestrating the various components.

Figure 2 illustrates the patterns and their relationships from the Layered Decomposition, Data Flow and Transformation, and Data Repository Views. In this and subsequent pattern overview diagrams, patterns which are not described in this pattern language but merely referenced, are rendered in gray.

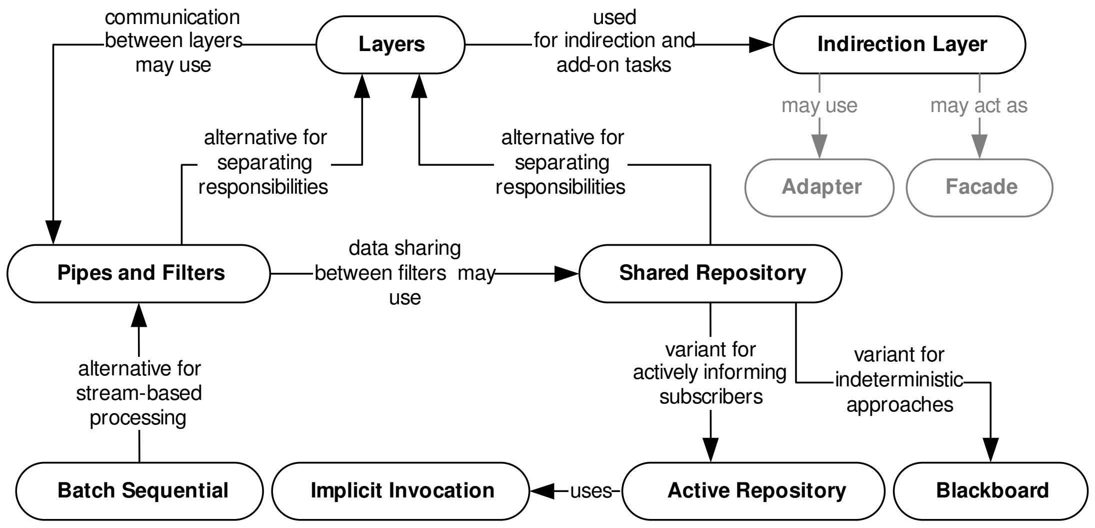
communication between layers may use
Layers
used for indirection and add-on tasks
Indirection Layer
may use
may act as
Adapter
Facade
alternative for separating responsibilities
alternative for separating responsibilities
Pipes and Filters
data sharing between filters may use
Shared Repository
alternative for stream-based processing
variant for actively informing subscribers
variant for indeterministic approaches
Batch Sequential
Implicit Invocation
uses
Active Repository
Blackboard

Figure 2: Overview: Patterns of the Layered Decomposition, Data Flow and Transformation, and Data Repository Views

Pattern: Layers

Consider a system in which high-level components depend on low-level components to perform their functionality, which further depend on even lower-level components and so on. Decoupling the components in a vertical manner is crucial in order to support modifiability, portability, and reusability. On the other hand, components also require some horizontal structuring that is orthogonal to their vertical subdivision.

To achieve these goals, the system is structured into LAYERS so that each layer provides a set of services to the layer above and uses the services of the layer below. Within each LAYER all constituent components work at the same level of abstraction and can interact through connectors. Between two adjacent layers a clearly defined interface is provided. In the pure or strict form of the pattern, layers should not be by-passed: higher-level layers access lower-level layers only through the layer beneath.

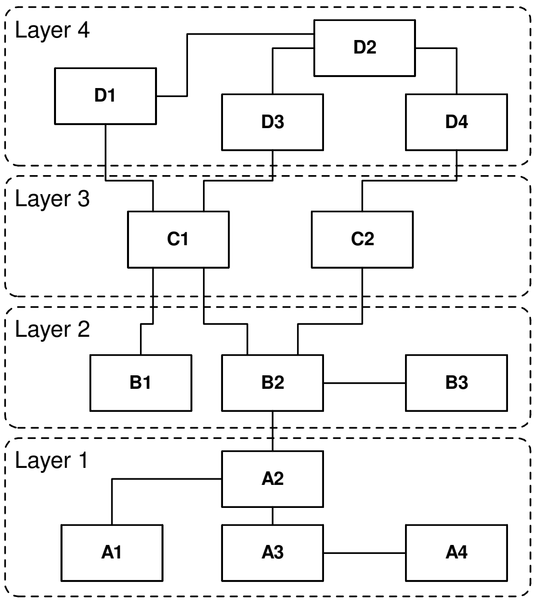

Figure 3: Layers example

Each layer offers a dedicated EXPLICIT INTERFACE [BH03] to the higher-level layers, which remains stable, whereas internal implementation details can change. This way the LAYERS pattern allows the work to be sub-divided along clear boundaries, which enables the division of labor. Two adjacent LAYERS can be considered as a CLIENT-SERVER pair, the higher layer being the client and the lower layer being the server. Also, the logic behind layers is especially obvious in the INDIRECTION LAYER where a special layer "hides" the details of a component or subsystem and provides access to its services. An example of LAYERS is shown in Figure 3.

LAYERS is useful for separating higher-level from lower-level responsibilities. By contrast, the patterns PIPES AND FILTERS and SHARED REPOSITORY place all components at the same level of abstraction. However, both of these patterns may use the LAYERS pattern for structuring the internal architecture of individual architecture elements.

A MICROKERNEL is a layered architecture with three LAYERS: external servers, the microkernel, and internal servers. Similarly the PRESENTATION-ABSTRACTION-CONTROL pattern also enforces LAYERS: a top layer with one agent, several intermediate layers with numerous agents, and one bottom layer which contains the "leaves" agents of the tree-like hierarchy.

A special layered architecture, which allows for implementing many of the following patterns (such as REFLECTION, VIRTUAL MACHINE, and INTERCEPTOR), is INDIRECTION LAYER:

Pattern: Indirection Layer

A sub-system should be accessed by one or more components, but direct access to the sub-system is problematic. For instance, the components should not get hard-wired into the sub-system, instead the accessors for the sub-system should be reused. Or the access should be defined in a way that it can be flexibly adapted to changes. The same problem appears at different levels of scale: it can happen between two ordinary components in one environment, components in two different languages, components in two different systems (e.g. if a legacy system is accessed).

An INDIRECTION LAYER is a LAYER between the accessing component and the “instructions” of the sub-system that needs to be accessed. The general term “instructions” can refer to a whole programming language, or an application programming interface (API), or the public interface(s) of a component or sub-system, or other conventions that accessing components must follow. The INDIRECTION LAYER wraps all accesses to the relevant sub-system and should not be bypassed. The INDIRECTION LAYER can perform additional tasks while deviating invocations to the sub-system, such as converting or tracing the invocations.

The INDIRECTION LAYER can either be integrated to the sub-system (as in “virtual machine” [CBB$^+$02]) or be an independent entity that forwards the invocations to the sub-system (as in ADAPTER or FACADE [GHJV94]). In both cases the accessing components are not aware of this, since the INDIRECTION LAYER aims at exactly that: hiding whatever it is that provides the services.

An example of an INDIRECTION LAYER architecture is shown in Figure 4. It shows a very simple form of INDIRECTION LAYER, consisting of wrappers that are independent of the sub-system providing the services. These wrappers forward the invocations to the hidden components of the subsystem, and perform some actions before and after the invocations. They can also be used to introduce add-on tasks such as logging. More complex INDIRECTION LAYERS, such as VIRTUAL MACHINES or INTERPRETERS, follow the same principle architecture, but perform more complex tasks and hide more components to which they forward requests.

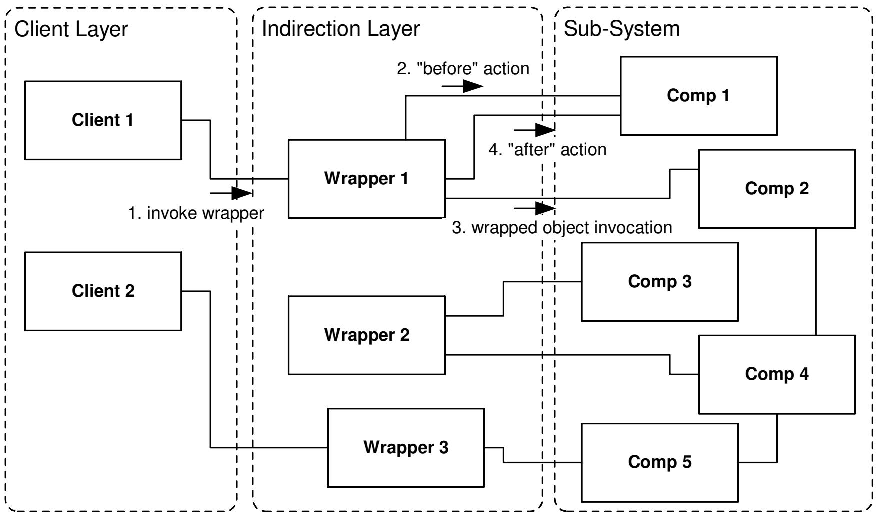

Figure 4: Indirection Layer example: simple wrapper layer

The INDIRECTION LAYER pattern can be thought of as a system of (at least) two or three LAY-

ERS: the INDIRECTION LAYER can be considered as Layer N which provides its services to Layer N+1, without disclosing the implementation details of the services to the latter. Sometimes the INDIRECTION LAYER provides all services itself, sometimes the services are provided by yet another Layer N-1 (e.g. a subsystem that is hidden by the INDIRECTION LAYER).

The INDIRECTION LAYER pattern is a foundation for the architectures of INTERPRETER, VIRTUAL MACHINE, and RULE-BASED SYSTEM. These patterns all provide an execution environment for a language defined on some platform. The INTERPRETER, VIRTUAL MACHINE, or RULE-BASED SYSTEM patterns interpose an INDIRECTION LAYER between the instructions of that language and the instructions of the platform.

The REFLECTION pattern might also be implemented using an INDIRECTION LAYER, so that the latter provides the reflective capabilities of the components defined on top of it. Specifically, the INDIRECTION LAYER can intercept all invocations of the components, and thus can use this information to record the current structure and behavior of the system. This information can then be provided using a reflection API.

4 Data Flow and Transformation View

The Data Flow and Transformation View shows how streams of data are successively processed or transformed by components. The major concerns addressed by this view are related to the design of the elements that perform the transformations, the elements that carry the streams of data, and the connections between those two kinds of elements. In addition, the view must describe the structure of the data and the rules that govern the transformations.

There are a number of quality attributes that are important for the Data Flow and Transformation View. Designing the system as a flow of data or a number of transformations, instead of e.g. using a single monolithic component, usually has the goal to enhance the Modifiability and Reusability of the two aforementioned types of elements. The elements should be flexibly combinable, each fulfilling a separate functionality, and thus they become reusable for different tasks. Flexible combination of the elements means that the architect has to deal with Integrability and Exchangeability concerns because it must be considered how the elements can be designed in such a way that they can easily be plugged together. Sending data through a number of elements, instead of only one, means that there are overheads regarding Efficiency, for instance because data needs to be transformed multiple times. However, there is also a potential for Efficiency gains, for instance because some elements can be executed in parallel.

The elements that perform the transformations are usually components that are independent of one another, and have input and output ports. The elements that carry the streams of data are connectors and similarly have data-in and data-out roles. The relationships between these elements are attachments that connect input ports of components to data-out roles of connectors, and output ports of components to data-in roles of connectors.

