Understanding and analyzing software architecture (of distributed systems) using patterns
Stal, Michael

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2007

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 30-12-2018
Chapter 2 - Patterns and Middleware

2 Patterns and Middleware

“Intellectuals solve problems; geniuses prevent them.” —Albert Einstein.

2.1 Patterns

2.1.1 Patterns and Software Quality

The quality of any software system depends directly on its underlying software architecture. Consequently, software architecture has a direct impact on success or failure of a software development project. In this context, the question arises what characterizes good software architectures, and likewise, what differentiates outstanding from moderate software architects? Because of the complexity of today's systems even the most experienced software architects are not capable of defining more than a coarse architectural baseline at the beginning of a project. In each development stage specific architectural decisions need to be taken. These decisions should be based on stable foundations, in particular the technical context, the predefined architectural baseline, the concrete application domain, and available experiences. In software engineering like in other disciplines the journey often is the reward. The aforementioned level of experience helps to differentiate software architects. An expert is characterized by the fact that he possesses a mental treasure chest of experiences, which enable him to come up with the right decisions. Transferred to the domain software architecture this means that experts usually select solutions, which already worked in similar contexts, instead of constantly reinventing the wheel as inexperienced developers would do. An experienced software architect applies the same solution for similar problems, even if there are multiple solu-
tions available in order to create uniform, orthogonal, easily readable and maintainable software architectures. The combination of single solutions that tactically address local problems should support the strategic goals of the architecture baseline. Examples for such strategic goals include non-functional qualities such as scalability or flexibility.

In software systems inherent complexity often is a direct consequence of emergent behavior. Emergent behavior denotes the principle that the combination of simple elements to an overall system may result in complex behavior instead of following a reductionism approach.

Which conclusions can be drawn from these considerations?

- A software architect requires knowledge how to solve frequently recurring problems that typically appear in certain contexts. Experts know an extensive set of such solutions, which they have acquired by many years of experience. Therefore it is essential that this kind of experience does not remain locked in the heads of software architects with a high risk of loss. Instead, these experiences should be made available explicitly, uniformly, and should be documented for other developers. That exactly is the purpose of software patterns.

- A software pattern defines a proven solution for a recurring problem in a specific context. Problem comprises all forces (i.e., constraints, requirements, desired properties) the solution should offer. The Solution describes a canonical design approach\(^5\) that resolves these forces. Context denotes a recurring set of situations for which the pattern provides a suitable solution.

- Patterns do not claim to describe the best solution that resolves the forces, nor would it usually be possible to prove formally that a given pattern provides an op-

\(^5\) I.e., a spatial configuration of participants including their relationships.
timal solution. This is a direct consequence of the fact that patterns represent architectural blueprints with a potentially infinite number of possible implementations and variations. Hence, it is impossible to formalize patterns in general. Nor would it make sense in most cases to implement a library of predefined pattern implementations.

- Patterns are no islands. They represent fundamental components of application architecture, which are often tightly connected to other patterns and architectural entities. Only the systematic and combined use of micro-architectural solutions such as patterns leads to a coherent and consistent software architecture. This does not imply by any means that high-quality software architectures do necessarily exhibit a high pattern density. Nor does it imply that software architectures with high pattern density automatically offer high quality. Rather it implies that all components of a concrete software design should stay in harmony with each other. Various categories of patterns exist to support architectural design. These categories address different levels of granularity, as well as different phases and activities of the development process.

- Software projects fail only seldom because they can not meet functional requirements, given that all functional requirements are specified consistently and completely. A substantially more complex and more risky issue is the challenge of addressing non-functional qualities. The reason for this problem is obvious, since development paradigms such as object-oriented programming and component-based software development focus primarily on functional aspects for design and implementation. Non-functional properties often tend to spread across the entire system, thus acting as cross cutting concerns. Therefore, patterns must consider non-functional aspects in two ways, on one hand by special pattern systems for the systematic realization of such requirements and on the other hand by consideration of these aspects in the patterns themselves.
Individual patterns as well as patterns applied in combination help address a lot of problems, which software architects face in their daily work. Eventually, experienced software architects own a broad set of patterns they can apply for solving architectural problem efficiently and effectively.

### 2.1.2 Documentation of Patterns

The fundamental advantage of patterns consists of the fact that they offer solutions for recurring problems. Moreover, they provide a basis to a software architect for structuring the problem and solution space. Abstraction introduced by patterns exceeds the granularity of individual classes and objects, thus offering a more appropriate abstraction level for architects. As already pointed out, patterns concentrate not only on the supply of functionality but also on non-functional aspects. The problem description which includes the forces as well as a "Consequences" part provides valuable information to the software architect. In order for patterns to be easily usable, they must be presented in documented form. Effective usage, however, requires that all pattern descriptions use a uniform documentation template. Concrete examples of a possible form for pattern description are introduced in the remainder of this thesis. Note, however, that different description forms are available in the literature. Despite of all differences, the currently available forms strongly resemble each other. In accordance with [14] the pattern description should reveal the following parts:

<table>
<thead>
<tr>
<th>PART</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Pattern name and a short description in few lines which essentially outline the application context and the rough solution.</td>
</tr>
<tr>
<td>Example</td>
<td>A concrete example which illustrates the problem. The succeeding parts of the pattern description typically refer to the running example.</td>
</tr>
<tr>
<td>Context</td>
<td>Exact context in which the pattern should be applied.</td>
</tr>
<tr>
<td>Problem</td>
<td>The description of the problem definition including the es-</td>
</tr>
</tbody>
</table>
Chapter 2 - Patterns and Middleware

<table>
<thead>
<tr>
<th>Solution Strategy</th>
<th>Coarse outline and overview of the solution strategy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Structure</td>
<td>The static structure of the solution, i.e., the necessary components and their relations to each other. Here CRC Cards are useful as well as UML object and class diagrams.</td>
</tr>
<tr>
<td>Dynamics</td>
<td>Typical scenarios, usually visualized using UML sequence diagrams.</td>
</tr>
<tr>
<td>Implementation (Micro Process)</td>
<td>A suggestion for necessary implementation steps.</td>
</tr>
<tr>
<td>Variants</td>
<td>Possible pattern variants.</td>
</tr>
<tr>
<td>Known Uses</td>
<td>Known uses in which the pattern was used in existing systems.</td>
</tr>
<tr>
<td>Example Resolved</td>
<td>Further refinement of the running example.</td>
</tr>
<tr>
<td>Resulting Context (Consequences)</td>
<td>Consequences, i.e. benefits and liabilities, resulting from the application of the pattern.</td>
</tr>
<tr>
<td>Related Patterns (See Also)</td>
<td>Related, similar, or complementary patterns.</td>
</tr>
<tr>
<td>Testability</td>
<td>In [58] the authors added a further part to guarantee the quality of the implementation: How to test an implementation of the pattern.</td>
</tr>
</tbody>
</table>

Figure 1: Document Template for pattern descriptions.

