

Chapter 3: The Distributed Architecture of Multi-Tiered Enterprise Applications

In the previous chapter, we gave a general introduction to enterprise applications from a middleware point of view. We defined enterprise applications as transactional, distributed multi-user applications. We discussed underlying technology and multi-tier structures for enterprise applications. In this chapter, we have a closer look at the distributed architecture of enterprise applications. As a basis, we first define an *architectural style* for multi-tiered enterprise applications. In addition to the multi-tier structure, the style focuses on an application's *process topology* which consists of distributed processes, data stores, and client/server communication relationships. Usually, the design of an adequate process topology for an enterprise application is the task of a software architect. Typically, a design has to address a broad range of requirements, such as scalability, performance, availability, security, the need to integrate existing subsystems, or organizational and legal requirements. As a guide to the design of process topologies, we present a catalogue of patterns – so called *process topology patterns* – that form a pattern language for our architectural style.

This chapter is structured as follows: In Section 3.1, we give a general introduction to patterns and architectural styles for software systems. Section 3.2 then defines an architectural style specific to multi-tiered enterprise applications. Along with the style, we present several examples. In Section 3.3, we discuss process topologies and factors that typically influence their design. Subsequently, Section 3.4 presents a catalogue of patterns for designing process topologies. In Section 3.5, a few selected questions are discussed in more detail. And finally, Section 3.6 gives a brief summary of this chapter.

3.1 Introduction to Patterns and Architectural Styles

In this section, we introduce patterns, pattern languages, and architectural styles. First, we outline the work of Christopher Alexander who proposed patterns for traditional (building) architecture. Then we explain how Alexander's concept of patterns has been applied to software systems. Finally, we present the notion of architectural styles. This introduction is a prerequisite for sections 3.2 and 3.4, where we present an architectural style for multi-tiered enterprise applications and a pattern language for that style, respectively.

3.1.1 Christopher Alexander's Patterns

In the context of complex design problems, a *pattern* describes a common solution to a recurring problem¹. Originally, the notion of patterns was introduced by Christopher Alexander [Alex79] [AIS+77] in the context of urban design, building architecture, and construction. [AIS+77] presents a collection of 253 patterns that address a broad range of design problems. Examples are *promenade*, *hierarchy of open space*, and *cooking layout*. Alexander describes a pattern as follows:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [AIS+77]

¹ Note that, in other contexts, the word *pattern* might be used with a different meaning – for example, in the image processing domain.

In [Alex79], a pattern is defined more formally as a

“three-part rule, which expresses a relation between a certain context, a problem, and a solution.”

Such a three-part rule states that when problem *X* occurs in context *Y* then solution *Z* can be applied to solve the problem. Collections of related patterns are often referred to as *pattern languages*. A pattern language defines a common vocabulary and thus helps designers to better communicate problems and solutions. Also, designers can use different patterns of a pattern language and combine them in their particular designs.

3.1.2 Patterns for Software Systems

Later, the idea of patterns was successfully applied to software engineering. Early work includes [BC87], [Coad92], [Gamm92], and [John92]. For example, [BC87] present a language of five patterns for user interface design in the Smalltalk programming language, and [Coad92] proposes seven patterns, such as “broadcast” and “event logging”.

Design patterns [GHJV95] [Pree95] [BMR+96], which are used especially in object-oriented design, probably have received most attention so far. [GHJV95] define design patterns as

“descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”

They identify four main elements of design patterns: (1) name, (2) problem, (3) solution, and (4) consequences. For the presentation of design patterns, they use a more detailed template that consists of the following sections: pattern name and classification, intent, also known as, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, and related patterns. In [GHJV95], twenty-three design patterns are presented, for example, “factory method”, “singleton”, “iterator”, and “mediator”. The patterns are divided into three categories: creational, structural, and behavioral design patterns.

The pattern approach has been quickly adopted both in research and practice. Until today, practitioners and researchers have published numerous patterns for a broad range of systems and application domains – such as patterns for EJB-based systems [Mari02], systems with limited memory [NW00], business object frameworks [CCG00], as well as concurrent and networked software [SSRB00].

The patterns presented in [GHJV95] address different levels of granularity. For instance, occurrences of the “flyweight” pattern are typically more fine-grained than occurrences of the “mediator” pattern. Nevertheless, all patterns are referred to as design patterns by the authors. [BMR+96], in contrast, differentiates between three groups of patterns:

- *Idioms* (e.g., [Copl92]) are patterns with a low-level of abstraction. They address both design and implementation, and are often specific to a particular programming language. An example is the “counted-pointer” idiom [BMR+96], which is useful for C++ but not for languages/runtime environments with automatic garbage collection like Smalltalk or Java.
- *Design patterns* are patterns with a medium level of abstraction. Usually, they are independent of a particular programming language. Design patterns can have a strong impact on the architecture of a subsystem but have no significant impact on the architecture of the overall system. An example is the “observer” pattern described in [GHJV95].
- *Architectural patterns* address the highest level of abstraction and describe the architecture of an overall system. In [BMR+96], they are defined as follows: “An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.” Subsystems defined by an architectural pattern are often more

coarse-grained than classes or objects (which are typical elements of design patterns). For instance, a subsystem can represent an operating-system-level process. Examples of architectural patterns are the “model-view-controller” and “broker” patterns [BMR+96].

In the context of this thesis, particularly design patterns and architectural patterns are important.

Idioms, design patterns, and architectural patterns – they all address the structure of a software system. However, it should be noted that there are other types of patterns that are of interest in the context of software engineering. For example, analysis patterns [Fow197] deal with the structure of an application domain, and process patterns [Cop195] address software development processes, collaborative work, and organization.

In this subsection, we discussed *software patterns*, which are based on Alexander’s pattern concept. These software patterns comprise problem, context, and solution parts and are intended to encapsulate design knowledge. Note that in computer science, the term *pattern* is sometimes used with different meanings (e.g., pattern matching in strings or pattern recognition in the context of digital image processing). These patterns are completely different from software patterns described in this subsection.

3.1.3 Architectural Styles

Parallel to the research on patterns (and mostly independent of it), a research direction called “software architecture” emerged [PW92]. Similarly to patterns, software architecture also deals with the structure(s) of a software system. There are many definitions of software architecture. For example, the reference model of open distributed processing (RM-ODP) [ISO95a] defines architecture as

“a set of rules to define the structure of the system and the interrelationships between its parts.”

Or, in [GP95], software architecture is defined as

“the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.”

More specifically, the architecture of a software system is often treated as a system of *components and connectors*, such as in [GS93]:

“The framework we will adopt is to treat an architecture of a specific system as a collection of computational components – or simply components – together with a description of the interactions among these components – the connectors.”

In software architecture, the notion of *architectural styles* plays an important role [PW92] [AAG93] [GS93] [SG96]. In [SG96], an architectural style is described as follows:

“An architectural style (...) defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of component and connector types, and a set of constraints on how they can be combined.”

To design a concrete software system based on a given architectural style, one is thus required to select a specific configuration of components and connectors that fulfils the set of constraints defined by that style. To avoid confusion with low-level system configurations, we call these *architectural configurations*. Furthermore, it is necessary to provide implementations of all component and connector types for a concrete system. For several types of systems, architectural styles have been presented. Examples are “pipes and filters”, “layered systems”, and “repositories” [GS93] [SDK+95] [SG96]. Each style may have an arbitrary number of variations and sub-styles. A sub-style is a specialization of a style in the sense that components/connectors/constraints of the sub-style are more specific (i.e., more restricted) than those of the super-style. Thus, the set of all valid configurations of a sub-style is a subset

of the valid configurations of its super-style. For instance, “Unix pipes and filters” can be regarded as a sub-style of the “pipeline” style, which in turn is a sub-style of the more general “pipes and filters” style.

Like patterns, architectural styles are often described informally and as part of catalogues. In [SC97], a uniform descriptive standard and a classification framework for architectural styles are proposed. The authors suggest to focus on the following aspects and to use them as criteria for classification:

- types of components and connectors used in the style,
- how control is shared, allocated, and transferred among components,
- how data is communicated through the system,
- interaction of data and control (e.g., whether the topology of the data flow and the topology of the control flow are isomorphic), and
- type of reasoning compatible with the style.

There are also more formal approaches to architectural styles, which can be found, for instance, in [AAG93] and [AAG95].

While some systems are based on “pure” styles, many others use *combinations* of different architectural styles [SG96] [SC97]. There are several possibilities of combining styles:

- When different subsystems follow different styles, multiple architectural styles can coexist in a single system.
- Different styles can be used at different levels of granularity. For instance, a component that is a filter component of the “pipes and filters” style can be designed internally as a layered system.
- A system may also use a combination of several styles at the *same* level of granularity. In such cases, a component may use connectors of different styles. For instance, a component can be both a filter of the “pipes and filters” style and a client component of the “blackboard” style.
- A system can use a single style that combines properties of various (pure) styles.