Pattern: Batch Sequential

Consider a complex task that can be sub-divided into a number of smaller tasks, which can be defined as a series of independent computations. This should not be realized by one monolithic component because this component would be overly complex, and it would hinder modifiability and reusability.

In a BATCH SEQUENTIAL architecture the whole task is sub-divided into small processing steps, which are realized as separate, independent components. Each step runs to completion and then calls the next sequential step until the whole task is fulfilled. During each step a batch of data is processed and sent as a whole to the next step.

BATCH SEQUENTIAL is a simple sequential data processing architectural pattern. It is useful for simple data flows, but entails severe overheads for starting the batch processes and transmitting data between them. PIPES AND FILTERS is more suitable for stream-oriented data-flow processing, where the filters incrementally transform their input streams into output streams.

Pattern: Pipes and Filters

Consider as in BATCH SEQUENTIAL the case where a complex task can be sub-divided into a number of smaller tasks, which can be defined as a series of independent computations. Additionally the application processes streams of data, i.e. it transforms input data streams into output data streams. This functionality should not be realized by one monolithic component because this component would be overly complex, and it would hinder modifiability and reusability. Furthermore, different clients require different variations of the computations, for instance, the results should be presented in different ways or different kinds of input data should be provided. To reach this goal, it must be possible to flexibly compose individual sub-tasks according to the client's demands.

In a PIPES AND FILTERS architecture a complex task is divided into several sequential sub-tasks. Each of these sub-tasks is implemented by a separate, independent component, a filter, which handles only this task. Filters have a number of inputs and a number of outputs, and they are connected flexibly using pipes but they are never aware of the identity of adjacent filters. Each pipe realizes a stream of data between two components. Each filter consumes and delivers data incrementally, which maximizes the throughput of each individual filter, since filters can potentially work in parallel. Pipes act as data buffers between adjacent filters.

The use of PIPES AND FILTERS is advisable when little contextual information needs to be maintained between the filter components and filters retain no state between invocations. PIPES AND FILTERS can be flexibly composed. However, sharing data between these components is expensive or inflexible. There are efficiency overheads for transferring data in pipes and data transformations, and error handling is rather difficult.

An example of PIPES AND FILTERS is shown in Figure 5. Forks/joins as well as feedback loops are allowed in this pattern, but there is also a variant referred to as a *pipeline*, that forbids both, i.e. it has a strict linear topology.

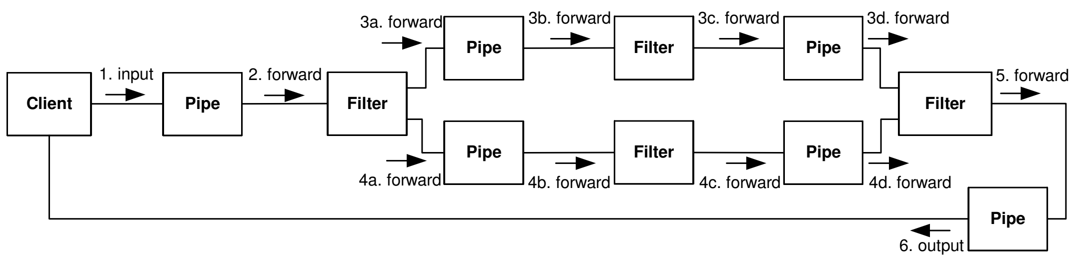

Figure 5: Pipes and filters example

In contrast to BATCH SEQUENTIAL, where there is no explicit abstraction for connectors, the PIPES AND FILTERS pattern considers the pipe connector to be of paramount importance for the transfer of data streams. The keyword in PIPES AND FILTERS is flexibility in connecting filters through pipes in order to assemble custom configurations that solve specific problems. Also in PIPES AND FILTERS there is a constant flow of data streams between the filters, while in BATCH SEQUENTIAL, the processing steps are discrete in the sense that each step finishes before the next step may commence.

The pure form of the PIPES AND FILTERS pattern entails that only two adjacent filters can share data through their pipe, but not non-adjacent filters. Therefore pure PIPES AND FILTERS is an alternative to LAYERS and SHARED REPOSITORIES, only if data sharing between non-adjacent processing tasks is not needed. On the other hand, more relaxed forms of the PIPES AND FILTERS pattern can be combined with data repository architectures like SHARED REPOSITORY, ACTIVE REPOSITORY, or BLACKBOARD to allow for data-sharing between filters. PIPES AND FILTERS can also be used for communication between LAYERS, if data flows through layers are needed.

5 Data Repository View

The Data Repository View shows how multiple components access a central repository of data. In other words, a persistent, shared data store is accessed and modified by a number of elements. The view shows how the shared data store is created, accessed, and updated, and how the data is distributed. An important concern is whether the data store is passive or active, i.e. does it notify its accessors or are the accessors responsible of finding data of interest to them? Also, the (distributed) communication between the data store and the elements, which access it, must be described. For instance, the accessor elements might either communicate indirectly through the shared data or directly with each other.

The Data Repository View is mainly related to the following quality attributes. Using a central data store architecture means that Reusability of the data store and its access mechanisms is supported for all concerns related to data storing. Separating the elements from each other, and connecting them only loosely through the central data store, means that Changeability and Maintainability of the architecture are supported. Since all data access depends on only one component that is usually accessed remotely, Scalability and Efficiency issues might become a problem. Special measures might thus be needed to eliminate Scalability and Efficiency problems. Communication through a central data store is a simple solution to solve Integrability problems of software systems.

The data store and the elements that access it are components. The data store is independent of the components, and the components are usually independent of one another. It is possible that there is more than one data store. The elements that transfer data written or read from the data stores are connectors that are attached to the data store(s) and the accessors.

Pattern: Shared Repository

Data needs to be shared between components. In sequential architectures like LAYERS or PIPES AND FILTERS the only way to share data between the components (layer elements or filters) is to pass the information along with the invocation, which might be inefficient for large data sets. Also it might be inefficient, if the shared information varies from invocation to invocation because the components' interfaces must be prepared to transmit various kinds of data. Finally the long-term persistence of the data requires a centralized data management.

In the SHARED REPOSITORY pattern one component of the system is used as a central data store, accessed by all other independent components. This SHARED REPOSITORY offers suitable means for accessing the data, for instance, a query API or language. The SHARED REPOSITORY must be scalable to meet the clients' requirements, and it must ensure data consistency. It must handle problems of resource contention, for example by locking accessed data. The SHARED REPOSITORY might also introduce transaction mechanisms.

An example of a SHARED REPOSITORY architecture is shown in Figure 6.

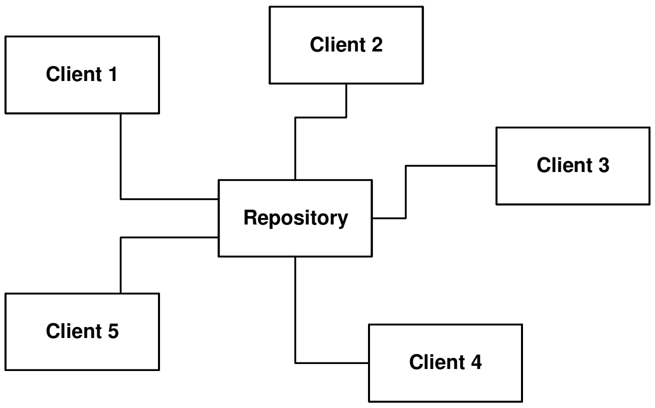

Figure 6: Shared repository example

A SHARED REPOSITORY also might offer additional services, such as security or logging. Some systems offer higher-level access mechanisms, such as query languages or tuple spaces [GCCC85].

A SHARED REPOSITORY offers an alternative to sequential architectures for structuring software components, such as LAYERS and PIPES AND FILTERS, that should be considered, when data sharing or other interaction between non-adjacent components is needed. SHARED REPOSITORIES can be used in a PIPES AND FILTERS architecture to allow for data sharing between filters.

A SHARED REPOSITORY, where all its clients are independent components, can be considered as CLIENT-SERVER, with the data store playing the server part. Similarly it can be considered as a system of two LAYERS where the higher level of clients uses the services of the lower level of the SHARED REPOSITORY.

A variant of the SHARED REPOSITORY pattern is the ACTIVE REPOSITORY pattern[2]:

Pattern: Active Repository

A system needs to have a SHARED REPOSITORY, but it should not just be *passively* accessed by accessor components. Clients need to be immediately informed of specific events in the SHARED REPOSITORY, such as changes of data or access of data. "Polling" (i.e. querying in frequent intervals) the SHARED REPOSITORY for such events does not work, for instance, because this does not deliver timely information or inflicts overhead on the system performance.

[2]Note that this pattern is introduced as a style named "Blackboard" in [SC97, SG96, BCK98]. We renamed it for this pattern language to ACTIVE REPOSITORY to avoid confusion with the BLACKBOARD pattern from [BMR$^+$96] which we discuss below.

An ACTIVE REPOSITORY is a SHARED REPOSITORY that is “active” in the sense that it informs a number of subscribers of specific events that happen in the shared repository. The ACTIVE REPOSITORY maintains a registry of clients and informs them through appropriate notification mechanisms.

The notification mechanism can be realized using ordinary EXPLICIT INVOCATIONS, but in most cases IMPLICIT INVOCATIONS, such as PUBLISH-SUBSCRIBE, are more appropriate.

Another variant of the SHARED REPOSITORY pattern is the BLACKBOARD pattern, which is appropriate when a SHARED REPOSITORY is used in an immature domain in which no deterministic approach to a solution is known or feasible.

Pattern: Blackboard

Consider the case where a SHARED REPOSITORY is needed for the shared data of a computation, but no deterministic solution strategies are known. Examples are image recognition or speech recognition applications. However, it should be possible to realize a solution for these types of applications.

In a BLACKBOARD architecture the complex task is divided into smaller sub-tasks for which deterministic solutions are known. The BLACKBOARD is a SHARED REPOSITORY that uses the results of its clients for heuristic computation and step-wise improvement of the solution. Each client can access the BLACKBOARD to see if new inputs are presented for further processing and to deliver results after processing. A control component monitors the blackboard and coordinates the clients according to the state of the blackboard.

An example of a BLACKBOARD architecture is shown in Figure 7. Even though the control component is designed as a separate component, it may as well be part of the clients, the blackboard itself, or a combination of the above.

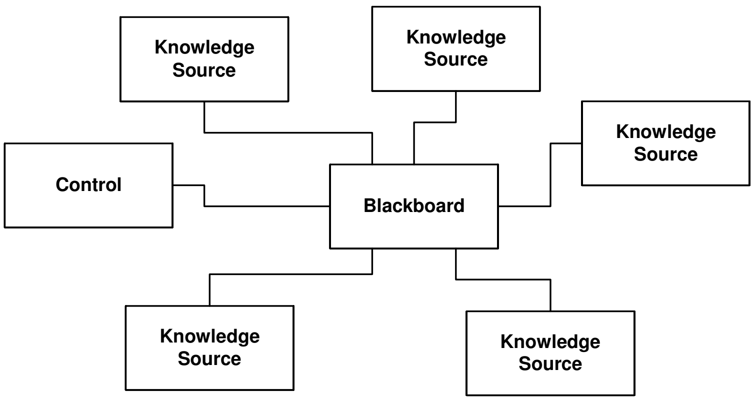

Figure 7: Blackboard example

6 Adaptation Infrastructure View

The Adaptation Infrastructure View shows the infrastructure that a system uses to adapt itself during evolution. The system is viewed as a core part that remains invariable and an adaptable part that either changes over time or in different versions of a system. Therefore, the major concern in this view is to differentiate the system functionality that is more likely to change and the functionality that will possibly remain invariable. The view also describes how the invariable parts communicate with the adaptable parts.