The documentation of patterns by uniform description templates\(^6\) permits organizing and indexing patterns in pattern catalogs as well as systematic search for patterns. Thus, patterns enable the explicit documentation of expert knowledge. Even less experienced

\(^6\) An underestimated benefit of pattern description templates like the one used throughout this thesis is that they also offer effective instruments to structure all kinds of architectural solutions. Thus, pattern description templates are valuable even outside of their pattern context, as they provide a structure to architectural solutions.
developers and architects get the opportunity to leverage expertise of other experts, no matter whether they obtain this knowledge from colleagues or from pattern literature. Pattern descriptions can be very lengthy with a lot of details. Patlets represent short pattern descriptions, which contain only a subset of the full pattern description.

A further advantage consists of the fact that patterns introduce a vocabulary for communication between software engineers. For example: If an architect refers to the Observer pattern, the majority of software engineers will already know the pattern and its underlying abstraction, without requiring lengthy or detailed explanations.

2.1.3 Patterns and Granularity

2.1.3.1 Rationale

Software architectures do not form flat structures, but hierarchies of different levels of abstraction and granularity. The overall software system is based on a core architecture design or reference architecture, which can be subdivided into layers and subsystems. The implementation of these architectural artifacts takes place via abstractions such as components, objects and classes, which exhibit tight connections to implementation technologies such as programming languages, operating systems or middleware. Since software architects and developers must transfer each of these abstraction steps finally into executable implementations, they also face different abstraction and granularity levels of design problems. Hence, there are also different categories of patterns for these different abstractions and granularity levels.

2.1.3.2 Architecture Patterns

Architecture patterns help to enforce the strategic design decisions for a software system under development. They have an impact on the core design center of the software architecture. Examples include the Broker pattern [14] that serves as a base for remoting middleware platforms, the Model-View-Controller pattern [14] used to implement graphi-
cal user interfaces with explicit separation of user interaction, presentation and business logic. The Pipes & Filter pattern is suitable for implementing modular and flexible pipeline processing (e.g., compilers and command processors). The Layers pattern [14] helps to the separation of functionality in different horizontal or vertical layers (tiers). All of these architecture patterns affect the whole software system, for example the structuring into subsystems as well as their relations. However, this doesn't mean that only one single architecture pattern can be implied in a given context. For instance, when building an IDE, architects could apply the Model-View-Controller pattern for user interaction and the Pipes & Filters pattern for combining different processing steps.

2.1.3.3 Design Patterns

The seminal book "Design Patterns: Element of Reusable Object Oriented software" [33] introduces patterns that focus on design problems at medium levels of granularity: Design patterns play an important role for designing subsystems. Like architecture patterns they possess the property to be independent of concrete application domains. That is a direct consequence from the fact that design patterns mainly deal with the structuring of the implementation, not with the implementation itself. The Proxy pattern [33] [14] is a typical example for design patterns. Proxies represent surrogate objects, which act as transparency layer between a real object and a client. The Observer Pattern [33] is applied in almost all software systems. It deals with the decoupling of event consumers from event producers.

2.1.3.4 Idioms

While architecture and design patterns do not depend on concrete application domains, as they affect the design of the software architecture and its subsystems, the developer also faces recurring problems on the implementation level. These problems result from the technologies used. Different paradigms and different characteristics of programming languages such as C++ or Java result in different implementation strategies. For such implementation specific design problems, idioms help developers to use the program-
ming language or infrastructure effectively. An example is the Explicit Termination Method (ETM) applicable to programming languages with garbage collection like C# or Java. ETM introduces a termination method \(\text{Dispose()}, \text{Close()}\) to a resource class that the client must explicitly invoke to free resources held by class instances.

### 2.1.3.5 Using Patterns for Strategic and Tactical Design

The categorization of patterns into architecture and design patterns can be sometimes challenging. For example, architecture patterns like Layers [14] often have an impact on the global software architecture, but may also be applied to local subsystem design. Hence, I propose an additional classification that depends on the usage of the pattern in the software architecture:

- **Strategic pattern application** means: a software pattern is applied to implement architectural strategies that have an impact on the design center of the software architecture. While architecture patterns are particularly suitable for strategic pattern application, in a given problem context design patterns might also be a reasonable choice for strategic design.

- **Tactical pattern application** means: a software pattern is applied to define tactical architecture decisions. Although design patterns were introduced for that purpose, architecture patterns might also be applicable for tactical design.

The software architecture must be constructed in such a way that strategic design decisions always dominate tactical design decisions. Consequently, strategic design happens before tactical design. In other words, strategic pattern application should shield the strategic architecture core from being tampered by tactical pattern application.
2.1.4 Proto Patterns

For many domains no pattern languages exist, because they include areas that can not be fully covered by patterns. Hence, when software engineers are building new applications within those domains, it is very likely that at least a few existing software patterns are applicable. In addition, some white spaces exist for which no patterns are currently available. There might be several reasons for this:

- There are appropriate software patterns not yet known to the software engineers.

- There are some potential pattern candidates but no pattern mining activities (see 2.1.13) have been applied so far.

- The problems in the white spaces can be easily covered by conventional design techniques and common architectural means. Thus, there are multiple solutions for these problems but none of these solutions applies for becoming a pattern of its own.

Potential pattern candidates are often called Proto Patterns or Blueprints. They should be documented like patterns using short description templates (patlets). Proto patterns may become full software patterns if they meet all criteria such as the existence of at least three known (and independent) uses.