What is the relationship between architectural styles and patterns? Typically, architectural styles are more coarse-grained and address a higher level of abstraction than design patterns (a comparison of architectural styles and design patterns can be found in [MKMG97]). Architectural styles roughly correspond to architectural patterns (which are also sometimes called *architectural design patterns*). For example, [SG96] use the terms “architectural style” and “architectural pattern” synonymously; and [BMR+96] present architectural patterns, such as *layers* and *pipes-and-filters*, which have previously been denominated as architectural styles in [GS93]. However, as both terms come from different communities, there are also minor differences: Patterns focus on mapping a problem to a particular solution (in a given context). By contrast, architectural styles describe common structures of families of systems – without necessarily emphasizing problem and solution parts. Also, architectural styles focus on the type of components/connectors and on constraints for designing systems.

In this thesis, we adopt the notion of *architectural styles* to describe the top-level architecture of a software system. The structure of subsystems may then be described by means of patterns, e.g., design patterns.¹

¹ Note that the distinction between architectural styles and design patterns (as well as the distinction between architectural patterns and design patterns) is fuzzy in some cases. The classification of patterns and styles in the literature is based on the *typical* level of abstraction/granularity addressed by a pattern/style. Nevertheless, in a concrete software system, the granularity of a particular instance of a pattern/style may differ from that assumed in the literature. For example, there are cases where a pattern typically regarded as a design pattern describes the top-level architecture of a system. Also, an architectural style (architectural pattern) might be used to describe the architecture of a subsystem only instead of the architecture of the overall system.

3.2 Architectural Styles for Enterprise Applications

In the previous section, we gave a general introduction to patterns and architectural styles, without focusing on a particular class of software systems. In this section, we examine how architectural styles can be used to describe the architecture of enterprise applications. First, Subsection 3.2.1 identifies three “pure” architectural styles that capture different relevant aspects of a typical enterprise application’s architecture. Then, in Subsection 3.2.2, we present a new architectural style for describing the *overall* architecture of multi-tiered enterprise applications. Our style combines elements of the three existing “pure” styles. Finally, Subsection 3.2.3 provides several examples of our architectural style.

3.2.1 Existing Pure Styles

In the literature, different architectural styles for many classes of systems have been presented, but there is no single style that adequately and in sufficient detail reflects the architecture of multi-tiered enterprise applications (which were described in Chapter 2). However, there are three styles that capture at least some important *parts* of the architecture:

- The “layered systems” style [SG96] (which corresponds to the “layers” architectural pattern in [BMR+96]) describes the architecture of a system as a set of layers $L_1..L_n$. Layers can be arbitrary collections of system elements (distributed and/or non-distributed). Each layer communicates with adjacent layers only: For all $i < n$, layer L_i is a client of layer L_{i+1} (i.e., consumes services from the adjacent layer below). Accordingly, for all $i > 1$, layer L_i is a server for layer L_{i-1} (i.e., provides services to the adjacent layer above). “Relaxed-layered-system” is a variant of this style, where layers may consume services from all layers below and may provide services to all layers above.
- The “client-server” architectural style [SG96] describes distributed systems that are organized into two tiers: An arbitrary number of client processes communicate with a central server process which provides services to the clients. Control typically flows from the clients to the server, whereas data typically flows in the opposite direction.
- The “repositories” style [SG96] describes architectures where an arbitrary number of client components (not necessarily distributed) operate on a central repository. The client components access data within the repository and are independent of each other, except for interactions via the contents of the repository.

Although the “layered systems” style in general does not specifically address distributed systems, it nicely reflects the organization of an enterprise application into multiple tiers. A tier is a specific type of layer that corresponds to a collection of distributed processes as defined by the operating system. While the multi-tier view is useful for providing a rough overview, it still hides many details and is usually too abstract to reason about properties such as scalability, availability, or security. For example, from Figure 2-2 or Figure 2-4 in Chapter 2, we cannot learn whether there are components that may become a bottleneck or represent a single point of failure. Except for the client tier(s), it is unclear whether a given tier consists of a single process or represents a collection of processes. Furthermore, when a tier consists of a collection of processes, it would also be important to know which processes of a given tier may directly communicate with which processes of an adjacent tier, e.g., 1:1, 1:n, n:1, or n:m.

In contrast to the “layered systems” style, the “client-server” style provides more details regarding distribution. However, categorizing processes strictly either as clients or as servers is suitable for simple two-tier enterprise applications only. In enterprise applications with multiple server processes and/or more than two tiers, the “client-server” style can describe only a subset of the distributed structure, i.e., a single server process in tier i and its client processes in tier $i-1$.

Finally, the “repositories” style reflects that, in enterprise applications, multiple clients access shared data. Transactional data stores play the role of repositories - but also processes of intermediate tiers can be

viewed as repositories for their respective clients. Like the “client-server” style, the “repositories” style is appropriate only for structures with two layers.

3.2.2 The Multi-Tiered Enterprise Application Style

Each of the three “pure” styles, “layered systems”, “client-server”, and “repositories” captures one specific, important aspect of the architecture of multi-tiered enterprise applications. In order to describe the *overall* architecture, we now present our architectural style that combines elements of the three styles outlined above. First, we define the types of components and connectors of our style. Then we define constraints for combining components and connectors into valid configurations. Examples are given in the following subsection.

In our “multi-tiered enterprise application” style, there are three types of components:

1. *Transactional data stores*

An instance of the first component type represents a transactional data store which persistently stores shared business data and offers transactional data access functionality to client processes. Examples are relational database management systems, object database management systems, and all kinds of other repositories that support transactional access to data.

2. *Processes*

An instance of the second component type represents a distributed, heavyweight process at the operating system level – see Section 2.4 for more details on processes. *Distributed* means that, while each process is located on exactly one machine, different processes may be located on different machines in a network¹. A process must be heavyweight, which means that lightweight threads do not qualify as a component. Processes that implement transactional data stores are also not regarded here because they are covered by the first component type.

We distinguish between two types of processes: application processes and infrastructure processes. An application process is defined as “a process which is specific to the requirements of a particular information system” [ISO95b]. Typical application processes contain application-specific business logic, presentation logic, and/or data access logic. In contrast to application processes, infrastructure processes are usually application independent, deal with technical infrastructure concerns, and provide services to the application processes. Typically, infrastructure processes are part of an operating system or middleware. Application processes are always represented as component instances in an architectural configuration of the “multi-tiered enterprise application” style. Infrastructure processes may be included or ignored, as required. For instance, a software architect may decide to include infrastructure processes that correspond to web browsers, web servers, and CORBA object request broker daemons. On the other hand, he may decide to ignore all infrastructure processes that deal with low-level details – such as logging, buffering, or graphical window management – because these processes typically do not play an important role at the architectural level.

3. *Tiers*

An instance of the third component type represents a tier, which is a set of components. Each component in the set is either of type process or of type transactional data store.

¹ In the literature, the term *distributed processes* refers to the distribution of a collection of processes (similar to the term *distributed objects*) and does not imply that an individual process is distributed among multiple machines.

There are two types of connectors for the “multi-tiered enterprise application” style:

1. *Client/server communication relationships between processes or between processes and transactional data stores (C1 connectors)*

An instance of the C1 connector type connects two components: The first component (client) is of type process; the second component (server) is of type process or transactional data store. A C1 connector represents a potential client/server communication relationship between the connected components. More specifically, it defines that

- client and server are able to communicate via a remote¹ communication protocol (but are not required to),
- the remote communication is direct, i.e., no intermediate components that are modeled as part of the architectural configuration are in the communication path,
- the server offers services to the client, and
- the client may consume these services (but is not required to).

The service offered by a transactional data store includes transactional access to persistent data (e.g., insert, update, delete, and lookup operations). The service offered by an application process includes access to transactional data and/or execution of functions that access transactional data. The implementation of a service in a process may in turn rely on services the process consumes as a client of other processes and/or data stores. There are two approaches to implementing a service in a process: *stateful* and *stateless*. With the stateful approach, a process maintains state information between client requests. With the stateless approach, no state information is maintained.

The C1 connector type does not define a remote communication protocol or a specific communication style (such as synchronous or asynchronous communication). The only requirement is that communication in both directions is supported.

2. *Client/server relationships between tiers (C2 connectors)*

An instance of the C2 connector type connects two components of type tier – a client tier and a server tier. The C2 connector type corresponds to the connector type defined for the “layered systems” architectural style: The client tier consumes services provided by the server tier. The C2 connector type does not define a specific interaction style or communication protocol.

In the following, we use the term *server* to denote a component that is either of type transactional data store or of type process.