The most important quality attributes addressed in the Adaptation Infrastructure View are, of course, Modifiability, Extensibility, Exchangeability and Evolvability. But Reusability is also of central importance because the invariable elements can be reused, whereas the adaptable elements change. Thus Reusability concerns often help us to find a proper differentiation between variable and invariable elements. Adaptable architectures are also often used to support Integrability because integration often requires unforeseen changes. Often adaptable architectures require runtime indirections, thus Efficiency might be influenced negatively by more adaptable architectures.

The two basic types of elements in this view are the invariable components and the adaptable components (these are often called variation points). These two kinds of components communicate with each other through connectors that have clearly specified interfaces. Note that some kinds of connectors are adaptable as well, i.e. they are also used as variation points.

Figure 8 summarizes the patterns and their relationships from the Adaptation Infrastructure View.

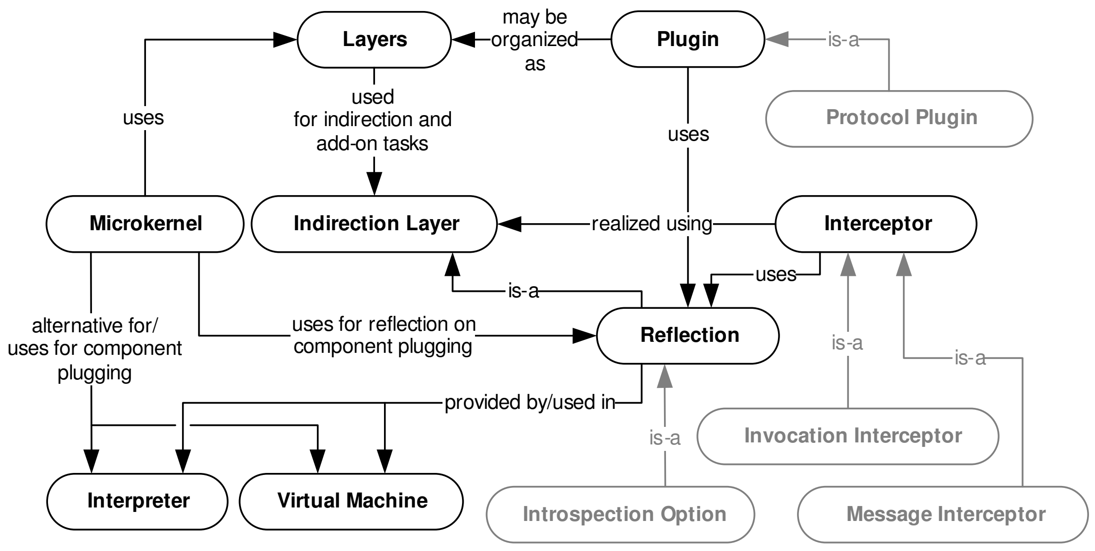

Figure 8: Overview: patterns of the Adaptation Infrastructure View

Pattern: Microkernel

Consider a system family where different versions of a system need to be supported. In each version, components can be composed in different ways and other details, such as the offered services, public APIs, or user interfaces, might be different. Nonetheless, the system family should be realized using a common architecture to ease software maintenance and foster reuse.

A MICROKERNEL realizes services that all systems, derived from the system family, need and a plug-and-play infrastructure for the system-specific services. Internal servers (not visible to clients) are used to realize version-specific services and they are only accessed through the MICROKERNEL. On the other hand, external servers offer APIs and user interfaces to clients by using the MICROKERNEL. External servers are the only way for clients to access the MICROKERNEL architecture.

The MICROKERNEL pattern promotes flexible architectures which allow systems to adapt successfully to changing system requirements and interfaces. An example of a MICROKERNEL architecture is shown in Figure 9.

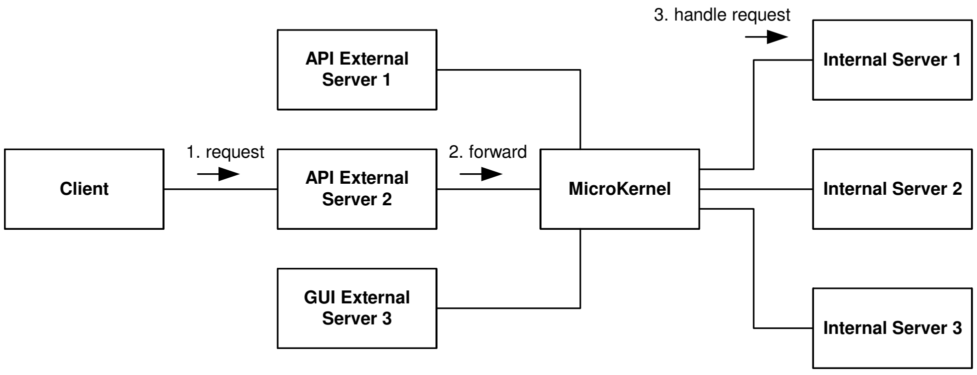

Figure 9: Microkernel example

MICROKERNELS are usually structured in LAYERS: the lowest layer implements an abstraction of the system platform, the next layer implements the services (of the internal servers) used by the MICROKERNEL, the next layer implements the functionality shared by all application versions, and the highest layer glues the external and internal servers together. Apparently, the lowest layer is in fact an INDIRECTION LAYER hiding the low-level system details from the application logic.

In some areas, the patterns INTERPRETER and VIRTUAL MACHINE are alternatives to MICROKERNEL since they all offer a way to integrate or glue components. The MICROKERNEL can integrate version-specific components, while most INTERPRETERS and VIRTUAL MACHINES also offer some way to be extended with components. Thus all three patterns can be used to develop a plug-and-play environment for components. For instance, most scripting languages use their INTERPRETER to offer a gluing framework for components written in the language in which the scripting language itself is implemented (e.g. C, C++, or Java). It is even possible to combine an INTERPRETER or VIRTUAL MACHINE with a MICROKERNEL by implementing the plug-and-play environment of the former as a MICROKERNEL.

A MICROKERNEL introduces an indirection that can be useful in certain CLIENT-SERVER configurations: a client that needs a specific service can request it indirectly through the MICROKERNEL, which establishes the communication to the server that offers this service. In this sense all communication between clients and servers is mediated through the MICROKERNEL, for reasons of e.g. security or modifiability. To develop distributed MICROKERNEL architectures, the MICROKERNEL can be combined with the BROKER pattern to hide the communication details between clients that request services and servers that implement them.

For all environments that support plug-and-play of components, REFLECTION is useful because it allows to find out which components are currently composed in which way.

Pattern: Reflection

Software systems constantly evolve and change over the time, and unanticipated changes are often required. It is hard to automatically cope with changes that are not foreseen.

In a REFLECTION architecture all structural and behavioral aspects of a system are stored into meta-objects and separated from the application logic components. The latter can query the former in order to execute their functionality. The meta-objects can of course be modified through a special protocol. Thus REFLECTION allows a system to be defined in a way that allows for coping with unforeseen situations automatically.

The REFLECTION pattern is organized into LAYERS: the meta-level contains the meta-objects which encapsulate the varying structure and behavior; and the base level contains the application logic components that depend on the meta-objects. However this is not the pure LAYERS pattern since not only the base layer uses the services of the meta layer but also the opposite may happen. This is useful for building a reflective system from scratch. The REFLECTION can also be realized with other architectures. For instance, many existing INTERPRETERS and VIRTUAL MACHINES are reflective in the sense that the information in the implementations of these patterns can be used to provide REFLECTION.

An example of a REFLECTION architecture is shown in Figure 10.

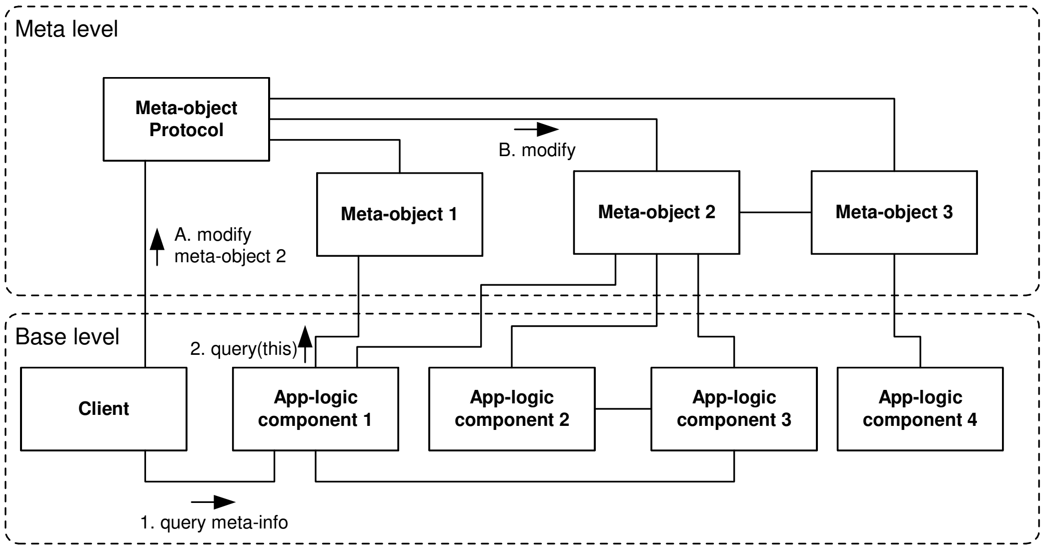

Figure 10: Reflection example

In [Zdu04, Zdu03] more detailed patterns about the realization of REFLECTION in the context of aspect-oriented composition frameworks are presented. The respective variant of REFLECTION is the pattern INTROSPECTION OPTION.

INDIRECTION LAYER is a more generic pattern than REFLECTION. It can be used to build a REFLECTION-infrastructure, but also other "meta-level" patterns, such as INTERPRETER or VIRTUAL MACHINE.

In cases where components are dependent on multiple runtime environments, PLUGINS are useful:

Pattern: Plugin

Consider multiple runtime environments (or protocols) need to be supported by a component. A simple solution is to introduce an interface between the component and each runtime environment. But this means that changes of the runtime environment cannot happen at runtime, but instead the editing, rebuilding, and redeploying of the component is necessary. If such configurations are needed for multiple runtime environments and components, the changes might be scattered throughout the system.

PLUGINS resolve both problems by providing a centralized configuration that can be performed at runtime. PLUGIN components also introduce an interface, but at a single, central point so that configuration can be easily managed. PLUGIN components can be connected and disconnected at runtime rather than during compilation. That is, reconfigurations do not require rebuilding and redeploying.

Configuration of PLUGINS is often done in text files or XML files, or using a user interface. PLUGINS often use REFLECTION because REFLECTION enables connecting components without compile-time dependencies on them. The REFLECTION facility simply looks up the PLUGIN class at runtime and connects it. In environments that do not support REFLECTION, PLUGINS can be introduced by compiling all viable configuration options into the system. This however means that the PLUGIN architecture is less flexible regarding unforeseen changes than the REFLECTION variant, because only foreseen configuration options are covered.