2.1.5 Pattern Organization and Classification

In addition to the organization of patterns into pattern categories different criteria exist for pattern classification.

Patterns in the Gang-of-Four book [33] are classified by their purpose and whether they are applicable on the object or class level:
• Class patterns deal with the static relations between classes and subclasses, whereas object patterns address the static or dynamic relations between objects.

• The purpose of a pattern addresses the question whether a pattern mainly deals with creating objects (creational patterns), the composition of objects and classes (structural patterns) or the interaction of classes and objects (behavioral patterns).

The POSA 1 book [14] introduces problem categories for the organization of patterns. The fundamental idea consists of using the problem addressed by the pattern as classification criteria. For example, patterns that deal with the construction of distributed systems could be placed into the problem category "Distributed Systems". The classification of problems using problem categories is however not always straightforward, so that the same patterns might end in different problem categories.

Often the classification schema suggested by [33] is too generic for organizing large quantities of patterns, while the organization by problem categories might result in many pattern categories.

As a consequence, it proves to be useful to index patterns using different ontologies and classification schemas instead of relying on a single scheme.

2.1.6 Pattern Catalogs, Systems, and Languages

Often different patterns solve different sub-problems within the same application domain. Hence, one idea is to combine patterns from the same domain to libraries in order to make these libraries available to developers as a repository of re-usable design artifacts. Pattern libraries and/or pattern books such as [33] contain loose catalogs of fundamental patterns for general design problems. In these catalogs patterns are available as independent entities, which do mostly not exhibit dependence to other patterns. In contrast patterns systems such as [14] and [85] contain patterns that frequently refer to each other. For example, the Model-View-Controller architecture pattern [14] refers to the Ob-
server pattern in its implementation section in order to address the problem of event notifications. Pattern systems are not limited to provide a loose catalog but need also to define relationships between patterns as well as their complementary application. To that extent pattern systems are comparable with programming languages, whereby patterns take over the role of the language vocabulary and rules for pattern implementation and combination take over the role of the grammar. Pattern systems do not provide complete coverage of a domain. If complete coverage is provided, then the pattern system is a pattern language. In practice, full coverage of domains proves as an almost unattainable goal due the inherent complexity of most domains. For this reason so far only few pattern languages exist, and the existing pattern languages cover very limited application domains.

A pattern encyclopedia contains a complete set of short descriptions of all patterns known. The patterns contained are typically organized using various ontologies. For each pattern there is also a reference to its full documentation. To that extent such an encyclopedia can be regarded as a kind search engine for patterns. One of the few examples represents [78]. Disadvantage of encyclopedias is the continuous modification of existing patterns and addition of new patterns. That is the reason why a Web-based, constantly updated pattern encyclopedia would be beneficial.

2.1.7 Best Practice Patterns

Software platforms such as Java EE and .NET offer the possibility of realizing the most diverse types of applications on a homogeneous and transparent basis independently of details of the underlying system infrastructure. In particular, Web-based applications and distributed enterprise applications often use one of the aforementioned platforms as runtime environment. Despite of the homogeneity and orthogonality of the programming interfaces provided by these and other platforms, their extent and complexity remain substantial hurdles. In order to master these hurdles and build efficient applications effectively, best practice patterns have evolved. Best practice patterns are architecture and design patterns which give up independence from system details and address a concrete
platform instead. To that extent best practice patterns are comparable with programming language-specific idioms. Best practice patterns are typically composed to complete pattern systems. Prominent examples include [3] for the software development under Java EE as well as [56] for .NET application development. These books are focusing mainly on the development of Web applications. They address problems like navigation between web pages when using JSP and/or ASP.NET, the efficient transmission of data between browsers and Web servers, or covering system details and minimizing network load by introduction of session facades. It is not unusual that best practice pattern systems contain general architecture and design patterns which have been adapted for the concrete platform.

2.1.8 Patterns and Architecture Qualities

Pattern literature considers patterns as vehicles to achieve architecture qualities. I’d like to explain briefly why patterns actually help to support architecture qualities. For more detailed discussions on architecture qualities, I recommend [5].

- **Conceptual integrity** and **Orthogonality**: If patterns are used to solve recurring problems for building the software architecture, then conceptual integrity and orthogonality will be improved if patterns are able to cover many parts of the software architecture and if the same patterns are always applied for the same problems.

- **Expressiveness**: Another quality aspect is whether the software architecture is easily understandable. As an experiment for checking this quality property, one of the architects could call a colleague who is not involved in the architecture design and try to explain the software architecture to her or him. If this is not possi-

---

7 A high pattern density implies that architects use well-proven solutions instead of reinventing the wheel. However, as already pointed out in this chapter, a high pattern density does not automatically represent a proof for high architecture quality. If for instance, different patterns were applied for solving similar problems, this leads to a decrease of conceptual integrity.
ble within 5 minutes, then the software architecture might be not expressive. As patterns help to cover the software architecture providing a kind of idiomatic language, it is obvious that pattern help improve expressiveness.

• **Simplicity:** Software architects should abide to the KISS principle (Keep it Simple, Stupid) in order to achieve simplicity. Thus, it is necessary to “use the simplest solution that could possibly work” (Kent Beck). Smart design that no one understands and which solves possible future requirements adds accidental complexity and, thus, potential sources for errors. As Brian Foote noted, “patterns are an aggressive disregard of originality”. By using patterns proven architectural solutions become part of the software architecture under development. Hence, the software architecture will be simpler to understand, especially when the applied patterns are already familiar. For checking this architecture quality, the same telephone check could be helpful like the one for expressiveness. It is important to mention that simplicity and expressiveness are not two sides of the same coin. A simple architecture is not necessarily expressive.

• **Correctness:** As patterns represent well-proven solutions that have already been successfully applied in other applications, they define correct design solutions. The more “tested” components and patterns architects integrate in a system, the less correctness problems they will face. It is necessary to keep in mind in this context that developers could implement a pattern incorrectly. This, however, is unlikely as pattern descriptions provide guidance for their own implementation which always makes them preferable to proprietary home-grown solutions (“space where no man has gone before”).