Having defined the components and connectors types of our architectural style, we now define how instances of the components and connector types can be combined into specific configurations. The following constraints apply:

- (a) A tier must contain at least one component (application process or transactional data store).
- (b) Each process and each transactional data store is included in exactly one tier.
- (c) A C1 connector connects a client component in tier T_1 and a server component in tier T_2 (with $T_1 \neq T_2$) \Leftrightarrow a C2 connector connects T_1 and T_2 such that T_1 is client tier of T_2 .

¹ In addition to remote communication, operating systems typically offer mechanisms for communication between processes located on the same machine. For example, pipes, shared memory, local files, and signals can be used for local inter-process communication [Gray03]. In this thesis, we focus on remote communication, which is more general because it can be used for communication between both local and remote processes. Nevertheless, when there are collocated processes in a concrete enterprise application, an implementation might make use of local inter-process communication mechanisms to optimize communication.

- (d) For each process P , there is at least one C1 connector with P as a client.

Let PT be a directed graph with instances of the process and transactional data store component types as vertices and C1 connectors as directed edges (direction from the client component to the server component). We call PT an enterprise application's *process topology*.

- (e) PT must be acyclic and connected.

Let TT be a directed graph with instances of the tier component type as vertices and C2 connectors between the tiers as directed edges (direction from the client tier to the server tier). We call TT an enterprise application's *tier topology*. From (a), (b), (c), and (e), it follows that

- (f) TT is acyclic and connected.

The “multi-tiered enterprise application” style combines elements and properties of the “layered systems”, “client-server”, and “repositories” styles: Tiers and C2 connectors directly stem from “layered systems”. Both totally ordered multi-tier structures and partially ordered multi-tier structures can be modeled (see Section 2.4). Processes, transactional data stores, and C1 connectors correspond to the elements of the “client-server” style. In addition, arbitrary client/server topologies with the shape of directed acyclic graphs are allowed instead of two tier topologies with a central server only. And finally, each transactional data store and each process can be viewed as repository components (of the “repositories” style) that provide their clients with access to shared (transactional) data.

Our definition does not permit cyclic graphs. Cycles in a topology would allow a process to be a client of itself (directly or indirectly via other processes), which increases complexity and is often not desired. For that reason, many typical enterprise applications – especially those built with a strong emphasis on multi-tier structures – have *acyclic* process topologies. In this thesis, we therefore consider only directed acyclic graph (DAG) topologies. For enterprise applications with cyclic process topologies, a different architectural style could be defined (also see the discussion in Subsection 3.5.1 and future work in Section 8.2)

The “multi-tiered enterprise application” style is a relatively abstract style because many aspects (such as communication via C1 connectors) are not specified in detail. Due to the high level of abstraction, our style covers a broad range of systems. For more detailed and thus more specific and restricted architectures, appropriate sub-styles can be defined. For instance, a sub-style could specify that C1 connectors represent synchronous remote object invocations from client processes to servers. Another sub-style could be defined for systems where application processes disseminate new and updated data items asynchronously and on a best effort basis to directly connected clients via a broadcast protocol.

In summary, systems that follow the “multi-tiered enterprise application” style have the following main architectural characteristics:

- There are two directed acyclic graph (DAG) structures: the tier topology and the process topology. The tier topology describes tiers and client/server relationships between them. The process topology describes processes, transactional data stores, and client/server communication relationships. The process topology details the tier topology and, at the same time, is constrained by the tier topology.
- Control typically flows from client components to server components through the process topology; the topology of the control flow is a DAG.
- The topology of the control flow and the topology of the data flow are isomorphic to each other.
- Data tends to flow in the opposite direction, i.e., from server components to client components (although data flow from clients to servers is also allowed, e.g., when users insert new business data into the system).

Which type of reasoning is compatible with our architectural style? As our style combines elements of three pure styles, it also inherits the types of reasoning from them, e.g., data integrity from the “repositories” style and levels of service/modularity from the “layered systems” style. The process topology part of an architectural configuration is especially suited to reason about properties that are of interest from a middleware point of view, e.g., performance, scalability, and fault tolerance (see Section 2.3). However, as is the case with many other architectural styles, “reasoning” does not imply that system properties can simply be derived from an architectural style and a configuration. Instead, many properties depend on a broad range of implementation details and system parameters, and vary in different scenarios. For instance, let us consider the introduction of a new intermediate tier into a multi-tiered enterprise application and its effects on user response time:

- On the architectural level, we might conclude that the new tier introduces more latency for requests issued by users of the system (assumed that requests frequently have to traverse the process topology from the client application processes to transactional data stores).
- On the implementation level, we might take into account that the new tier was introduced to cache frequently accessed business data. Then we would expect a better average user response time.
- In a concrete scenario, we might find out that client access to data is evenly distributed and the size of the cache is small compared to the size of the complete set of business data. As a result, caching has no significant positive effect on average response time. Thus, the effect is that the new intermediate tier simply introduces more latency.

The example demonstrates that an architectural style and an architectural configuration alone are not sufficient to reason about certain system properties. It is often the case that reasoning makes sense only in the context of specific implementations and usage scenarios. Based on our architectural style, Section 3.3, Section 3.4, and the following chapters provide a more detailed discussion of various aspects and system properties of multi-tiered enterprise applications.

3.2.3 Examples

In this subsection, we provide several example architectural configurations of systems that follow the “multi-tiered enterprise application” style. The examples illustrate how our architectural style is applied to real-world systems and demonstrate that it covers a broad range of different systems. Table 3 shows the graphical notation we use for representing configurations:






Element	Symbol
transactional data store	
process	
tier	
C1 connector	
C2 connector	

Table 3. Graphical notation for representing configurations.

Example 1

An enterprise application consists of n clients and a central relational database. Each client runs as an application process on a separate desktop machine. Clients are implemented in C++ and access the

database via the ODBC [Geig95] protocol. The architecture of the enterprise application is based on a plain two-tier structure: Clients belong to the first tier and the central database represents the second tier. The architectural configuration of the given enterprise application is shown in Figure 3-1.

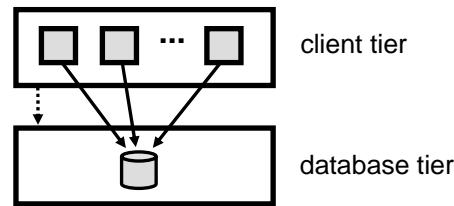


Figure 3-1. Example configuration of a simple two-tier enterprise application.

Example 2

Another enterprise application is structured into three tiers: client tier, middle tier, and database tier. The clients are rich (fat) clients implemented in Java and run on personal computers of their respective users. The clients use an implementation of the Java Message Service to exchange XML-formatted [BPSM00] messages with a central application server process located in the middle tier. The application defines several different message types, e.g., for retrieving data from the server, delivering data to a client, or requesting the server to update a given data object. The application server process, also implemented in Java, uses a proprietary protocol to communicate with three object database instances that form the database tier. Each database runs on a dedicated machine and contains a different set of data. The application server process provides its client with an integrated view of data. The architectural configuration for this example is shown in Figure 3-2.

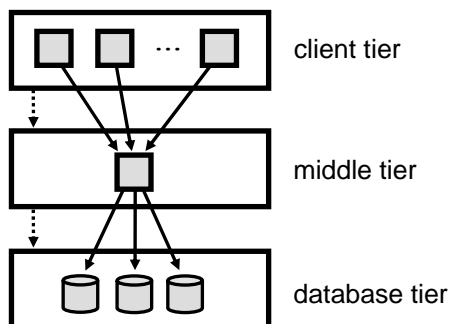


Figure 3-2. Example configuration of a three-tier enterprise application with multiple data stores.

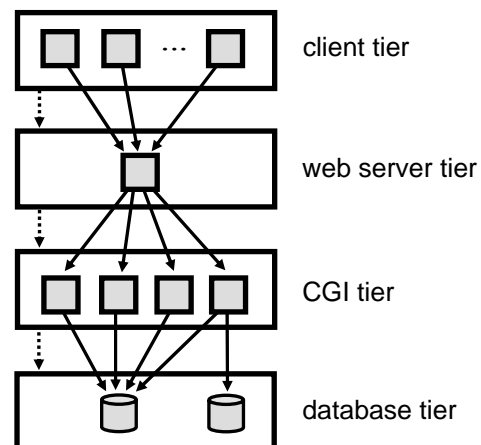


Figure 3-3. Example configuration of a web-based enterprise application with four tiers.

Example 3

The third example describes a four-tiered, web-based enterprise application. The client tier consists of web browser processes. The browser processes communicate via the HTTP protocol [FGM+99] with a central web server (which is modeled as a separate tier in this example). The web server delivers dynamically created HTML [RLJ99] documents to its clients. The web server delegates the task of dynamically creating documents to a number of other machines: For each client request, the web server

communicates through the FastCGI [Open96] protocol with one of several CGI application processes that execute Perl scripts [COW00]. On demand, the Perl scripts access databases in the last tier with the help of the Perl DBI module [BD00], generate HTML documents from persistent (transactional) data, and pass the documents back to the web server. Three of the CGI processes implement the same function and access only one of the two relational databases in the database tier. The web server can distribute the load of generating certain documents among those three processes. Other documents are handled by a fourth CGI process that implements another function and requires subsequent access to both databases. The architectural configuration of this enterprise application is depicted in Figure 3-3.