If an architecture provides multiple PLUGINS for one task that are exchangeable at runtime, it is often advisable to organize the PLUGINS in a plugin LAYER. For instance, consider multiple protocols are supported by a middleware to be used by clients and servers. Usually the BROKER of the middleware hides the details of the protocols in its lowest layer, a layer consisting of PROTOCOL PLUGINS. This variant of the PLUGIN pattern is discussed in [VKZ04].

In cases where we need an adaptable *framework* to accommodate future services, the INTERCEPTOR pattern is appropriate:

Pattern: Interceptor

A framework offers a number of reusable services to the applications that extend it. These services need to be updated in the future as the application domain matures, and they should still be offered by the framework, so that the application developers do not need to re-implement them. Furthermore, the framework developers cannot predict all such future services at the point of time where the framework is created, while application developers may not be able to add unanticipated extensions to the framework, in case e.g. that the framework is a black-box.

An INTERCEPTOR is a mechanism for transparently updating the services offered by the framework in response to incoming events. An application can register with the framework any number of INTERCEPTORS that implement new services. The framework facilitates this registration through dispatchers that assign events to INTERCEPTORS. The framework also provides the applications with the means to introspect on the framework's behavior in order to properly handle the events.

The INTERCEPTOR pattern can be realized using an INDIRECTION LAYER or one of its variants, such as INTERPRETER or VIRTUAL MACHINE. The incoming events are therefore re-routed through the INDIRECTION LAYER that consists of several INTERCEPTORS, before they are dispatched to the intended receiver.

INTERCEPTOR can use a REFLECTION mechanism in order to query the framework and retrieve the necessary information to process incoming events.

INTERCEPTOR is defined in special variants: for aspect-oriented composition as MESSAGE INTERCEPTOR in [Zdu04, Zdu03]; for middleware architectures as INVOCATION INTERCEPTOR in [VKZ04]. An example of an INTERCEPTOR architecture is shown in Figure 11. In the figure an invocation is intercepted automatically, and an interceptor manager is invoked, instead of the original target component. Two interceptors are configured for the target component, which are invoked before the invocation. After the interceptors, the target component itself is invoked. When the invocation returns, the interceptors are invoked again, this time in reverse order.

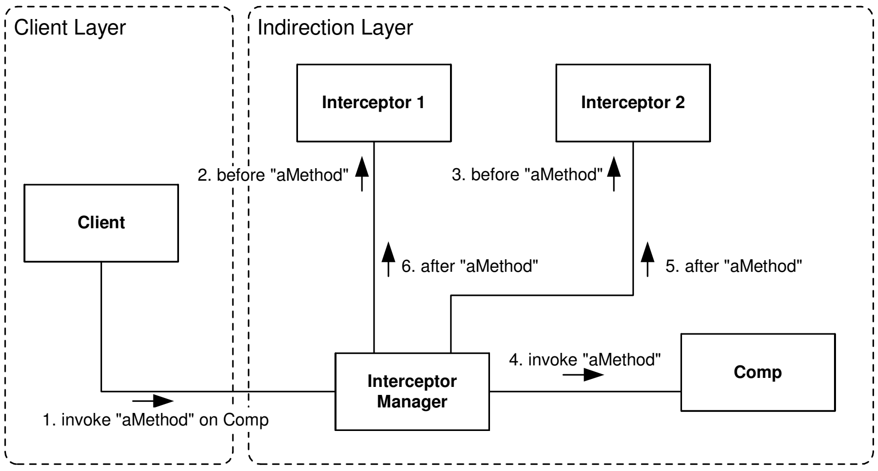

Figure 11: Interceptor example

A framework offers a number of reusable services to the applications that extend it. These services need to be updated in the future as the application domain matures and they should still be offered by the framework, so that the application developers do not need to re-implement them. Furthermore, the framework developer cannot predict all such future services at the point of time where the framework is created, while application developers may not be able to add unanticipated extensions to the framework, in case e.g. that the framework is a black-box. An INTERCEPTOR is a mechanism for transparently updating the services offered by the framework in response to incoming events. An application can register with the framework any number of INTERCEPTORS that implement new services. The framework facilitates this registration through dispatchers that assign events to INTERCEPTORS. The framework also provides the applications with the means to introspect on the framework's behavior in order to properly handle the events.

7 Language Infrastructure View

The Language Infrastructure View shows infrastructures for extending a system with languages, such as domain-specific languages. The system is viewed as a part that is native to the software/hardware environment and another part that is not. The view deals with how a part of the system that is written in a non-native language can be integrated with the system. The program instructions in the non-native part need to be translated into the native environment.

The goal of a language infrastructure is often to provide language abstractions at a higher abstraction level than the native language. For instance, Python introduces a higher-level language on top of the native language C in which it is implemented. A domain-specific language written in Java provides domain abstractions on top of Java.

A number of quality attributes is important for the Language Infrastructure View. Modifiability is often the reason for using a higher-level language. For instance, Python is often used because its scripts can be changed and evaluated at runtime, whereas its native language, C, requires re-compilation for changes. Portability is also often a concern when deciding for a language infrastructure. For instance, one benefit of Java and Python is that both languages are highly portable between platforms.

The native part of the application and the non-native part are components. These communicate indirectly through another type of component, an interpreter component that “translates” the latter into the former. The connector between these components is data that contains the program instructions in the non-native language, as well as the internal state of the non-native part.

Figure 12 gives an overview of the patterns of the Language Infrastructure View.

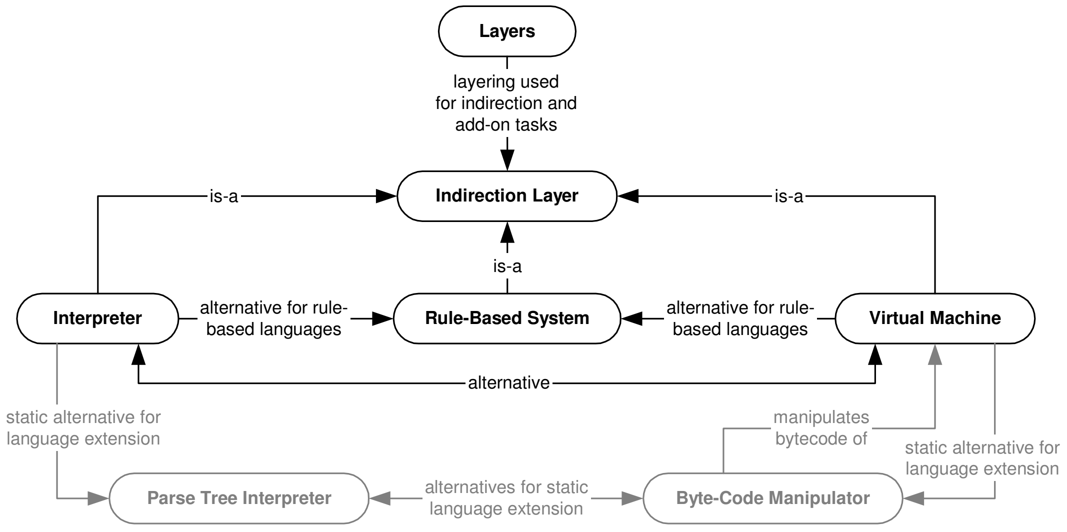
Layers
layering used for indirection and add-on tasks
Indirection Layer
is-a
is-a
is-a
Interpreter
alternative for rule-based languages
Rule-Based System
alternative for rule-based languages
Virtual Machine
alternative
static alternative for language extension
manipulates bytecode of
static alternative for language extension
Parse Tree Interpreter
alternatives for static language extension
Byte-Code Manipulator

Figure 12: Overview: patterns of the Language Infrastructure View

Pattern: Interpreter

A language syntax and grammar needs to be parsed and interpreted within an application. The language needs to be interpreted at runtime (i.e. using a compiler is not feasible).

An INTERPRETER for the language is provided, which provides both parsing facilities and an execution environment. The program that needs to be interpreted is provided in form of scripts which are interpreted at runtime. These scripts are portable to each platform realization of the INTERPRETER. For instance, the INTERPRETER can define a class per grammar rule of the language. The parser of the interpreter parses language instructions according to these rules and invokes the interpretation classes. Many more complex INTERPRETER architectures exist.

Some INTERPRETERS use optimizations like on-the-fly byte-code compilers. They thus realize internally elements of a VIRTUAL MACHINE. Note that an INTERPRETER is different to a VIRTUAL MACHINE because it allows for runtime interpretation of scripts, whereas the VIRTUAL MACHINE architecture depends on compilation before runtime:

Pattern: Virtual Machine

An efficient execution environment for a programming language is needed. The architecture should facilitate portability, code optimizations, and native machine code generation. Runtime interpretation of the language is not necessarily required.

A VIRTUAL MACHINE defines a simple machine architecture on which not machine code but an intermediate form called the byte-code can be executed. The language is compiled into that byte-code. The VIRTUAL MACHINE can be realized on different platforms, so that the byte-code can be portable between these platforms. The VIRTUAL MACHINE redirects invocations from a byte-code layer into an implementation layer for the commands of the byte-code.

An alternative to INTERPRETERS and VIRTUAL MACHINES, when rule-based or logical languages are needed, is a RULE-BASED SYSTEM:

Pattern: Rule-Based System

Logical problems are hard to express elegantly in imperative languages that are typically used in INTERPRETERS and VIRTUAL MACHINES. Consider for instance an expert system that provides the knowledge of an expert or a set of constraints. In imperative languages these are expressed by nested if-statements or similar constructs which are rather hard to understand.

A RULE-BASED SYSTEM offers an alternative for expressing such problems in a system. It consists mainly of three things: facts, rules, and an engine that acts on them. Rules represent knowledge in form of a condition and associated actions. Facts represent data. A RULE-BASED SYSTEM applies its rules to the known facts. The actions of a rule might assert new facts, which, in turn, trigger other rules.

As mentioned before, INDIRECTION LAYER is the architectural foundation for INTERPRETER, VIRTUAL MACHINE, and RULE-BASED SYSTEM, since either the instructions of the language or the byte-code are re-directed dynamically (at runtime). We do not show example diagrams for these three patterns, since their structure is similar to that of an INDIRECTION LAYER.

In [Zdu04, Zdu03] a number of static alternatives for building language extensions are presented in the context of aspect-oriented composition frameworks. These, for instance, manipulate the parse tree (PARSE TREE INTERPRETER) or byte-code (BYTE-CODE MANIPULATOR) of a language. The relationships to these patterns for aspect-oriented composition are also shown in Figure 12.

8 Interaction Decoupling View

The Interaction Decoupling View shows how the interactions in a system can be decoupled, for instance, how to decouple user interface logic from application logic and data. Thus this view shows what is the data and the application logic that is associated to the user interface, and how these elements are decoupled from each other.

If a user interface is involved in the Interaction Decoupling View, one major concern is the Usability of the system. Decoupling interactions usually has the goal to increase the Modifiability of the system as a whole. Also, Maintainability and Reusability of the system elements might benefit from decoupling them. Decoupling however might increase Complexity and decrease Efficiency because of additional communication overhead.

The elements that present data to the user, accept the user input, and contain the application logic and data are implemented as components. The components interact with each other through connectors that pass data from one to another. This interaction is usually a message-based change notification mechanism.