• **Symmetry:** Structural and functional symmetry are important aspects for architectural quality. Structural integrity is similar to conceptual integrity. Functional integrity defines aspects such as “when there is an open method, there should also be a close method”. Since most patterns reveal functional symmetry, their application helps achieve architectural symmetry. As opposed to what many architects
think even patterns might violate symmetry. For example, the Abstract Factory pattern [33] introduces object creation but not object destruction. This is a liability as it is very likely, that if an object is difficult to create, then object destruction will also be complex. Note, however, that in some albeit rare cases asymmetry might be the better choice.

In addition to supporting the aforementioned architecture qualities, patterns also ease prevention of some quality problems. Two examples:

- **Violation of Layering**: Violation of layers among other liabilities introduces unnecessary dependencies in the software architecture. When applying the Layers pattern [14] and strongly conforming to the implementation recommendations the pattern description specifies, there won't be violations of the layering.

- **Dependency Cycles**: Cycles in the dependency tree cause various liabilities. Some roles of the subsystems participating in the cycle are only vaguely defined. Modifying or changing any of these subsystems is difficult, thus reducing flexibility and testability. As the roles in patterns usually are clearly defined and there are no cycles in high-quality software patterns, usage of patterns reduces the chance that software engineers accidentally introduce dependency cycles.

### 2.1.9 Patterns and Abstraction

One significant advantage of patterns is that they allow higher levels of abstraction. A software system that consists of million lines of code (LOCs) is neither maintainable nor understandable in terms of these LOCs. This is caused by sheer complexity, as a software engineer would have to consider all LOCs as well as the relations between these LOCs to understand the software architecture. By introducing operations as higher level building blocks the number of entities is reduced by a factor of 10. Classes lead to another reduction by 10 as does the introduction of components and subsystems. While classes, components, and subsystems represent uniform artifacts with constrained types
of relationships, patterns introduce pattern-specific roles and pattern-specific relations. Hence, software patterns introduce additional abstractions to all existing abstraction layers and thus increase readability.

### 2.1.10 Patterns and other Methods

A pattern is not an isolated piece of architectural design unrelated to its environment. This statement applies also to the architectural concept of software patterns itself. Architects use a whole set of technologies during software development. But what is the relationship between patterns and other methods and architectural concepts? Patterns represent an ideal addition of other technologies as for example RUP (Rational Unified Process), eXtreme Programming, or UML (Unified Modeling Language). They do not make any of these technologies obsolete. The reason for it is obvious. Methods like RUP and modeling languages such as UML offer the necessary means for software engineering to systematically achieve the project goals, but they don't provide any assistance solving concrete problems of the application or solution domain. Rather they represent domain-neutral general-purpose tools. In contrast, patterns address solutions for concrete problems, so that their employment always serves as complementary addition for methods and technologies. During the software development process additional pattern categories are helpful:

- **Organizational patterns** regulate the composition of teams and communication of the team members.

- **Analysis patterns** are helpful during analysis, beginning with requirement analysis, over application scenarios up to the first design sketch. Martin Fowler dedicated a book to this topic [32].

- **Architecture and design patterns** offer solutions for design problems in different dimensions of abstraction and granularity.
- *Idioms* support the implementation activity.

- *Test patterns* give assistance for testing, beginning with unit tests up to system and integration tests.

- *System Infrastructure patterns* deal with proven solutions how to configure the system infrastructure.

For generative approaches as for example MDSD (Model-Driven Software Development) patterns provide a means to construct both the domain-specific, i.e. platform-independent, and the platform-specific architecture. Also the generators, which are responsible for the individual transformation steps from the software architecture to the implementation, can be realized using patterns. AOP (Aspect-Oriented Programming) and patterns show some similarities. On one hand patterns can facilitate the design of aspects. On the other hand patterns can be injected to applications by aspect weavers. Moreover, there are some problems, in which both AOP and patterns offer suitable solution alternatives. As standard example component containers should be mentioned, which need to dynamically provide services to components such as security or object persistence. This can be achieved in two ways: Either by a descriptive specification of component configuration and the application of the Interceptor patterns [85] or by injecting these services into an application using an aspect weaver.

To summarize, the application of patterns is not limited to architectural design, but can address all activities, where recurring problems in similar contexts are encountered. This comes to no surprise as patterns have their roots not in computer science, but originated from other disciplines such as building industry. Most influential for software patterns has been the work of the architect Christopher Alexander "The Timeless Way of Building" [1].
2.1.11 Pattern Relations

Software patterns do not exist isolated in time and space, or as John Vlissides once said "a pattern is no island". Instead, there are many possible interactions and relations between patterns.

- **Usage and/or refinement**: A pattern uses another pattern for its own implementation. In particular, architecture patterns often refer to design patterns. The result can be complete pattern systems with the architecture pattern defining the core of the underlying domain. An example is the usage of the Observer pattern [33] to implement the subscription and notification functionality within the MVC pattern [14].

- **Variants**: Often patterns specify a special configuration or a specific implementation of their participants. Moreover, a pattern is often applicable for different application areas, but the pattern description for the sake of brevity and readability focuses only on one application area. For other application domains deviations from this main variant are often necessary. These slightly modified patterns do not represent patterns of their own, but variants of the pattern they were derived from. An example of such a variant is the integration of a queue between event producers and consumers in the Observer pattern (event channel variant). However, it is not always easy to decide whether a pattern represents a variant of another pattern or a pattern of its own.

- **Alternative**: For the same problem different solutions and thus different patterns could exist. In this case, the developer decides which pattern is more suitable with respect to constraints and consequences.

- **Combination**: A given problem cannot always be solved by applying individual patterns. In many cases it turns out to be necessary to combine multiple patterns. Example: For providing a XML Web service applications different tasks need to be
solved. The Proxy pattern in its remote variant [14] introduces transparent communication to the client application and to the XML Web service.

These relations between patterns mostly define architectural composition techniques. Obviously, they are different from composing finer-grained artifacts such as classes or objects.

2.1.12 Instantiating Patterns

The correct and effective use of software patterns requires a systematic process. In principle, the application of a pattern can be regarded as an iterative micro development process which starts with requirement analysis and ends with an implementation.