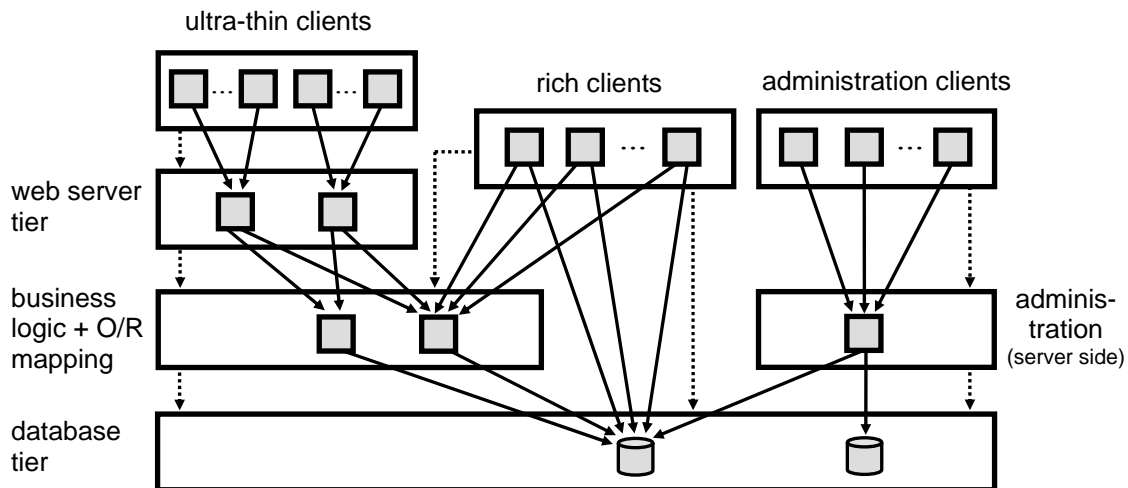


Figure 3-4. Example configuration of an enterprise application with a partially ordered multi-tier structure.

Example 4

The first three examples describe systems with a totally ordered multi-tier structure where each tier communicates with adjacent tiers only. In this example, we examine a *partially ordered* multi-tier structure. We further detail the enterprise application that is outlined in Section 2.4 and shown in Figure 3-4.

The ultra-thin clients of the application are web browser processes that use HTTP to access web servers in the web server tier. There are two web server processes; one of them handles ultra-thin clients from the Internet and the other serves ultra-thin clients in the intranet of the organization. The web servers are implemented in Java; each of them runs Java servlets [HC01] within the same process. The servlets generate HTML documents based on business data that is represented as EJB entity bean [CFF02] instances. The instances are managed by J2EE [Sun03a] compliant application server processes in the “business logic and O/R mapping” tier. For load balancing and availability reasons, two application server processes are employed. The servlets use Java RMI [Gros01] to transactionally access the remote bean instances. The application server processes map the entity bean instances to relational data stored in a relational database of the database tier. JDBC [Rees00] is used for accessing the database. The second type of clients, rich clients implemented in Java, mostly access entity beans of the “business logic and O/R mapping” tier. Instead of RMI, CORBA/IIOP [OMG02] is used as communication protocol. Each of the rich clients accesses one of the two application server processes. For a couple of performance-critical statistical functions, rich clients also access a relational database directly via JDBC. The third type of clients, administration clients, are implemented in Java and use plain TCP sockets for communicating with a central server process that processes all administration tasks, such as initiating backups, user management, or low-level access to relational data. The server process is implemented in C++ and accesses both databases of the enterprise application via an SQL/CLI implementation. Figure 3-4 shows the architectural configuration of this enterprise application.

Example 5

The last example demonstrates a number of less commonly used – but nonetheless valid – configuration options:

- (a) A transactional data store can belong to a tier that is not a leaf node in the tier topology.
- (b) A C1 connector can connect a client process and a server that belong to the same tier.
- (c) A client process that implements a user interface can belong to a tier that is a server tier for another tier.

The example is taken from the domain of federated database systems [BKLW99] [SL90]. Figure 3-5 shows the architectural configuration of the system. The software architect decided to structure the application into three tiers: client tier, federation tier, and database tier. All client processes of the client tier are connected with a central server process in the federation tier. The central server process accesses business data from three different data sources (database systems) in the database tier. The databases use different schemas. The central server process integrates schemas and data from the data sources and provides its clients with a unified view of business data. Meta data used by the central server process (e.g., schema and mapping information) is stored in a separate data store. The software architect decided to place that data store in the federation tier instead of the database tier (see option a) because he wanted the federation tier to contain all federation-related components. The central server process uses wrappers to access business data in underlying data sources. Two of the data sources are accessed directly because the corresponding wrappers run inside the central server process. The third wrapper is a heavy-weight component that runs within a separate process that belongs to the federation tier, too (see option b). In addition to the three data sources, the database tier contains a process that is used by an administrator for low-level access to data (see option c). The architect placed that process in the database tier instead of the client tier because he regarded it as part of the database domain rather than the client domain.

This example shows that often there is more than one possibility of defining individual tiers and the tier topology for an enterprise application. For instance, other software architects (who do not focus on the federation aspect) may decide to place individual servers in other tiers or may define a different tier topology for the same enterprise application.

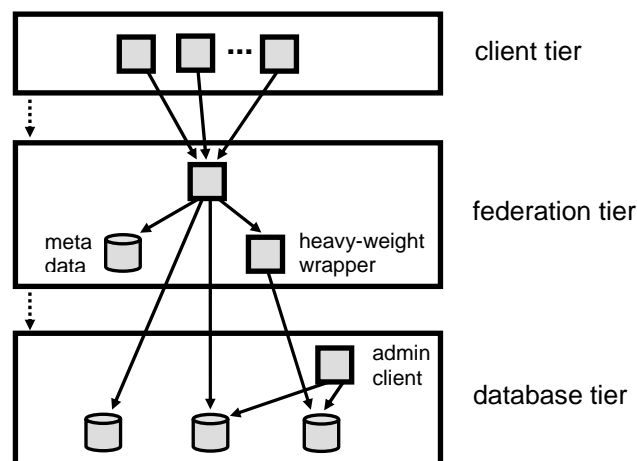


Figure 3-5. Example configuration of a federated database system.

3.3 Distributed Process Topologies

In Subsection 3.2.2, we introduced process topologies, which consist of processes, transactional data stores, and client/server communication relationships. Usually, the design of an appropriate process topology for an enterprise application is the task of a software architect. In this section, we discuss process topologies and various factors that typically influence the design of process topologies.

In the previous section, we defined a process topology as a directed graph of processes, transactional data stores, and C1 connectors. As tiers and C2 connectors are not included, a process topology is a *subset* of a specific architectural configuration for the “multi-tiered enterprise application” architectural style. A process topology of a system that adheres to the “multi-tiered enterprise application” style is always acyclic (i.e., is a DAG) and connected – see constraint (e) in Subsection 3.2.2. From the definition of C1 connectors and constraint (d) it follows that inner nodes of the DAG are processes and leaf nodes are transactional data stores. Examples of process topologies are shown in Figure 3-6.

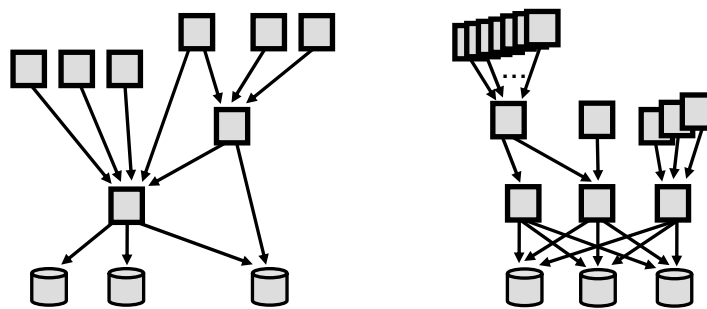


Figure 3-6. Two example process topologies.

A process topology describes the distributed structure of an enterprise application as a network of nodes and C1 connectors. Our process topology view can be seen as a small and very specific subset of the process view in Kruchten's "4+1" view model [Kruc95] or of deployment diagrams in the Unified Modeling Language (UML) [OMG03]. Note that a process topology does not necessarily correspond to the physical structure of the underlying communication network. Instead, a process topology describes a logical network, i.e., an overlay network built on top of the underlying communication network.

Traditionally, the design of a suitable process topology for an enterprise application is the task of a software architect who decides on distribution and other high-level design details early in the development process. In this section, we outline the most important driving forces a software architect has to take into account when designing a process topology.