The relationships between the patterns of this view are shown in Figure 13.

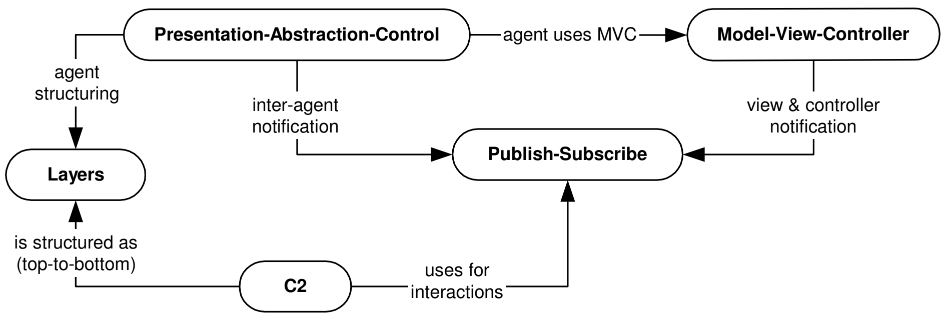

Figure 13: Overview: patterns of the Interaction Decoupling View

Pattern: Model-View-Controller

A system may offer multiple user interfaces. Each user interface depicts all or part of some application data. Changes to the data should be automatically and flexibly reflected to all the different user interfaces. It should be also possible to easily modify any one of the user interfaces, without affecting the application logic associated with the data.

The system is divided into three different parts: a *Model* that encapsulates some application data and the logic that manipulates that data, independently of the user interfaces; one or multiple *Views* that display a specific portion of the data to the user; a *Controller* associated with each View that receives user input and translates it into a request to the Model. Views and Controllers constitute the user interface. The users interact strictly through the Views and their Controllers, independently of the Model, which in turn notifies all different user interfaces about updates.

The notification mechanism that updates all Views and Controllers according to the Model can be based on PUBLISH-SUBSCRIBE. All Controllers and Views subscribe to the Model, which in turn publishes the notifications. An example of a MODEL-VIEW-CONTROLLER architecture is shown in Figure 14.

Pattern: Presentation-Abstraction-Control

An interactive system may offer multiple diverse functionalities that need to be presented to the user through a coherent user interface. The various functionalities may require their own custom user interface, and they need to communicate with other functionalities in order to achieve a greater goal. The users need not perceive this diversity but should interact with a simple and consistent interface.

The system is decomposed into a tree-like hierarchy of agents: the leaves of the tree are agents that are responsible for specific functionalities, usually assigned to a specific user interface; at the middle layers there are agents that combine the functionalities of related lower-level agents to offer greater services; at the top of the tree, there is only one agent that orchestrates the middle-layer agents to offer the collective functionality. Each agent is comprised of three parts: a *Presentation* takes care of the user interface; an *Abstraction* maintains application data and the

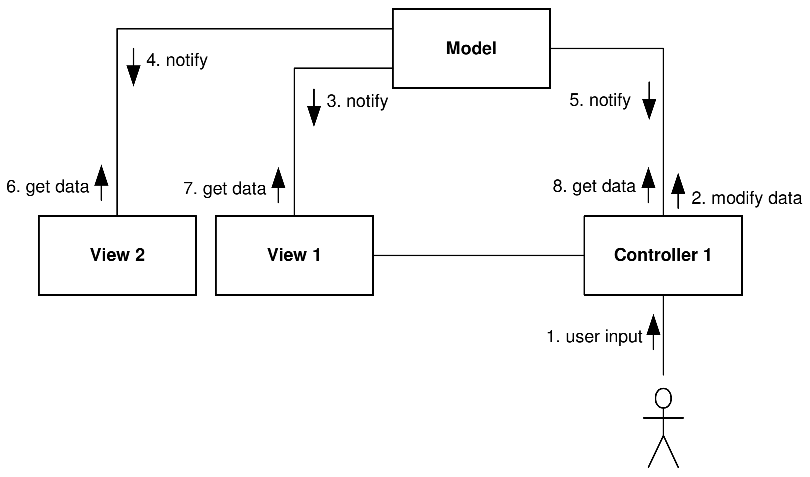

Figure 14: Model-View-Controller example

logic that modifies it; a *Control* intermediates between the Presentation and the Abstraction and handles all communication with the Controls of other Agents.

The PRESENTATION-ABSTRACTION-CONTROL pattern is in essence based on MODEL-VIEW-CONTROLLER, in the sense that every agent is designed according to MVC: the Abstraction matches the MVC Model, while the presentation matches the MVC View and Controller.

On a more macroscopic level, the PRESENTATION-ABSTRACTION-CONTROL pattern is structured according to LAYERS: the top layer contains the chief agent that controls the entire application; the middle layer contains agents with coarse-grained functionality; the lower layer is comprised of fine-grained agents that handle specific services, which users interact with. An example of a PRESENTATION-ABSTRACTION-CONTROL architecture is shown in Figure 15.

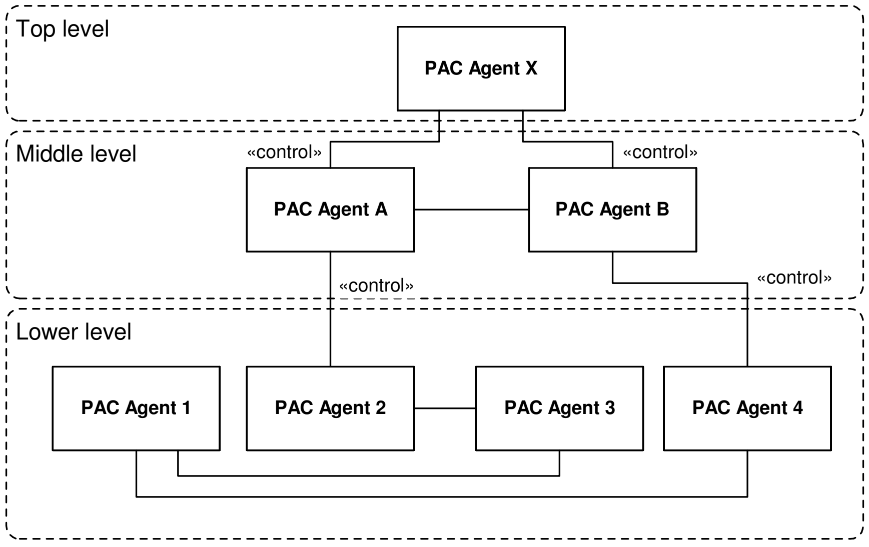

Figure 15: Presentation-Abstraction-Control example

The various agents usually need to propagate changes to the rest of the agent hierarchy, and this can be achieved again through the PUBLISH-SUBSCRIBE pattern. Usually higher-level agents subscribe to the notifications of lower-level agents.

An alternative to MVC and PAC for applications with extensive user interface requirements and other particular requirements is the C2 architectural pattern.

Pattern: C2

An interactive system is comprised of multiple components such as GUI widgets, conceptual models of those widgets at various levels, data structures, renderers, and of course application logic. The system may need to support several requirements such as: different implementation language of components, different GUI frameworks, distribution in a heterogeneous network, concurrent interaction of components without shared address spaces, run-time reconfiguration, multi-user interaction. Yet the system needs to be designed to achieve separation of concerns and satisfy its performance constraints.

The system is decomposed into a top-to-bottom hierarchy of concurrent components that interact asynchronously by sending messages through explicit connectors. Components submit request messages upwards in the hierarchy, knowing the components above, but they send notification messages downwards in the hierarchy, without knowing the components lying beneath. Components are only connected with connectors, but connectors may be connected to both components and other connectors. The purposes of connectors is to broadcast, route, and filter messages.

An example of a C2 architecture with four components in three layers, and two connectors that delimit the layers, is shown in Figure 16.

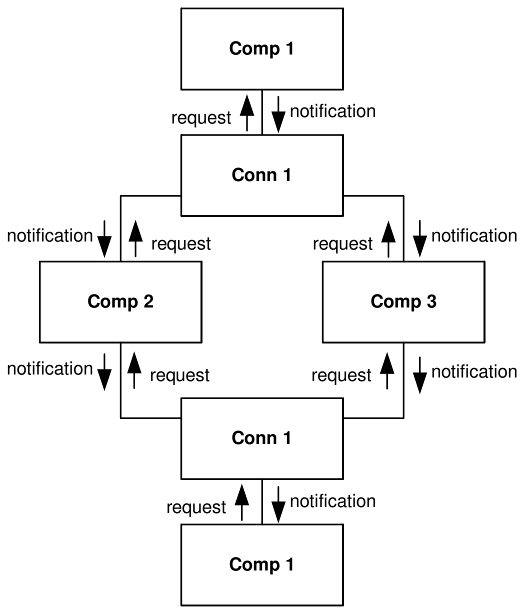

Figure 16: C2 example

The C2 top-to-bottom hierarchy resembles the form of a LAYERS architecture in an upside-down order. A C2 component that belongs to a given layer uses the services of the layers above it by invoking services on them and provides services to the layers below it by sending notifications to them.

Since the C2 pattern provides *substrate independence* [TMA+96], isolating a component from the components underneath it, the layer where a component is placed is in essence an INDIRECTION LAYER.

The interaction between the C2 components takes place through asynchronous message exchange, thus utilizing an IMPLICIT INVOCATION mechanism, and specifically callbacks, e.g. PUBLISH-SUBSCRIBE.

9 Component Interaction View

The Component Interaction View shows how individual components interact with each other but retain their autonomy. In the Component Interaction View the system is viewed as a number of independent components that interact with each other in the context of a system. The view shows how the independent components interact, e.g. by exchanging messages and how they are decoupled from each other.

A number of quality attributes play a role in a decision for a component interaction mechanism. Emphasis is on Modifiability because each component interaction mechanism should support design for change so that component interactions can be adapted to new situations. Also, Integrability is central because components of different kinds should be able to interact via a component interaction mechanism.

The components retain their independence, since they merely exchange data but do not directly control each other. The components interact with each other through connectors that pass data from one to another. This interaction can be performed synchronously or asynchronously and can be message-based or through direct calls. This view is closely connected to the distributed view, since the independent components might be distributed in various network nodes or processes.

Figure 17 shows the relations of the two basic patterns in the Component Interaction View, IMPLICIT INVOCATION and EXPLICIT INVOCATION.

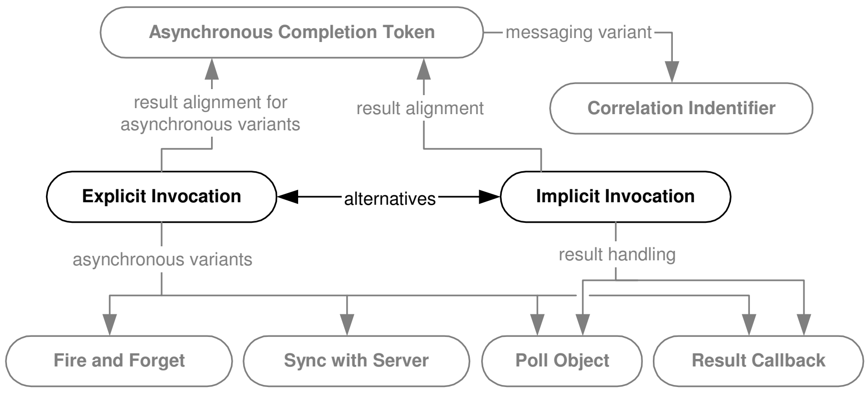

Figure 17: Overview: patterns for basic component interaction

The two major patterns in this view differentiate whether the components interact through explicit or implicit invocations[3]:

[3]Note that in [BCK98] an EXPLICIT INVOCATION is given the opposite meaning as it is considered a sub-pattern of the EVENT SYSTEMS pattern. However we have chosen this name to show the contrast between components explicitly (e.g. direct method call) and implicitly (e.g. events) invoking each other.