- **Analysis**: First an exact problem description is necessary. Which problem in which problem domain is to be solved and which level of granularity needs to be addressed? Which functional and non-functional characteristics and constraints must be considered?

- **Search**: Do software engineers already know a pattern which addresses exactly the specified problem or is possible to find such a pattern in existing literature? If several patterns exist, the constraints and consequences resulting from the pattern’s usage should serve as decision basis. If several patterns can balance the forces, then the simplest solution should be preferred according to the KISS principle (Keep it Simple, Stupid). If, however, no appropriate patterns exist or no pattern can balance all the forces, then engineers should resort to normal development methods.

- **Implementation**: If the architect finds a suitable pattern to a given problem, the instructions in the implementation part of the patterns should be followed. The pattern needs to be carefully integrated into the existing architecture design. If the
applied pattern requires further patterns for its implementation, this procedure is to be repeated incrementally and recursively.

For the instantiation of patterns some additional issues must be considered.

Patterns define roles instead of prescribing concrete software artifacts such as classes or components. An individual class in a software design can therefore at the same time take different roles in different patterns. This can be made visible in a UML diagram by using stereotypes such as <<RoleName::PatternName>>. When adding a pattern to an existing software design, the developer must specify accordingly, whether already existing design components are to take over such roles, which artifacts must be added, and how the pattern embeds itself into the already existing design context. Since patterns specify only a blueprint for their own implementation, this kind of integration process requires a decision which pattern configuration provides an adequate solution of the design problem. If several patterns are to be applied during design, first the patterns with highest priority and highest abstraction level should be integrated using an incremental process.

2.1.13 Pattern Mining

2.1.13.1 Rationale

The preferential use of patterns is for design and implementation of software architectures. But where do all the patterns and pattern systems come from? The answer is pattern mining which describes the systematic reflection over existing applications on the search for proven solutions for recurring problems. In general, patterns are not invented but found. Particularly promising search objects are product families and platforms (including application frameworks and libraries), because they must be instantiable respectively usable in different application contexts. In addition, they are subject to a continuous evolution process. Hence, they must exhibit high architectural purity and clarity. That becomes apparent for example by the fact that in these software architectures all architecture decisions take place explicitly and are then also explicitly visible. The most well-
Chapter 2 - Patterns and Middleware

known literature to patterns originates therefore from software experts, who were involved in the development or use of application frameworks. Extracting patterns from existing systems is not trivial, because it is not sufficient to convert the implementation of a potential pattern candidate (see 2.1.4) simply into a generic pattern. Instead the complexity consists of extracting the key idea of the pattern as well as the basic constraints and consequences for its applicability. In this process, experts also need to examine whether the pattern prototype actually concerns a recurring solution and, likewise, which possible variations it offers. Since patterns are being applied to many applications in many contexts, they must reveal high quality. Nevertheless, patterns are not carved in stone, but are subject to continuous evolution. Actual application of patterns reveals additional scope and restrictions. New knowledge might lead to a restructuring, an extension or a further detailing of the pattern. Eventually, a pattern could even become obsolete, either because better solutions for the addressed problem have been detected or, albeit very rarely, because the pattern was incorrect. Pattern mining however not exclusively serves for discovering new patterns. Since in the meantime many software architects and developer use patterns for application development, applications can be understood better, if their software architectures and thus their underlying patterns are known.

2.1.13.2 A Pattern Mining Process

The challenge of pattern mining is how to apply it systematically. In many aspects it closely resembles the definition of re-usable software components. As already mentioned a Commonality/Variability analysis for a product line or platform is a good opportunity for pattern mining as patterns can be regarded as core design assets of a product line, for the same reason software components represent re-usable implementation assets. Platforms and program families also represent ideal places for finding new patterns. There is a convincing reason for this. Software families and platforms for a specific domain must address core problems that characterize the domain under consideration. All these problems represent recurring problems for all of the applications being developed within this domain. Thus, the solutions provided by software families and platforms for
those recurring problems are automatically potential pattern candidates. Hence, all patterns that describe a domain, help express the domain in a kind of idiomatic approach.

I have developed a coarse grained process (see Figure 2) that supports pattern mining which I used for the pattern descriptions I’ve contributed to [14][85]:

1. **Identification of recurring problems**: In the application or solution domain identify common recurring problems. For this purpose a Commonality/Variability analysis of existing applications (such as platforms, frameworks, product families, or libraries) is appropriate. Work flows and use cases are particularly helpful in this context. E.g., in remoting middleware one problem consists of transparently accessing a remote object from a client (see also 2.2.3.2).

2. **Checking for solutions**: For each problem identified in step 1 analyze which solutions have already been applied to address the problem. E.g., in .NET, CORBA, DCOM intermediate objects are representing the remote object in the address space of the client.

3. **Analyze solution commonalities**: Find if there are commonalities among all these solutions. If not continue with step 6. In the running example this turns out to be straightforward as the intermediate objects introduced in the previous steps must be indistinguishable for a client from the remote object and must hide network communication. This is achieved by forcing clients to always access the interface and not the remote object directly. In addition, the intermediate object must implement the same interface as the remote object. The methods in the intermediate object do not provide business logic but functionality for connection establishment, method forwarding, etc.

4. **Abstract to patterns**: Abstract the different solutions to a common pattern if an appropriate pattern is not already available in existing pattern literature. Handle possible deviations from the main patterns as variants. Note the found solution
could also be a variant of an already known pattern. *This abstraction directly leads to the Proxy pattern in its remote variant* [33][14].