One of the most basic process topologies is a structure that consists of n client processes (for users of the enterprise application) and a central data store. The first example in Subsection 3.2.3 describes an application with such a process topology. Sometimes, such a simple standard topology is sufficient – especially when the enterprise application is not mission-critical, runs in a local area network, has few users, and load is not an issue. However, in many cases, more advanced topologies (such as those given in Figure 3-6 or the fourth example in Subsection 3.2.3) are needed to fulfill the requirements of applications that are more demanding. The following list outlines the most important driving forces that frequently influence the design of process topologies:

F1. Performance, Scalability, and Fault Tolerance Requirements

Large-scale enterprise applications built on simple two tier topologies (e.g., like the first example in Subsection 3.2.3) are likely to run into severe problems: First, with many concurrent users and/or heavy load generated by users, the central data store becomes a bottleneck. Users are likely to experience bad performance (response time) and the enterprise application is not scalable with respect to the number of concurrent users. Second, the central data store is a single point of failure. When the data store is down or

unavailable (e.g., due to a software failure, network problems, or maintenance tasks), the whole enterprise application is immediately affected.

To improve performance, scalability, and fault tolerance, a software architect can, for example, add data stores to the topology or insert one or more layers of intermediate processes between client processes and data stores.

F2. Underlying Hardware, Network Topology, and Bandwidth of Connections

A process topology for an enterprise application should take into account the intended physical distribution of processes among machines, properties of these machines, the underlying network topology, and bandwidth that is available for individual connections. For example, replication of processes for better availability is not very effective when these processes are deployed on the same machine or on different machines that communicate with clients via a single router: Although the processes are replicated, a single machine or a single router still represent single points of failure. Another example is the introduction of an intermediate process for caching between a client and a data store: Reducing load on the data store through caching can be a significant advantage but does not make sense when all components are located in a single network segment and the (shared) bandwidth is the limiting factor in the system.

F3. Restrictions Imposed by Middleware, Frameworks, and Tools

Usually, enterprise applications are not developed completely from scratch. Instead, they rely on existing software such as middleware, third-party application components, or business object frameworks. In addition, most projects employ a collection of tools to support development and administration tasks. For example, there are tools for testing, modeling, code generation, automatic distribution and deployment, or remote administration. Existing software and tools may impose restrictions on the process topologies that can be used. For example, reusing an existing application component that relies on direct access to a relational database makes it difficult to insert intermediate processes between the process that executes the component and the database. Another example is an integrated development environment with built-in support for three-tier thin client architectures based on Java Server Pages [Berg02] and EJB entity beans. Although, in principle, it is possible to build any process topology with the IDE, only a small subset of topologies is well-supported; integrated tools for debugging, documentation, configuration, profiling, and code generation work only for a small set of pre-defined (three-tier) topologies.

F4. Organizational and Legal Concerns

Distribution is often a result of organizational requirements. For example, organization *A* (a travel agency) runs an enterprise application that uses, aggregates, and enhances services provided by enterprise applications of organizations *B* (a car rental agency) and *C* (an air line). Although, technically, *A*, *B*, and *C* could use a single, integrated enterprise application, they will typically decide to keep their applications separate in order to preserve their autonomy. This way, each organization is responsible for hardware issues, administration, software updates, and security of their respective subsystems and thus remains (relatively) independent of the others.

Moreover, in some cases, distribution is not only an option but also a legal requirement. For example, both tax offices and social insurance agencies are required to store and maintain certain data. Although it might be legal for an application to access data from both organizations (under strict constraints), both data sets have to be kept strictly separate and must not be stored in a single database.

F5. Heterogeneity and Integration

When software components have been developed in different programming languages and/or rely on different runtime environments (e.g., a Visual Basic component and a Java component), it is often difficult to run them as part of the same application process. Instead, it is much easier to run them in

separate application processes that communicate with each other. Furthermore, software components that have been developed (or compiled) for different operating systems or processor architectures usually have to be deployed on different machines in order to use them. Also, some existing software components may be developed for a specific (part of a) process topology. For example, a component responsible for order tracking may require exactly one direct connection to a data stores. When such software components are reused in an enterprise application, this narrows the range of options for selecting a process topology.

Also, it is not uncommon that a complete existing enterprise application with a given process topology has to be integrated into a new enterprise application. An architect may decide to leave the existing application (including its hardware, system configuration data, and process topology) as it is and integrate it as a subsystem into a new enterprise application. The same applies when multiple existing enterprise applications are to be integrated, e.g., when three online reservation systems are integrated because their respective enterprises cooperate.

F6. Security

In many cases, security considerations influence the design of a process topology. For instance, it is very risky to permit client processes of Internet users to directly access data stores of an enterprise application. Even with an ideal authentication and authorization mechanism, a malicious client would still be able to perform denial of service attacks directly on internal machines. To simplify authentication and authorization and to shield internal servers, Internet clients often are required to communicate with dedicated machines/processes that act as facades to other (internal) machines/processes of the enterprise application.

Also, when security is an important issue, machines can be assigned to different security zones (e.g., internal, demilitarized, external). In that case, a process topology and its processes have to be designed with that structure in mind.

F7. Costs for Hardware and Maintenance

Costs, e.g., for hardware, deployment, and maintenance, are almost always an important factor. For instance, an enterprise application based on a topology with few processes and no redundancy is usually much easier (and thus less costly) to deploy and maintain than an enterprise application with a large, complex process topology.

F8. Modularity

Sometimes, also the desire for a modular design influences the shape of a process topology. For instance, when data is distributed among multiple data stores, a software architect may want to shield clients from the complexities of accessing distributed data. A particular modular solution is to introduce one or more intermediate processes that handle access to distributed data and present an integrated view to the clients.

Depending on the complexity and requirements of the enterprise application, none, some, or all of the forces above have a significant impact on the design of a process topology. Unfortunately, there are no precise rules or an algorithm a software architect could follow for constructing complete, suitable process topologies. There are several reasons for that:

- A design usually requires a trade-off between various *conflicting* driving forces such as security, availability, performance, or cost. For instance, a process inserted (for security reasons) between a group of clients and the rest of the topology may impair response time for clients. Or, replication of a data store might help to better distribute load but is not well supported by development tools used in a given project. Costs frequently conflict with most other forces. Finding a reasonable trade-off is often non-trivial and depends heavily on the particular context of a project.

- An additional problem is that many important properties – such as autonomy, security, or modularity – cannot be easily quantified.
- As already discussed in Section 3.2.2, the architectural style and the architectural configuration alone is generally not a sufficient basis for predicting system properties. Instead, there are also a large number of system parameters, implementation details, and usage scenarios to be considered. For example, the average response time of a client could depend on client access patterns, locking strategy, remote communication mechanism, data store performance, cache replacement strategy, swapping and scheduling of operation systems, and many other factors. What is even worse, the effects of most of these factors cannot be analyzed separately because, typically, there are complex interdependencies between them.

However, it is often neither required nor desired to construct a complete process topology for an enterprise application from scratch. Instead, in many situations, an existing enterprise application with a given process topology evolves and has to be adapted. Although, in such cases, it is technically possible to restructure the complete existing process topology, a software architect will typically try to restrict his changes to certain *parts* of the topology only. Heuristics (patterns) for changing parts of topologies are presented in the following section. Evolution and adaptability are discussed in more detail in Chapter 3-4.

3.4 A Pattern Language for Process Topologies

In the previous section, we outlined the most important driving forces that influence the design of process topologies. Along with the driving forces, we provided several examples to illustrate the effects of the forces. In addition, we explained why, in the general case, there are no precise rules or an algorithm for constructing suitable process topologies. However, that does not mean that finding or changing a process topology depends solely on the creativity of a software architect. While it is difficult to design a *complete* process topology, there are heuristics for designing and changing *parts* of a process topology: In many enterprise applications, similar topology-related problems occur. Over time, software engineers have developed best practices to address those problems and thus typical problems are often solved in the same way. Analogically to *design patterns*, we call those typical solutions *process topology patterns*. In this section, we introduce the notion of process topology patterns and present several typical topology patterns, which can be found in many enterprise applications. The patterns form a pattern language specific to our “multi-tiered enterprise applications” style.

We define *process topology patterns* as software patterns (see Section 3.1) that focus on the description and/or transformation of (parts of) process topologies.

At first glance, descriptions and transformations seem to be two completely different concepts: While a description captures a static situation, a transformation deals with (rules for) changing one static situation into another one. However, in the context of software patterns and conventional patterns (as described in the literature), it is often hard or impossible to separate these two aspects. The reason for this is that, by definition, a pattern comprises a problem and a solution part. The problem part of a pattern is usually described in the context of a system (a static situation) – may it be an imaginary system (“if both components would communicate directly, then ...”) or a real system that is to be improved by a pattern. Problem and solution parts of a software pattern both represent static situations. The description of how to change a problem situation into a solution, i.e., how to implement a pattern, represents a transformation. In summary, we can say that problem, context, solution, description of a situation, and transformation are all essential parts of a software pattern. However, depending on the application domain and typical usage of a particular pattern, a description of that pattern may emphasize one or more of these aspects.