Pattern: Explicit Invocation

Consider a component, the client, which needs to invoke a service defined in another component, the supplier. Coupling the client with the supplier in various ways is not only harmless but often desirable. For example the client must know the exact network location of the component which offers the service in order to improve performance; or the client must always initiate the invocation itself; or the client must block, waiting for the result of the invocation, before proceeding with its business; or the topology of the interacting clients and suppliers is known beforehand and must remain fixed. How can these two components interact?

An EXPLICIT INVOCATION allows a client to invoke services on a supplier, by coupling them in various respects. The decisions that concern the coupling (e.g. network location of the supplier) are known at design-time. The client provides these design decisions together with the service name and parameters to the EXPLICIT INVOCATION mechanism, when initiating the invocation. The EXPLICIT INVOCATION mechanism performs the invocations and delivers the result to the client as soon as it is computed. The EXPLICIT INVOCATION mechanism may be part of the client and the server or may exist as an independent component.

An example of an EXPLICIT INVOCATION architecture is shown in Figure 18, where the EXPLICIT INVOCATION mechanism is implemented with the help of a BROKER and a PROXY, as part of both the client and the supplier.

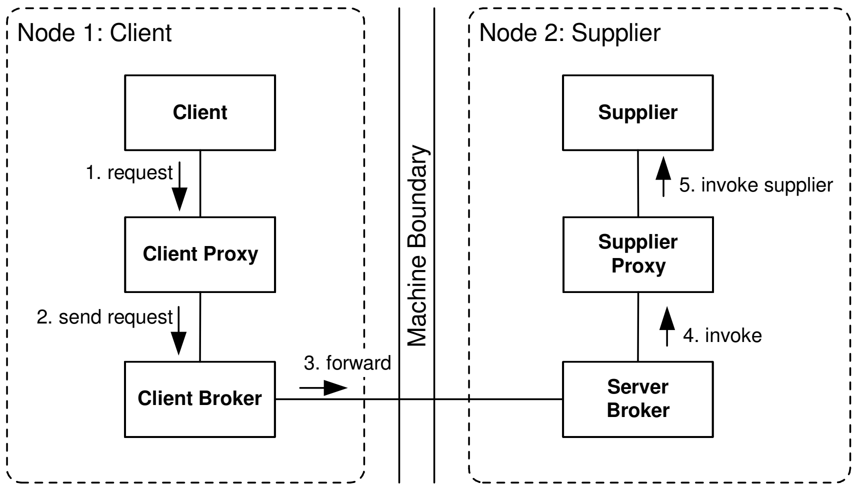

Figure 18: Explicit Invocation example

During the EXPLICIT INVOCATION the identification of the service supplier, can be realized, for instance by using the pattern OBJECT ID [VKZ04]. The client also knows the location of the service supplier, and furthermore, in some systems, the service supplier needs to know about the location of the client, so that the result can be sent back. This can be achieved by OBJECT IDS enriched with location information, as mandated by the pattern ABSOLUTE OBJECT REFERENCE [VKZ04].

There are two main variants of EXPLICIT INVOCATIONS: synchronous, explicit invocations and asynchronous, explicit invocations. In a synchronous invocation, the client blocks until the result is available. In an asynchronous invocation, the client continues with its work immediately, and the result is delivered at a later point, after it is computed. There are four patterns that describe different variants of asynchronous invocations for distributed systems [VKZ04]:

- The FIRE AND FORGET pattern describes best effort delivery semantics for asynchronous operations but does not convey results or acknowledgments.
- The SYNC WITH SERVER pattern describes invocation semantics for sending an acknowledgment back to the client once the operation arrives on the server side, but the pattern does not convey results.
- The POLL OBJECT pattern describes invocation semantics that allow clients to poll (query) for the results of asynchronous invocations, for instance, in certain intervals.
- The RESULT CALLBACK pattern also describes invocation semantics that allow the client to receive results; in contrast to POLL OBJECT, however, it actively notifies the requesting client of asynchronously arriving results rather than waiting for the client to poll for them.

These four patterns for realizing EXPLICIT INVOCATIONS in distributed systems, can be hard-coded, but this only makes sense, if specific aspects need to be optimized. Otherwise the BROKER pattern can be applied, which provides a reusable implementation of the individual Remoting Patterns [VKZ04] (those named above and others).

A general alternative to EXPLICIT INVOCATIONS are IMPLICIT INVOCATIONS, even though they can be met together in a single system:

Pattern: Implicit Invocation

Consider the case where an invocation is needed, such as in EXPLICIT INVOCATION. Furthermore, the client must be decoupled in various ways from the supplier, during the delivery of the invocation and of the result: the client might not know which supplier serves the invocation; or the client may not initiate the invocation itself but is merely interested in the invocation result; or the client does not need the result right away so it can be occupied with another task in the meantime; or the supplier might not be ready to reply to the client until some condition has been met; or clients may be added or removed dynamically during the system runtime; or the client does not know that the supplier is up and running and, if the supplier is down, the system should suspend the invocation until the supplier is up again; or the client and the supplier are part of dissimilar systems and thus the invocation must be transformed, queued, or otherwise manipulated during delivery. How can such additional requirements during delivery be met?

In the IMPLICIT INVOCATION pattern the invocation is not performed explicitly from client to supplier, but indirectly through a special mechanism such as PUBLISH-SUBSCRIBE, MESSAGE QUEUING, or broadcast, that decouples clients from suppliers. All additional requirements for invocation delivery are handled by the IMPLICIT INVOCATION mechanism during the delivery of the invocation.

An example of implicit invocation is the synchronization between Model, View, and Controller in the MODEL-VIEW-CONTROLLER pattern, as depicted in Figure 15. The Model notifies its Views and Controllers whenever its data have been changed, and so Views and Controllers implicitly invoke the Model to get the updated data. The Views and Controllers are decoupled from the Model, since they do not initiate the invocation, but the Model does it when it accepts an certain event. Models and Controllers may also be added and removed dynamically.

IMPLICIT INVOCATION can be both synchronous and asynchronous as can EXPLICIT INVOCATION, meaning that the client can either block or not, waiting for the invocation result. However IMPLICIT INVOCATIONS are most often asynchronous, in contrast to EXPLICIT INVOCATIONS, which are usually synchronous. Thus the aforementioned patterns for asynchronous result handling, POLL OBJECT and RESULT CALLBACK, should be used for IMPLICIT INVOCATION as well. An even more prominent contrast between them is that in EXPLICIT INVOCATION the invocation is always deterministic from client to supplier, while in IMPLICIT INVOCATION the trigger happens through an event and is not necessarily initiated by a client (e.g. by the producer in PUBLISH-SUBSCRIBE).

Same as in EXPLICIT INVOCATION, distributed IMPLICIT INVOCATIONS usually use a BROKER to hide the details of network communication and allow the components to contain only their application logic. Note that an IMPLICIT INVOCATION mechanism decouples clients from suppliers, while the BROKER pattern decouples both from the communication infrastructure.

There are different IMPLICIT INVOCATION variants, with respect to the tasks performed during the delivery of the invocation. For instance, in a broadcast mechanism the location of the invocation receiver is unknown to the client, since the invocation is broadcast through the network. This variant is used, e.g. for looking up the initial reference in a PEER-TO-PEER system. An event system realizes PUBLISH-SUBSCRIBE, in order to decouple producers and consumers of data. The MESSAGE QUEUING pattern queues invocations and results to increase delivery reliability, handle temporal outages of the supplier, and perform other tasks.

Among the implicit and explicit invocation patterns and their variants, only the synchronous variant of EXPLICIT INVOCATION can align a result unambiguously to an invocation, because the client blocks on the result. For all other cases – when invocations are performed asynchronously, it is possible that one client sends multiple invocations after another, and results for these invocations arrive in a different order than the invocations. Because the same client performs the invocations, the OBJECT ID of the client cannot be used for aligning a result to an invocation. An ASYNCHRONOUS COMPLETION TOKEN [SSRB00] contains information that identifies the individual invocation and perhaps also other information such as a behavior to be executed in the client when the result is processed. The ASYNCHRONOUS COMPLETION TOKEN is sent along with each asynchronous invocation of a client, and the service supplier sends it back with the result. Thus the client can use this information to align the result to the invocation. The ASYNCHRONOUS COMPLETION TOKEN is used in IMPLICIT INVOCATIONS and asynchronous EXPLICIT INVOCATION to align invocations to incoming results.

Next, we show variants of the two basic patterns, IMPLICIT INVOCATION and EXPLICIT INVOCATION. These patterns are also refined in the Distributed Communication View. Hence, Figure 19 gives an overview of all the patterns for Component Interaction as well as Distributed Communication.

There are two variants of the EXPLICIT INVOCATION pattern: CLIENT-SERVER and PEER-TO-PEER.

Pattern: Client-Server

Two components need to communicate, and they are independent of each other, even running in different processes or being distributed in different machines. The two components are not equal peers communicating with each other, but one of them is initiating the communication, asking for a service that the other provides. Furthermore, multiple components might request the same service provided by a single component. Thus, the component providing a service must be able to cope with numerous requests at any time (i.e. the component must scale well). On the other hand, the requesting components using one and the same service might deal differently with the results. This asymmetry between the components should be reflected in the architecture for the

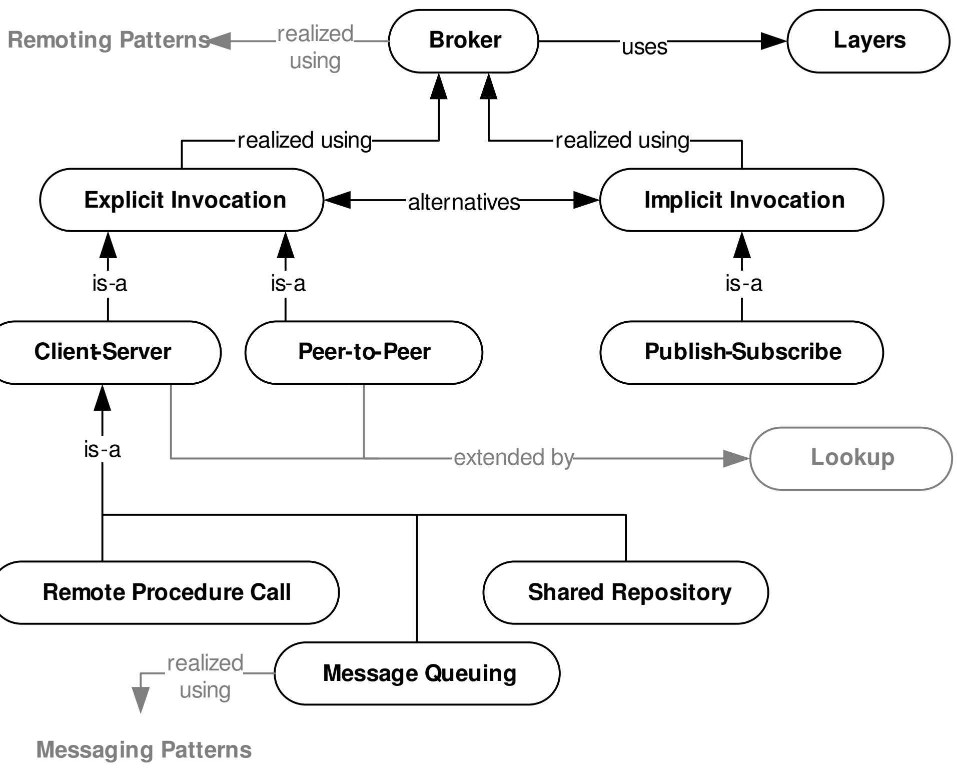

Figure 19: Overview: patterns for component interaction and distribution

optimization of quality attributes such as performance, shared use of resources, and memory consumption.