5. **Analyze higher abstraction levels**: Analyze if the given problem is integral part of a higher level problem. If yes, also recursively iterate with step 2 for this higher level problem. *For example, the whole handling of remote communication, including registering and locating remote objects, as well as transparently integrating remote objects lead to the problem that can be solved by the Broker architecture pattern.*

6. **Divide-et-conquer**: Try to partition the problem into sub-problems and for each sub-problem, continue with step 2. *In the Proxy example, sub-problems include connection establishment and connection handling, marshalling and de-marshalling.*

7. **Check if it really is a pattern**: Analyze whether the pattern candidate identified in the previous steps is a solution with at least three known uses. If no, treat it as proto pattern (also known as blueprint: see 2.1.4). *The Proxy pattern (remote variant) is applied in more than three known uses (CORBA, DCOM, RMI, RPCs, XML Web services).*

8. **Documentation**: Document the pattern following a canonical pattern description form. *For the Proxy pattern this was done in [33][14].*

9. **Relationship analysis**: Test for relationships between identified patterns so that possibly a pattern system or pattern language can be identified. *E.g., the Broker pattern uses the Proxy pattern for its implementation. This is a using-relationship.*

10. **Quality Assurance**: Publish the pattern and use writer’s workshops to obtain further feedback. *This was done for Proxy before it appeared in [33].*
11. **Continuous evolution**: When further applications use the patterns, re-check continuously whether the pattern documentation is still appropriate. Change the pattern, refactor it, extend it or even remove it if necessary. This lead to the refinement of the Proxy variants in [14].

![Pattern Mining Process](image.png)

**Figure 2**: A Pattern Mining Process.
2.1.13.3 Pattern Mining and Model-Driven Software Development

The pattern mining process applied to program families and platforms within a domain returns an idiomatic language that expresses this domain respectively its potential applications in terms of patterns and proto patterns. Note that the result does not define a domain specific language. Software patterns in general are entities of software architecture respectively implementation (i.e., the solution domain), but not of an application domain (i.e., the problem domain). However, the system of patterns provides insight into the inner structures of the application domain. Another possible usage of the patterns retrieved is Model-Driven Software Development. If a system of patterns covers a complete or subset of a domain or at least of some sub-domains, then the patterns extracted are valuable components that participate in the software architecture of the target application. Hence, a generator could use these patterns to control the generation of the target application.

2.1.13.4 Pattern Mining and Software Architecture Understanding

Pattern mining is not only applicable for extracting new software patterns from platforms or program families that cover a particular domain, but also for analyzing whether these platforms or program families already contain well-known patterns. In this context, the pattern mining process represents a valuable tool for reverse engineering activities. In order to understand a given domain such as Remoting Middleware, it is not essential that the application or domain can be fully covered by a pattern language. Nor is it mandatory to understand all patterns used. Instead, it is sufficient to understand the strategic core aspects of the underlying software architecture using patterns. These core aspects are expressible by a system of pattern, proto patterns and additional architectural entities, all of them interwoven in different layers of abstraction.
2.1.14 Derivation of Best Practice Patterns

When patterns respectively pattern systems idiomatically express the core concepts of a domain, software engineers are able to derive best practice patterns. This is possible by analyzing the consequences and constraints of the patterns applied as well as the characteristics of the underlying system infrastructure.

Example: using the Broker pattern to implement a XML Web services toolkit on top of HTTP leads to the liability of communication overhead due to expensive XML-based marshalling and HTTP performance penalties. Thus, remote objects should export methods with coarse-grained result and response data types. In addition, gateway objects may be introduced on server nodes that initiate workflows involving different local objects instead of allowing clients to access these objects remotely. These are examples for two best practice patterns.

2.1.15 Reengineering and Refactoring using Patterns

If the software architecture of an existing software system reveals some weaknesses or must be opened for change, software engineers might want to perform reengineering or refactoring activities. In the majority of existing literature reengineering denotes the complete modification of the software architecture as a whole. As opposed to reengineering refactoring describes local improvements: the externally visible behavior of a local part of the software architecture remains the same while its internal structure is subject to change. Hence, reengineering has a strategic impact on the software architecture and refactoring has a tactical impact. Refactoring in this context (see [31]) consists of small modifications of a given structure such as moving methods from subclasses to superclasses or eliminating unnecessary dependencies on member variables. In his book “Refactoring to Patterns” Joshua Kerievsky introduces more advanced refactorings that modify implementations so that patterns become applicable (see [46]). These refactorings help increase pattern density in a given software system and thus improving quality
by transforming proprietary solutions into well-proven solutions. Kerievsky primarily uses the design patterns from [33] for this purpose.

This approach can be taken one step further leveraging software patterns in a more holistic approach. Instead of constraining refactoring to design patterns, refactorings might be defined for all pattern abstraction and granularity levels like idioms, design patterns, and architecture patterns. It is even thinkable that whole pattern systems drive refactoring activities. By introducing pattern-based refactoring, the difference between refactoring and reengineering might be eliminated. For this purpose, a refactoring method is necessary that includes different levels of refactorings. These challenges, however, should be subject to further research.

2.1.16 Feature Modeling

Feature modeling facilitates architecture analysis because it helps map requirements to architectural entities and implementation artifacts. Therefore, using feature models, software architects can model requirements and support their traceability in an existing software system. Likewise, a software pattern provides an architectural solution that resolves forces (requirements) in a given context. This enables the association of requirements with architectural and implementation artifacts. As patterns and features deal with separation of concerns, feature models and software patterns complement each other. Their ability of depicting commonalities and variabilities makes feature models an appropriate tool for domain engineers and application architects. Furthermore, feature models are applicable for visualizing relationships between patterns, especially when pattern systems and pattern languages are involved. This is illustrated using an example. For the instantiation of the Broker architecture pattern [14], it is mandatory to apply the Proxy pattern [14] (mandatory use = commonality). The Proxy might be realized using the Forwarder-Receiver pattern as base [14] (optional use). In addition, a broker which provides extensibility might leverage either an Interceptor framework [85] or a Component Configurator [85] for this purpose (variability). The feature model in Figure 3 does only present a subset of all patterns involved for the sake of brevity.
2.1.17 Software Metrics and Patterns

It is important to mention a difficult issue that appears when software architecture analysis is performed on a pattern-based software system. This problem relates to software metrics. To illustrate the problem, the Cyclomatic Complexity introduced by McCabe [54] should be applied to the Proxy pattern [33][14]. The Cyclomatic Complexity (CC) for a graph $G$ representing control flow in a software system is defined as follows:

\[
\text{Cyclomatic Complexity} = E - N + P
\]

Where:

$E$ denotes the number of edges in $G$;

$N$ denotes the number of nodes in $G$;

$P$ denotes the number of connected nodes in the graph $G$.