In the following subsections, we present a catalogue of several process topology patterns. The description of a process topology pattern details

- rules for selecting a subset of an enterprise application’s process topology,
- rules for transforming the given subset (and, thus, implicitly the complete process topology), and
- (possible) advantages of the transformed topology over the initial topology.

If essential, variants of the pattern and an outline of implementation options are described, too.

In contrast to typical design patterns, we decided to present process topology patterns with a special focus on transformations because:

- We wanted to emphasize the evolutionary aspect. Evolution of an enterprise application often includes and requires evolution of its process topology. Then the application of a process topology pattern represents one of many incremental changes to a process topology. In Chapter 4, the evolutionary aspect is discussed in more detail.
- A pattern is much easier to present and understand as a transformation because the reader can compare system properties of an initial system with system properties of a transformed system.
- Transformations are a good conceptual basis for using *combinations* of process topology patterns. Using a combination of multiple patterns can be described as a sequence of transformation steps. More details on combining process topology patterns can be found in Chapter 4.

Nevertheless, it is important to keep in mind that the effectiveness of a process topology pattern in a concrete enterprise application finally depends on the details of the underlying implementation. Usually, there are many different ways to implement a process topology pattern, each with specific advantages and drawbacks (e.g., eager replication versus lazy replication in case of the *data distribution* pattern). If any, a description of a process topology pattern outlines only the most important options for an implementation. More specific design and implementation details can be found in Chapter 5 and Chapter 6.

How do process topology patterns relate to patterns and architectural styles? They are more fine-grained than architectural styles (architectural patterns) because a transformation affects only a part of a system and its topology (instead of the overall system). On the other hand, they are much more coarse-grained than typical design patterns because they focus on distributed processes instead of classes and objects. In our thesis, we consider process topology patterns as high-level, non-object-oriented design patterns.

In the following, we present a pattern language of six typical topology patterns that can be found in many enterprise applications. The patterns are intended to help software architects to design appropriate process topologies for their enterprise applications. The pattern language is not *complete*, i.e., it neither contains all possible patterns nor is it a basis for constructing all possible process topologies. We expect that there are many other useful topology patterns in addition to those covered in this thesis.

3.4.1 Pattern 1: Wrapper Insertion

The *wrapper insertion* pattern inserts a new process between a set of clients and their server. This pattern is a process version of the adapter design pattern in [GHJV95].

Given

- a non-empty set of processes $\{C_1..C_n\}$ (“client processes”),
- a server¹ S (“wrapped component”), and
- a set of C1 connectors $\{E_1..E_n\}$, where each E_i ($1 \leq i \leq n$) is a connector from C_i to S .

¹ As already mentioned in Subsection 3.2.2, a *server* denotes a component of type transactional data store or process.

The transformation for this pattern is defined as follows:

- A new process W (“wrapper process”) is added to the topology.
- Each E_i ($1 \leq i \leq n$) is replaced with a C1 connector from C_i to W .
- A new C1 connector from W to S is added.

An example transformation is shown in Figure 3-7.

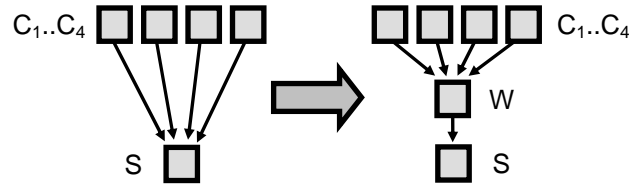


Figure 3-7. Example of the *wrapper insertion* pattern.

Note that it is not a requirement for all client processes of S to take part in the transformation; a subset is sufficient. A wrapper process that provides the same interface to the client processes as the wrapped component did is also called a *proxy process* (see proxy design pattern in [GHJV95]). Potential advantages of using the *wrapper insertion* pattern are:

Vertical load distribution. The pattern allows for vertical load distribution because tasks can be shifted from the wrapped component to the wrapper process. For example, the wrapper process may cache data from the wrapped component, be responsible for authentication and authorization of clients or encryption and decryption of data sent to/from clients or perform a pre-evaluation of client requests to reject invalid requests before they enter the core system.

Bandwidth and client response time. A wrapper process that is placed “near” the clients may improve the clients’ response time and/or reduce network traffic to the wrapped component. For instance, when the wrapper process caches data from the wrapped component and connections between clients and the wrapper process have more bandwidth and less latency than the connection between the clients and the wrapped component had, then client access to cached data can be performed much more quickly. The effect is comparable to that of a local, caching web proxy that is employed by many organizations to reduce external network traffic and to improve response time of at least a subset of client requests. Especially in situations where clients require better performance, but the wrapped component belongs to another organization (that is unable or unwilling to upgrade its hardware or to replicate the wrapped component by applying Pattern 2), caching of data in the wrapper process can be an effective way to improve client performance.

Modifying an existing service. The *wrapper insertion* pattern may also be used for adding functionality to or modifying an existing service without having to touch the component that provides the original service. This can be important, e.g., when the wrapped component is part of a running production system that must not be shut down or modified because other systems rely on it.

On the downside, the introduction of an intermediate process may increase client response time, for instance, if requests frequently have to traverse the complete path of processes from client processes to transactional data stores.

3.4.2 Pattern 2: Process Replication

The *process replication* pattern replicates a given process, for example, to improve fault tolerance or to distribute load.

Given

- a process R ,
- a non-empty set of processes $\{C_1..C_n\}$ (“client processes”),
- a set of C1 connectors $\{E_1..E_n\}$, where each E_i ($1 \leq i \leq n$) is a connector from C_i to R ,
- a non-empty set of servers $\{S_1..S_m\}$ and
- a set of C1 connectors $\{F_1..F_m\}$, where each F_i ($1 \leq i \leq m$) is a connector from R to S_i .

The pattern’s transformation is defined as follows:

- A new process R' is added to the topology.
- For each E_i ($1 \leq i \leq n$), either (1) a new C1 connector from C_i to R' is added or (2) E_i is replaced with a C1 connector from C_i to R' or (3) nothing changes. At least one C1 connector has to be added or replaced. It is not permitted to replace all E_i .
- For each F_i ($1 \leq i \leq m$), a new C1 connector from R' to S_i is added.

Two example transformations are shown in Figure 3-8.

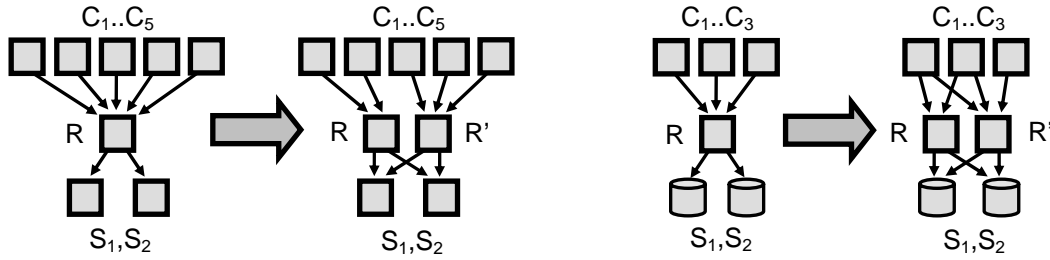


Figure 3-8. Two examples of the *process replication* pattern.

The transformation above covers the case that one replica is added. Transformations for two or more replicas can be defined accordingly. Potential advantages of using the *process replication* pattern are:

Load distribution. This pattern enables load distribution [CT88] [KGR94] [SKS92] as client requests can be distributed horizontally among the replicated processes. Ideally, n new replicas lead to an n -fold increase in throughput. Static or dynamic load balancing may be employed. In the latter case, clients may select a server process per transaction or per request within a transaction.

Exploiting locality. Clients may prefer to communicate with a replica that is located “near” to them to exploit locality.

Fault tolerance. Together with a fault tolerance mechanism that allows clients to switch to other replicas when a replica fails, availability can be improved (see [Gärt99]).

Potential drawbacks are higher maintenance costs and communication overhead to synchronize state information kept within the replicated processes. When at least one C1 connector from a C_i to R' is added as part of the transformation, an application of the *process replication* pattern also comprises an application of the *meshing* pattern (Pattern 3).

3.4.3 Pattern 3: Meshing

With the *meshing* pattern, one or more redundant C1 connectors are added to the process topology in order to create alternative paths of client/server communication relationships.

Given

- a process C ,
- a component S , and
- a path of C1 connectors $E_1..E_n$ ($n \geq 1$) from C to S .

The pattern's transformation is defined as follows:

- One or more new C1 connectors $F_1..F_m$ are added to the topology, such that there is a path $G_1..G_k$ ($k \geq 1$) from C to S with $\{E_1..E_n\} \cap \{G_1..G_k\} = \emptyset$ and $\{F_1..F_m\} \subseteq \{G_1..G_k\}$.