The CLIENT-SERVER pattern distinguishes two kinds of components: clients and servers. The client requests information or services from a server. To do so it needs to know how to access the server, that is, it requires an ID or an address of the server and of course the server's interface. The server responds to the requests of the client, and processes each client request on its own. It does not know about the ID or address of the client before the interaction takes place. Clients are optimized for their application task, whereas servers are optimized for serving multiple clients.

Both client and server must implement collective tasks, such as security, transaction, and systems management – something that is more complex in a CLIENT-SERVER architecture than in simple EXPLICIT INVOCATIONS.

Sophisticated, distributed CLIENT-SERVER architectures usually rely on the BROKER pattern to make the complexity of the distributed communication manageable. The same is true for the PEER-TO-PEER pattern.

Using the CLIENT-SERVER pattern we can build arbitrarily complex architectures by introducing multiple client-server relationships: a server can act itself as a client to other servers. The result is a so-called *n-Tier-architecture*. A prominent example of such architectures is the 3-tier-architecture (see Figure 20), which consists of:

- a client tier, responsible for the presentation of data, receiving user events, and controlling the user interface
- an application logic tier, responsible for implementing the application logic (also known

as business logic)

- a backend tier, responsible for providing backend services, such as data storage in a data base or access to a legacy system

It is also possible to combine the CLIENT-SERVER pattern with LAYERS in order to design a system where the client and the server components individually are layered. For example the ISO/OSI standard defines such an architecture, where there are seven layers in both the client and the server side, and each client communicates with the server at the same layer addressing a certain scope and responsibilities.

Furthermore SHARED REPOSITORIES or BLACKBOARDS can be perceived as CLIENT-SERVER, where the data store is the server and the data accessors are the clients.

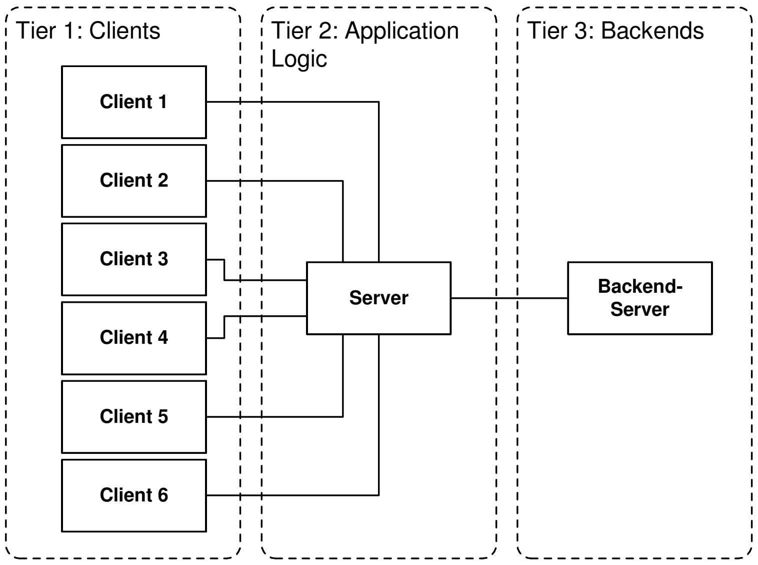

Figure 20: 3-tier client-server architecture: example

When CLIENT-SERVER is used in a distributed fashion, it can be extended so that the location of a remote component does not need to be hard-wired into the system. The pattern LOOKUP [KJ04, VKZ04] allows servers to register their remote components at a central service e.g. by name or property. Using the pattern LOOKUP, the client must only know the location of the lookup service instead of the potentially huge number of locations of the remote components it wants to communicate with. The LOOKUP pattern is thus an alternative to broadcast messages for getting initial references (e.g. using IMPLICIT INVOCATIONS). The problem that remains however is how to get the initial reference to the lookup component.

A general alternative to CLIENT-SERVER is PEER-TO-PEER:

Pattern: Peer-to-Peer

Consider a situation similar to that of a CLIENT-SERVER, but in contrast to CLIENT-SERVER, there is no distinction between the components: each component might both provide services and consume services. When a component provides a service it must perform well according to the demands of the requesting components. Each component must know how to access other components.

In the PEER-TO-PEER pattern each component has equal responsibilities, in particular it may act both as a client and as a server. Each component offers its own services (or data) and is able to access the services in other components. The PEER-TO-PEER network consists of a dynamic number of components. A PEER-TO-PEER component knows how to access the network. Before a component can join a network, it must get an initial reference to this network. This is solved by a bootstrapping mechanism, such as providing public lists of dedicated peers or broadcast messages (using IMPLICIT INVOCATION) in the network announcing peers.

Once an initial reference of the PEER-TO-PEER network is found, we need to find other peers in the network. For this purpose, each peer (or each dedicated peer) realizes the LOOKUP pattern [KJ04, VKZ04]. Using LOOKUP peers can be found based on their names or their properties. PEER-TO-PEER can be realized internally using CLIENT-SERVER, or other patterns. It usually also uses a BROKER architecture.

Whereas CLIENT-SERVER and PEER-TO-PEER concentrate on EXPLICIT INVOCATIONS, PUBLISH-SUBSCRIBE[4] is an interaction pattern that is heavily based on IMPLICIT INVOCATIONS:

Pattern: Publish-Subscribe

A component should be accessed or informed of a specific runtime event. Events are of different nature than direct interactions as in CLIENT-SERVER or PEER-TO-PEER. Sometimes a number of components should be actively informed (an announcement or broadcast), in other cases only one specific component is interested in the event. In contrast to EXPLICIT INVOCATIONS, event producers and consumers need to be decoupled for a number of reasons: to support locality of changes; to locate them in different processes or machines; to allow for an arbitrary time period between the announcement of interest in an event and the actual triggering of the event. Still, there must be a way to inform the interested components.

PUBLISH-SUBSCRIBE allows event consumers (subscribers) to register for specific events, and event producers to publish (raise) specific events that reach a specified number of consumers. The PUBLISH-SUBSCRIBE mechanism is triggered by the event producers and automatically executes a callback-operation to the event consumers. The mechanism thus takes care of decoupling producers and consumers by transmitting events between them.

An example of a PUBLISH-SUBSCRIBE architecture, where the PUBLISH-SUBSCRIBE mechanism is implemented as an independent subscription manager, is shown in Figure 21.

In the local context the PUBLISH-SUBSCRIBE can be based on the OBSERVER pattern [GHJV94], where the PUBLISH-SUBSCRIBE mechanism is implemented as part of the 'subject' (i.e. the event producer). For instance, most GUI frameworks are based on a PUBLISH-SUBSCRIBE model.

In the remote context PUBLISH-SUBSCRIBE is used in MESSAGE QUEUING implementations or as a pattern implementation of its own. PUBLISH-SUBSCRIBE makes no assumption about the order of processing events. This and other issues are solved by some messaging patterns [HW03] (see MESSAGE QUEUING explained below).

The PUBLISH-SUBSCRIBE pattern can be used in the context of the ACTIVE REPOSITORY pattern, so that accessors of data subscribe to the repository, which in turn notifies them when the data is updated.

[4] In [BMR+96] this pattern is called PUBLISHER-SUBSCRIBER and in [BCK98] it is called EVENT SYSTEM.

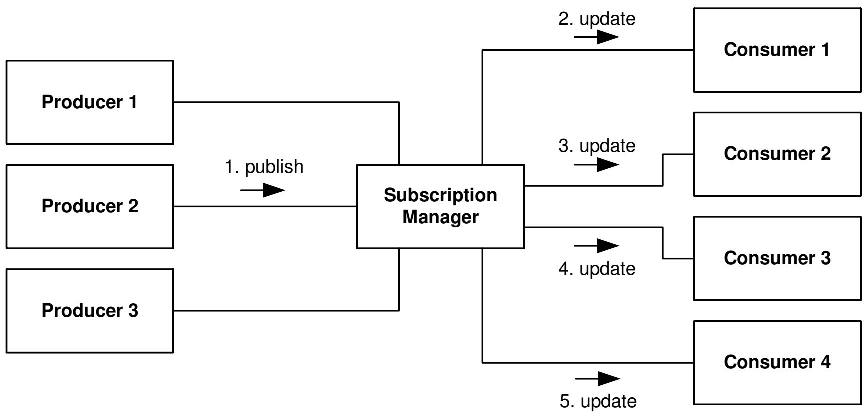

Figure 21: Publish-Subscribe example

PUBLISH-SUBSCRIBE is sometimes used to realize CLIENT-SERVER and PEER-TO-PEER: for instance, in distributed implementations of CLIENT-SERVER and PEER-TO-PEER it is necessary to bridge between the asynchronous network events and the synchronous processing model of the server. This can be done using a local PUBLISH-SUBSCRIBE model, where event handlers subscribe for the network events. This is described in detail in the REACTOR pattern [SSRB00].

In a similar way, PUBLISH-SUBSCRIBE models are used for integration tools, to ensure database consistency, and in user interfaces. For example, in the MODEL-VIEW-CONTROLLER pattern, the view and controllers subscribe to the model which then publishes any updates to them. Similarly in the PRESENTATION-ABSTRACTION-CONTROL pattern, higher-level agents subscribe to lower-level agents that handle the user interface. In general, PUBLISH-SUBSCRIBE offers a high potential for reuse and evolution, because it offers a strong decoupling between event producers and event consumers. There are also some potential problems, however: event consumers have to register for events, which is in general more complex than for instance a CLIENT-SERVER interaction. There is no guarantee that an event will be processed. Exchange of data is not as simple as parameter passing. Often a SHARED REPOSITORY must be used, which is slower than parameter passing.

10 Distributed Communication View

The Distributed Communication View shows how distributed components in a networked environment can communicate with each other. Hence, the main concerns in this view are how an architecture can support distributed component interaction and decoupling.

Thus all quality attributes relevant for distributed architectures must be considered in the Distributed Communication View. For instance, in a distributed components environment, Location Transparency is usually supported. Distributed components should also support Interoperability and Modifiability. The distributed invocations should offer the best possible Efficiency. Furthermore, Scalability and Reliability of the overall distributed system, or of specific distributed components, might need to be supported.

The components are physically located in different network nodes or processes. They interact with each other through connectors that pass invocations or data from one to another. The details of these interactions can be better explained in the Component Interaction View.