McCabe specifies that a value of
Chapter 2 - Patterns and Middleware

1-10 implies the architectural structure is simple,

11-20 implies moderate complexity and more risk,

21-50 indicates a complex, high risk program, and,

> 50 denotes an untestable program with very high risk.

Applied to the Proxy pattern (see Figure 4) and assuming that 50 clients connect to an original object using the same proxy, cyclomatic complexity yields $CC = 53 - 53 + 53 = 53$.

![Figure 4: Cyclomatic Complexity applied to the Proxy Pattern.](image)

In other words, the usage of the Proxy pattern in this context leads to an untestable program with very high risk according to the definition of cyclomatic complexity. The high value in this context originates from the fact that all client-to-proxy connections are counted although these runtime object relations originate in the same static class relationship. There are two possible implications software architects should consider for pattern-based software architecture:
• When software metrics are applied these effects should be taken into account, especially when tools are used to calculate such metrics.

• For metrics such as Cyclomatic Complexity, the formulas should be changed to reflect and solve the described problem.

2.1.18 Summary

In principle, the development of each software system always starts from scratch. Nevertheless, software developers and architects often face recurring problems which already have been solved successfully in similar contexts. The philosophy of software patterns consists of conserving and documenting this expert knowledge, thus making it systematically (re-)usable. Engineers knowledgeable about such re-usable “design components”, can apply these patterns instead of constantly re-inventing the wheel. In the descriptions of patterns functional characteristics as well as non-functional consequences of the solution are addressed. Patterns offer a vocabulary, which noticeably improves communication and the common understanding in teams. Since different levels of granularity and abstraction exist in a software system, also different kinds of patterns are available. Architecture and design patterns address problems at the design level. While architecture patterns affect the whole software architecture supporting strategic design decisions, design pattern affect subsystems and tactical design. Idioms help solve implementations-specific problems. Best Practice patterns enable engineers to leverage concrete system platforms. In addition to patterns for architectural design and implementation further types of patterns are available focusing on other phases and activities of software engineering. Patterns do usually not appear in isolation, but are related with other patterns in pattern catalogs, pattern systems or even pattern languages. The two latter variants have the characteristic that the patterns contained exhibit close relationship with each other, because they use themselves mutually or solve complementary problems. Pattern encyclopedias contain short descriptions of numerous patterns which are organized using various ontologies. Each short description also includes information where to find the full pattern description. To that extent encyclopedias serve as good ref-
Chapter 2 - Patterns and Middleware

Apart from the assistance for the active construction of a software system, patterns help understand existing architectures. Extracting new patterns from a software system is called pattern mining. By means of pattern mining experts have found proven problem solutions and could abstract many of them to patterns, mostly by scanning through program families or platforms.

2.2 The Domain of Distribution Middleware

2.2.1 Motivation

As pointed out in section 1.6 this thesis focuses on distribution middleware as an example domain to investigate the applicability of patterns for understanding and analyzing software architecture. I’ll introduce pattern systems in the subsequent chapters that reveal the key architecture concepts of the Remoting Middleware domain. For that purpose the concrete definition of the domain including its boundaries (scope) and the core requirements of the domain itself need to be specified. Thus, sections 2.2.2 and 2.2.3 will provide these prerequisites.

2.2.2 Classification and Characteristics

In many solution architectures communication middleware serves as the central backbone which helps to glue all core parts together. It is obvious that the selection of the appropriate middleware is a critical issue in any development project. Unfortunately, in many projects this decision is driven by other than technology or requirements related forces. Some software engineers do not recognize that the selection of inappropriate infrastructure technologies or tools can be problematic. Even if architects are ready to base this decision solely on use cases and requirements, it can be incredibly difficult to move to the right direction. As Andrew Tanenbaum once said with respect to standards "there are so many that it is difficult to choose from". Hence, the first question is: what kind of middleware is appropriate for what kind of problem? To answer that question it is important to get an overview of middleware types and paradigms.
Basically, the following kinds of middleware exist:

- **Messaging Middleware**: applications send messages to each other. Message contents are application-specific while the structure and header information is specified by the MOM (Message-Oriented Middleware). Messages might be sent from one peer to exactly one other peer (end-to-end or queue-based messaging) or (anonymously) from multiple publishers to multiple subscribers (publisher subscriber or topic-based messaging).

- **Remoting Middleware**: hides all communication details from developers by extending conventional operation calls over the network. Clients and servers can almost be implemented as if they were residing in the same address space. All communication issues are handled behind the scenes by glue components that are typically generated using tools.

- **Eventing Middleware**: focuses on distributing fine grained events. Very similar to messaging middleware. Eventing frameworks provide a reactive approach where receivers register with the framework to obtain events.

- **Distributed Transaction Monitors**: provide transactions management across different components. They are not covered in this thesis as transaction monitors today are mainly integrated into the other types of middleware.

- **Service-oriented Middleware** is a kind of meta-middleware approach mostly used to integrate other types of middleware with each other. Prominent example: XML Web services which are particularly useful in business-level integration scenarios. SOA (Service-Oriented Architecture) can also be considered a set of architectural principles with XML Web services being only one example. Thus, SOA will be addressed separately in this thesis.
• **Peer-to-Peer Middleware**: a combination of Messaging middleware and Eventing middleware where the locating of remote peers happens through discovery algorithms instead of relying on centralized repositories.

• **Multi-Agent-Systems (MAS)** denote distributed systems where multiple, autonomous, moving agents cooperate to achieve a common goal.

• **EAI** (Enterprise Application Integration) systems are basically built on top of the aforementioned middleware types.

When software architects must cope with the selection of appropriate middleware solutions, the first distinction should be whether the problem domain requires a more method-invocation based or more message-based approach. Asynchronous, non-blocking operation is one of the important issues here. Despite of the fact that some remoting middleware technologies have also introduced asynchronous but blocking method invocations (e.g., CORBA), messaging is more appropriate for asynchronous communication. Asynchronous non-blocking communication basically means, that the sender for its subsequent processing does either not expect a result from its communication peer or does not need the result immediately. Hence, senders and receivers should be decoupled. Example: sending a purchase order to the order processing subsystem.