An example is depicted in Figure 3-9.

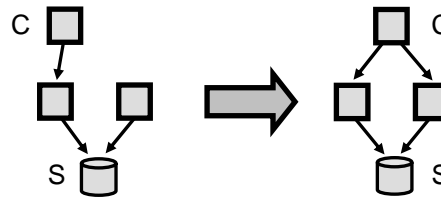


Figure 3-9. Example of the *meshing* pattern.

The alternative paths can be used for providing fault tolerance in case of process or network failures. Also, the *meshing* pattern is often used in conjunction with Pattern 2 and/or Pattern 4 in order to achieve horizontal load distribution.

3.4.4 Pattern 4: Data Distribution

Instead of storing data objects in a single data store, data objects can be distributed among multiple data stores [EN00] [ÖV99].

Given

- a non-empty set of processes $\{C_1..C_n\}$ ("client processes"),
- a transactional data store D_1 ,
- a set of C1 connectors $\{E_1..E_n\}$, where each E_i ($1 \leq i \leq n$) is a connector from C_i to D_1 ,

The pattern's transformation is defined as follows:

- A new transactional data store D_2 is added to the topology.
- For each E_i ($1 \leq i \leq n$), either (1) a new C1 connector from C_i to D_2 is added or (2) E_i is replaced with a C1 connector from C_i to D_2 or (3) nothing changes. At least one C1 connector has to be added or replaced. It is not permitted to replace all E_i .

An example transformation for the data distribution pattern is given in Figure 3-10.

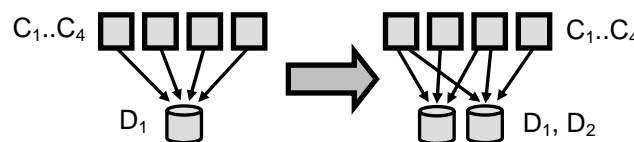


Figure 3-10. Example of the *data distribution* pattern.

The transformation above covers the case that exactly one data store is added to the topology. Transformations for an arbitrary number of new data stores $D_2..D_m$ ($m \geq 2$) can be defined accordingly.

In addition to the topology change, data objects previously stored in D_1 (before the transformation was performed) have to be distributed among $D_1..D_m$ according to some distribution scheme. All future read and write accesses to data previously stored in D_1 have to take that distribution scheme into account, i.e., have to be directed to the appropriate data stores. We distinguish three variants of the *data distribution* pattern. All three variants facilitate horizontal load distribution because data access operations can be distributed among the data stores.

- (a) **Replication.** Data objects are replicated among *all* data stores, i.e., each data object is stored in all $D_1..D_m$. Replication improves availability in case of data store failures, especially for read accesses. Depending on the underlying replication approach, availability for write accesses can be improved or reduced. On the downside, write accesses may lead to more load than in a non-replicated scenario because all replicas have to be kept consistent.
- (b) **Distribution based on a static partitioning criterion.** According to some partitioning function, the set of data objects is split into m disjoint subsets $Part_1..Part_m$. Each subset $Part_i$ ($1 \leq i \leq m$) is stored in data store D_i . For example, a partitioning function could specify that all *Customer* instances are mapped to $Part_1$, all *Order* instances with $date < 1/1/2001$ are mapped to $Part_2$, and all other instances are mapped to $Part_3$. Another example is a partitioning function that clusters *Order* instances together with their corresponding *Customer* instances. If one of the data stores fails, only the corresponding partition of data objects is affected, whereas the other partitions remain available for read and write access.
- (c) **Ad hoc distribution.** Data objects are distributed among all data stores as described in variant (b); the difference is that the partitioning function does not (only) depend on type and/or attribute values. The decision in which data store a particular data object is stored may, for example, depend on a random function, a round robin approach, or on the current load and fill ratio of available data stores. While ad hoc distribution can greatly improve throughput and availability for insert operations and can help to avoid data skew, it also may require much more lookup operations as no static criterion can be used to determine the location of objects. An example of ad hoc distribution (at the disk level) is round-robin fragmentation employed in the Informix database [IBM03a].

It is also possible to combine the three variants (see Subsection 5.2).

3.4.5 Pattern 5: Facade Insertion

This pattern inserts a facade process between a set of client processes and their servers. All communication between clients and servers has to go through the facade process.

Given

- a non-empty set of processes $\{C_1..C_n\}$ (“client processes”),
- a set of server components $\{S_1..S_m\}$, and
- a set of C1 connectors $\{E_1..E_k\}$, where each E_i ($1 \leq i \leq k$) connects a client process from $\{C_1..C_n\}$ with a server component from $\{S_1..S_m\}$. For each C_i ($1 \leq i \leq n$), there must be at least one E_j ($1 \leq j \leq k$) with C_i as client. For each S_i ($1 \leq i \leq m$), there must be at least one E_j ($1 \leq j \leq k$) with S_i as server.

The transformation for the facade insertion pattern is defined as follows:

- A new process F is added to the topology.
- All C1 connectors $\{E_1..E_k\}$ are removed.
- For each C_i ($1 \leq i \leq n$), a new C1 connector from C_i to F is added.
- For each S_i ($1 \leq i \leq m$), a new C1 connector from F to S_i is added.

In Figure 3-11, two example transformations are depicted. As in the other examples, the first example shows only the selected subset of the process topology that is being transformed. The second example depicts another valid transformation; it also shows several components and connectors of the architectural configuration that are not selected, i.e., do not take part in the transformation.

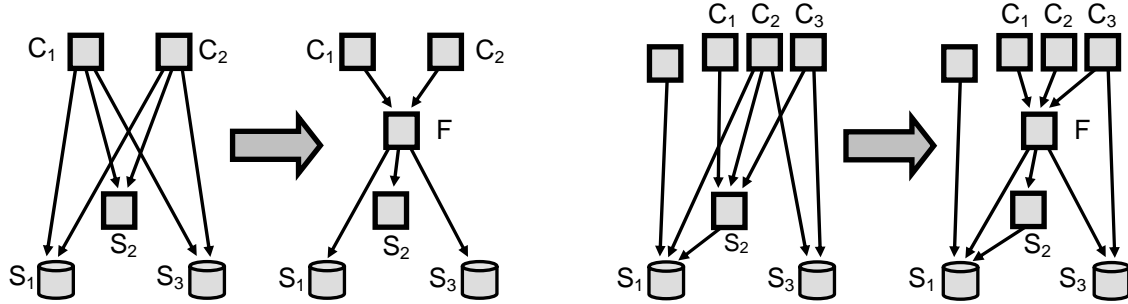


Figure 3-11. Two examples of the *facade insertion* pattern.

The new process F provides a single point of entry for clients, thereby simplifying their connectivity. In addition, the new process can be used to enforce certain access policies for all clients. Preventing untrusted clients from directly accessing central servers can also improve security: After introducing a facade process, the servers are no longer visible to a (potentially malicious) client. The pattern is similar to the facade design pattern in [GHJV95], but focuses on processes rather than interfaces.

3.4.6 Pattern 6: Integration of Subsystems

In many cases enterprise applications and their process topologies are not built from scratch but on top of existing subsystems. The *integration of subsystems* pattern helps to integrate two previously isolated subsystems. The process topologies of both subsystems are merged into a single process topology by adding components/connectors such that the resulting topology is connected.

Given the set of all servers components $\{A_1..A_n\}$ of the first subsystem's process topology and the set of all servers components $\{B_1..B_m\}$ of the second subsystem's process topology. As illustrated by the examples in Figure 3-12, we distinguish two variants of this pattern:

- Integration by introducing new client/server communication relationships.** The transformation for this variant is defined as follows: One or more new C1 connectors $E_1..E_k$ are added to the topology. Each of the new connectors connects either an A_i of type process with a B_j or a B_j of type process with an A_i ($1 \leq i \leq n$, $1 \leq j \leq m$). The resulting topology must not have any cycles.
- Integration by introducing a federation layer on top of existing subsystems.** The transformation for this variant is defined as follows: A set of new processes $\{C_1..C_p\}$ are added to the process topology. For each C_k ($1 \leq k \leq p$), at least one C1 connector from C_k to an A_i ($1 \leq i \leq n$) and at least one C1 connector from C_k to a B_j ($1 \leq j \leq m$) are added.

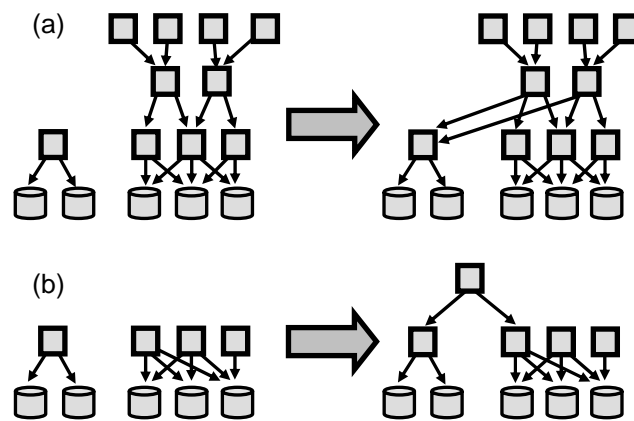


Figure 3-12. Examples of both variants of the *integration of subsystems* pattern.