Pattern: Broker

Distributed software system developers face many challenges that do not arise in single-process software. One is the communication across unreliable networks. Others are the integration of heterogeneous components into coherent applications, as well as the efficient use of networking resources. If developers of distributed systems must overcome all these challenges within their application code, they may lose their primary focus: to develop applications that efficiently tackle their domain-specific problems.

A BROKER separates the communication functionality of a distributed system from its application functionality. The BROKER hides and mediates all communication between the objects or components of a system. A BROKER consists of a client-side REQUESTOR [VKZ04] to construct and forward invocations, as well as a server-side INVOKER [VKZ04] that is responsible for invoking the operations of the target remote object. A MARSHALLER [VKZ04] on each side of the communication path handles the transformation of requests and replies from programming-language native data types into a byte array that can be sent over the transmission medium.

The BROKER is a compound pattern that is realized using a number of remoting patterns [VKZ04]. The most foundational remoting patterns in a BROKER architecture are mentioned above: REQUESTOR, INVOKER, and MARSHALLER. There are many others. Some important examples are: a CLIENT PROXY [VKZ04] represents the remote object in the client process. This proxy has the same interface as the remote object it represents. An INTERFACE DESCRIPTION [VKZ04] is used to make the remote object's interface known to the clients. LOOKUP [KJ04, VKZ04] allows clients to discover remote objects.

The BROKER uses a LAYERS architecture. The layers of BROKER are also described in [VKZ04].

Many well-known BROKER realizations are based on the CLIENT SERVER pattern. However, the other patterns for component interactions, such as EXPLICIT INVOCATION, PEER-TO-PEER, MESSAGE QUEUING, and PUBLISH-SUBSCRIBE, can also use a BROKER to isolate communication-related concerns, when used in a distributed setting. Only in very simple distributed systems or in distributed systems with severe constraints (e.g. regarding performance and memory consumption), it might be advisable not to use a BROKER.

An example for a BROKER realization is depicted in Figure 22.

The BROKER can be seen as the general structure that utilizes the patterns from the Component Interaction View in a distributed setting. The following patterns [VKZ04] are a number of variants of CLIENT-SERVER that usually operate in a distributed setting and are mutual alternatives. They usually employ a BROKER architecture internally.

Pattern: Remote Procedure Calls

Consider the case where you want to realize an EXPLICIT INVOCATION in a distributed setting. The use of low-level network protocols requires developers to invoke the send and receive operations of the respective network protocol implementations. This is undesirable because the network access code cannot be reused, low-level details are not hidden, and thus solutions are hard to maintain and understand.

REMOTE PROCEDURE CALLS extend the well-known procedure call abstraction to distributed systems. They aim at letting a remote procedure invocation behave as if it were a local invocation. Programs are allowed to invoke procedures (or operations) in a different process and/or on a remote machine.

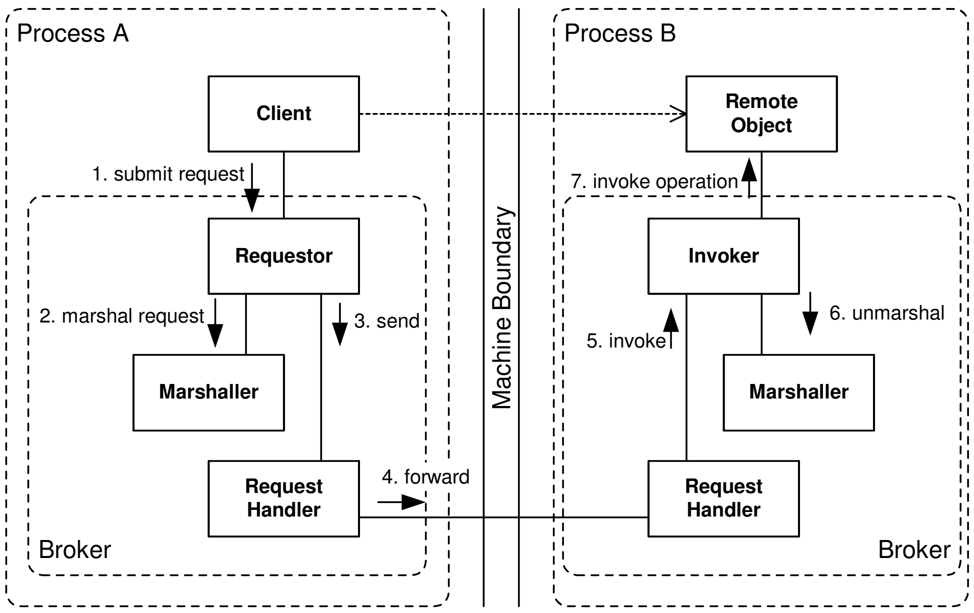

Figure 22: Example of a Broker architecture

REMOTE PROCEDURE CALLS leverage the CLIENT-SERVER pattern of interaction: a client invokes operations, and a server provides a well-defined set of operations that the client can invoke. To the client developer, these operations look almost like local operations. A major difference is that additional errors might occur during a remote invocation, for example because the network fails or the requested operation is not implemented by the server. These errors must be signaled to the client (see the pattern REMOTING ERROR [VKZ04]).

Pattern: Message Queuing

Consider a situation similar to that of REMOTE PROCEDURE CALLS, but it is necessary to decouple the sender from the receiver to realize queuing of invocations. Queuing is necessary, for instance, when temporal outages of the receiver should be tolerated or when heterogeneous systems should be integrated. For instance, when a legacy system using BATCH SEQUENTIAL should be integrated into a distributed system, only one invocation can be handled at a time by that system. Somewhere additional messages must be queued until the system is ready to process the next message.

Messages are not passed from client to server application directly, but through intermediate message queues that store and forward the messages. This has a number of consequences: senders and receivers of messages are decoupled, so they do not need to know each other's location (perhaps not even the identity). A sender just puts messages into a particular queue and does not necessarily know who consumes the messages. For example, a message might be consumed by more than one receiver. Receivers consume messages by monitoring queues.

MESSAGE QUEUING realizes CLIENT-SERVER interactions and implements IMPLICIT INVOCATION as the primary invocation pattern.

MESSAGE QUEUING architectures are explained in detail in [HW03] using a pattern language for messaging systems. An example of a MESSAGE QUEUING architecture is shown in Figure 23.

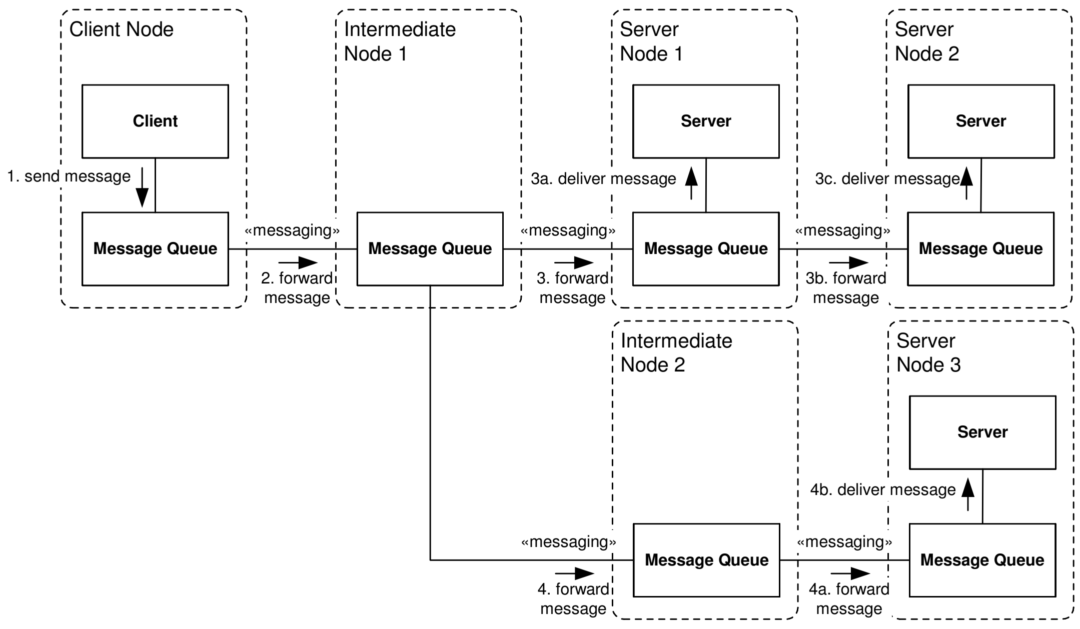

Figure 23: Message Queuing example

11 Epilogue

We have proposed to unite existing approaches of architectural patterns into a pattern language so that practitioners can benefit from a single comprehensive source of patterns. We emphasized on outlining the relations between the architectural patterns in order for them to acquire added value as a language, rather than a set of individual patterns. We also referenced the original sources for the patterns, so that interested parties can explore the rich details of each pattern.

Organizing the entire set of architectural patterns into a coherent pattern language is an immense amount of work; therefore we limited this paper to including patterns from the initial and fundamental catalogs and categorizations that deal with component and connector structures. We hope that other members of the community can give us feedback so that we can at least reach a consensus on these fundamental patterns.

Acknowledgments

We extend our thanks to the EuroPLOP ’05 writers’ workshop members: Diethelm Bienhaus, Frank Buschmann, Kevlin Henney, Wolfgang Herzner, Andy Longshaw, Klaus Marquardt, Amir Raveh for their feedback. Finally, we’d like to thank our EuroPLoP 2005 shepherd Lars Grunske.

References

[AIS$^+$77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jakobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language – Towns, Buildings, Construction.* Oxford Univ. Press, 1977.

[Ale79] C. Alexander. *The Timeless Way of Building.* Oxford Univ. Press, 1979.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, Reading, MA, 1998.

[BH03] Frank Buschmann and Kevlin Henney. Explicit interface and object manager. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPlop 2003)*, Irsee, Germany, July 2003.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-orinented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[CBB$^+$02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[Cop96] J. Coplien. *Software Patterns Management Briefing*. SIGS, 1996.

[Cou87] J. Coutaz. PAC, an Object Oriented Model for Dialog Design. In *Proceedings Interact'87, Stuttgart*, pages 431–436. IEEE Computer Society, 1987.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 12002.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in linda. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 255–263,, 1985.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GMSM00] J. Garcia-Martin and M. Sutil-Martin. Virtual machines and abstract compilers - towards a compiler pattern language. In *Proceedings of EuroPlop 2000*, pages 375–396, Irsee, Germany, July 2000.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[IEE00] IEEE. Recommended Practice for Architectural Description of Software Intensive Systems. Technical Report IEEE-std-1471-2000, IEEE, 2000.

[KJ04] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. J. Wiley and Sons Ltd., 2004.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[MM03] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, Helsinki, Finland, 2003. ACM Press.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[SC97] Mary Shaw and Paul C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13. IEEE Computer Society, 1997.

[SFHB05] Markus Schumacher, Eduardo Fernandez, Duane Hybertson, and Frank Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.

[Sha96] Mary Shaw. Some Patterns for Software Architechture. In John Vlissides, James Coplien, and Norman Kerth, editors, *Pattern Languages of Program Design, Vol 2*, pages 255–269. Reading, MA: Addison-Wesley, 1996.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Trans. Softw. Eng.*, 22(6):390–406, 1996.

[VKZ04] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns*. Pattern Series. John Wiley and Sons, 2004.

[Zac87] John A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 26(3):276–292, 1987.

[Zdu03] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[Zdu04] U. Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings Software*, 151(2):67–83, April 2004.