3.5 Discussion

In this section, we discuss a few selected aspects and questions in more detail.

3.5.1 Enterprise Applications that Follow Other Styles

Our “multi-tiered enterprise application” architectural style covers a broad range of systems. However, it should be noted that not all enterprise applications follow our style. For instance, there are systems with a strong peer-to-peer aspect, which require cyclic instead of acyclic process topologies. Such enterprise applications should not be modeled with our style. As an alternative, other styles or variants of our style can be used. In some situations it could also make sense to use our style to describe only a specific *subsystem* of a more complex enterprise application. In such cases, the enterprise application as a whole could follow another architectural style while the subsystem follows the “multi-tiered enterprise application” style.

3.5.2 Modeling Tiers as Components

We decided to treat tiers as components in our architectural style because our style uses elements of the “layered subsystems” style, where layers are treated as components, too. An alternative would be to treat only processes and transactional data stores as component types. A tiered structure could then be viewed as a set of additional constraints on processes, transactional data stores, and C1 connectors. While, in general, that is a feasible approach, we think that treating tiers as components (i.e., as computational entities) is much more intuitive from the viewpoint of software developers.

3.5.3 Architecture Development Process

In Section 3.3, we explained that, in general, there are no precise rules or an algorithm a software architect can follow to design complete, suitable process topologies. However, it is at least possible to structure the process of finding, evaluating, and comparing different design options. For instance, [KBAW94], [KABC96], [BCK98], [KKB+98], and [KKB+99] do this for architectural styles/configurations in general.

Our thesis primarily focuses on middleware and data management aspects and not on software and architecture development processes. We assume that there is a software architect who has sufficient knowledge about the enterprise application, business domain, requirements, and all major factors that influence design and quality of process topologies. We further assume that he uses an appropriate

decision making process for making architectural decisions, e.g., designing and changing process topologies. Given that an architect makes appropriate decisions (ideally, using our process topology patterns), we are particularly interested in how underlying *enterprise application middleware* should support the systems he wants to build.

3.5.4 Two-Level Topology Structure

The tier topology and the process topology of our architectural style each address different levels of granularity. It could be argued that an architectural style should focus on a top-level aspect, i.e., should address tier topologies only. Then each tier would represent a subsystem whose internal structure can be described by means of patterns and possibly other architectural styles. Process topologies would play a role only at the level of subsystems. However, we think that particularly this two-level meta structure is a main characteristic of multi-tiered enterprise applications: On the first level, there is the tier topology, and on the second level, there is the process topology, which details (and is constrained by) the tier topology. Defining a style for the tier topology alone would also suggest that tiers could be treated as black box subsystems. But this is not the case: For software architects, it is important to know which servers of a given tier communicate with which servers of the underlying tier. Parts of a process topology, e.g., a mesh (see the meshing pattern described in Subsection 3.4.3) may involve more than just one tier. For this reason, it is not advisable to view isolated subsets of a process topology only.

3.5.5 Patterns Based on Graph Transformations

We decided to regard and present our process topology patterns as software patterns. In general, software patterns address software developers. They encapsulate specific design knowledge and are designed to be of maximum use for software developers that have to solve practical design problems. They comprise problem, context, and solution parts. We present our process topology patterns as high-level design patterns, which are specific software patterns. In contrast to existing design patterns, descriptions of our process topology patterns also specify transformation rules. This helps to precisely describe the effects of their application on process topologies and, in addition, emphasizes the evolutionary aspect.

Alternatively, it would also be possible to base process topology patterns on the concept of graph transformations rather than on software patterns¹. More details on graph transformations can be found, for example, in [Ehri79], [PER95], and [Roze97]. In the following, we outline what changes are necessary to change our developer-oriented approach into a more formal, graph-transformation-oriented approach:

- (1) Our process topology patterns should be defined *as* transformation rules for process topologies instead of using transformations merely as part of the patterns' descriptions.
- (2) The process topology patterns, as presented in this chapter, are not atomic and too coarse-grained from the graph-transformation-oriented perspective. For example, in some cases an application of *the process replication* pattern implicitly includes an application of the *meshing* pattern. Instead, a small set of simple patterns based on atomic transformations could first be defined (e.g., for inserting or removing a single component or C1 connector). Then these atomic patterns could be combined (recursively) into more complex and coarse-grained patterns. Furthermore, patterns should be described as unique transformation rules (wherever possible), i.e., a specific transformation should correspond to a function that depends on a selected subset of the process topology only.
- (3) The patterns presented in this chapter describe transformations. Accordingly, these patterns can be named *topology transformation patterns*. Topology transformation patterns are a specialized class of process topology patterns. In addition to topology transformation patterns, we could introduce another specialized class of process topology patterns, so called *topology structure*

¹ R.-D. Kutsche, personal communication, January 9, 2003.

patterns. Topology structure patterns represent (static) subsets of process topologies. Examples are *single-client-multiple-servers* or *multiple-clients-single-server*. Topology structure patterns can be used not only to describe (parts of) process topologies; they also appear on the left side of transformation rules.

- (4) At the end of Subsection 3.1.3, we discussed similarities and differences between architectural styles and architectural patterns. We adopted the notion of architectural styles to describe the top-level architecture of a system. However, when we redefine our “multi-tiered enterprise application” style as an architectural pattern, we end up with a system of three types of patterns: First, there is the “multi-tiered enterprise application” *architectural pattern* which addresses the highest level of abstraction. Second, there are *topology structure patterns*, which describe (parts of) process topologies of enterprise applications. Third, there are *topology transformation patterns*, which (according to given rules) select topology structure patterns within process topologies and apply transformations to them.

3.5.6 Homogeneous Clusters of Data Stores

Many data store products can be clustered, which means that multiple data store instances are combined into a group. Members of a cluster collaborate, e.g., to synchronize data, balance load, or provide fail-over. Usually, clusters are tightly coupled, require a homogeneous environment, and rely on direct data store to data store communication. Many clusters attempt to keep distribution details transparent and to appear as a single data store to application processes. For example, with Oracle’s Real Application Clusters (RAC) [Orac02], each client can communicate with any one node of an Oracle database cluster; the cluster appears as a single database. Nodes are homogeneous and work on a shared disk subsystem. Each node requires direct network access to each other node and to each shared disk in the cluster.

In our “multi-tiered enterprise application” architectural style, we do not consider data store to data store communication. Such communication is not prohibited, but we provide no means to model it explicitly. When a cluster of data stores is employed, a software architect may decide to model the cluster either as a single data store in the process topology or as multiple data stores.

3.5.7 Extensions to the Graphical Notation

We expect that, in many cases, the notation we proposed in Table 3 is sufficient for graphically representing the top-level distributed architecture of multi-tiered enterprise applications. Nevertheless, there might be situations where software architects want to extend our notation in order to add details that are of special interest in the context of their particular projects. For example, details on communication protocols, machine boundaries, organizational boundaries, geographical distribution, hardware being used, network segments, or network bandwidth might be added. We encourage extensions to our graphical notation. However, we do not specify particular graphical elements (shapes, color, textual annotations etc.) for that purpose.

3.6 Summary

In this chapter, we first gave an introduction to architectural styles and patterns. Then we defined an architectural style specific to multi-tiered enterprise applications. Important elements of the style are the *tier topology* and the *process topology* of an application. On the highest level of abstraction, the tier topology describes the tiered structure of an enterprise application. The process topology describes an application as a graph of transactional data stores¹, processes, and client/server communication relationships. It details – and is constrained by – the tier topology. We discussed the design of process

¹ Note that process topologies could also be useful to describe enterprise applications that employ non-transactional data stores. However, as pointed out in Chapter 2, we focus on transactional data stores in this thesis.

topologies and presented a language of process topology patterns. Process topology patterns are high-level design patterns that help software architects to design adequate process topologies for their enterprise applications. To emphasize the evolutionary aspect of enterprise applications and their process topologies, we included rules for transforming process topologies in the descriptions of our patterns.

This chapter serves as a basis for the following chapters because it precisely describes the class of systems we focus on, i.e., multi-tiered enterprise applications. The chapter introduces the concepts of process topologies and process topology patterns, which are prerequisites for Chapter 4. With our architectural style, this chapter also provides an architectural model for multi-tiered enterprise applications. The architectural model is a basis for all following chapters and for Chapter 5 especially, since it proposes concepts for a middleware framework for multi-tiered enterprise applications.