Another issue for the preferred usage of messaging is when loose coupling between communication partners represents an inherent necessity. For example, a server that is sending notifications to different receivers which are interested to subscribe for obtaining different messages from different senders. A further example is the provisioning of advanced communication styles such as broadcasting or single request - multiple replies. These styles do not map to existing programming languages. Thus, it makes no sense to provide them through remoting middleware. Last but not least, scenarios where communication links are not reliable are more appropriate for messaging. Example: a mobile phone that lost connection with the network.
If, however, the sender needs an "immediate" result from a specific communication partner, remoting is much more suitable. Example: asking a credit card company for validation of a specific card in a Point-of-Sale system. The advantage of remoting middleware solutions is the transparency they provide. Basically, developers can ignore all communication details. But that is exactly the problem. Developers often tend to leverage remoting middleware as if they were developing non-distributed systems without taking issues such as latency into account. The performance of such systems does often suffer exactly for that reason. Another problem with transparency is hiding of details can be a disadvantage in terms of tracking errors. Because everything is hidden, error causes also are hidden.

**Messaging** middleware can be extended to provide remoting. There is a simple reason for this. Every remote invocation can be separated into two message transfers. A request message is transferred to a receiver which then sends back a result message to the originator. Actually, all remoting middleware is built upon some kind of messaging infrastructure. In contrast to some claims, the opposite is not true as asynchrony with non-blocking operation can not be guaranteed when remoting middleware is used as the underlying base for a messaging layer.

**Peer-to-Peer** middleware simply combines messaging and eventing with discovery strategies. Instead of looking up a receiver's location from a known repository, it is much more reliable to discover a resource, especially when the same resource is available more than once in the network. In a Peer-to-Peer system a consumer or sender does not care about which concrete provider or receiver it is using. Thus, Peer-to-Peer approaches are particularly helpful for coping with decentralization. In theory, a similar approach is possible for remoting middleware. Instead of asking a repository where the server object is located, an alternative approach could consist of using a trading service, ask for a specific service type with specific properties, and get possibly multiple object references returned that meet the specified type and property constraints. This is the right way if you need a remoting based approach combined with decentralized lookup strategies.
**Service-Oriented Architecture (SOA)** basically means loose coupling. For example, peers in service-oriented systems are implementation-agnostic. The only thing an application sees is the service interface of the application it need to communicate with. The communication itself uses a commonly agreed messaging protocol. This approach can be best implemented using XML Web services and Messaging middleware. SOA is particularly useful to integrate different middleware and application islands with each other.

This section could only scratch on the surface. There are a lot of issues that could have been discussed such as error handling, fault-tolerance, scalability, security. The intention was to illustrate that no middleware paradigm can be a general solution for all problems. Thus, it turns out to be essential to first specify the problem space and then identify which middleware solution is the appropriate one.

There is another aspect with far-reaching significance for software architecture. To meet operational and developmental qualities it is not sufficient considering middleware a transparency layer or black box. When middleware is leveraged as backbone for a software system then a precondition for the development of an efficient and effective software system is the efficient and effective usage of the underlying middleware infrastructure. Actually, this holds for any external component integrated into a software system, but is often ignored in projects. Operational and developmental qualities of the whole software system being developed are directly dependent on the internal software architecture of the middleware. As a consequence, software architects need to be knowledgeable about the inner working and principles of the middleware. In other words, it is not possible to meet any operational or developmental requirements when software and system architecture are in conflict with respect to these requirements. To understand middleware the underlying patterns applied in the middleware provide the right means.
2.2.3 Requirements of Remoting Middleware

2.2.3.1 Method

This thesis focuses on the domain of Remoting Middleware as central example. I won’t analyze this domain to full extent, but rather prove that patterns are helpful to understand the core concepts of remoting middleware solutions. As already pointed out in 2.1.13 a pattern mining process needs to be applied to uncover the central core concepts of a given domain as patterns. Two basic kinds of input are helpful when analyzing a domain:

- Core requirements to be addressed.
- Existing applications, platforms, frameworks, product lines that fall into the domain.

2.2.3.2 Basic Requirements

While the following chapters of this thesis introduce a system of patterns that addresses the domains of Remoting Middleware as well as Service-Oriented Architecture, this and the following sections focus on the challenges that arise in the aforementioned domains.

When providing remoting middleware, software engineers need to balance the following basic forces:

- **Location Transparency**: a client should not depend on the location of any remote object it accesses. Nor should a remote object depend on the location of its clients.

- **Communication**: clients and remote objects should be oblivious to the details of the underlying communication protocols. For the same reason, remote communication should be identical to local communication within the same address space.
In other words, communication should at least support a method invocation oriented approach.

- **Interfaces**: remote objects should be able to describe their service interfaces and clients should be able to consume services using these interface descriptions.

- **Registration**: there should be some means to register remote objects with the middleware infrastructure.

- **Discovery and Binding**: remoting middleware should provide functionality to clients to locate, instantiate, and get access to remote objects.

- **Identity**: identity of remote objects plays an essential role in this context. Many different strategies exist, e.g., one remote object for all clients, one remote object for each client, multiple (replicated) remote objects that appear as one single logical remote object. Depending on this strategy, the reference a client owns to access a remote object might mean different things.

- **Error Handling**: the remoting middleware must handle errors that appear during communication or processing in remote objects and report these errors to clients.

- **Infrastructural Glue**: middleware should support functionality or tools that automatically generate required glue components that integrate clients and remote objects with the remoting infrastructure.

- **Platform Independence and Interoperability (optional)**: one of the intentions of many distribution middleware solutions is to provide connectivity between heterogeneous systems. There are two ways to achieve this objective, either by assuring interoperability to other middleware protocols or by ensuring the availability of the distribution platform on different heterogeneous infrastructures.
• **Optimizations (optional):** it might be important that the middleware offers specific response times or even deterministic runtime behavior in real-time systems. For this purpose, some means such as configurable strategies should be supported.

### 2.2.3.3 Advanced Requirements

The requirements listed in 2.2.3.2 comprise fundamental issues the remoting middleware implementation must cope with. In addition, the following more advanced challenges should be addressed but are not mandatory:

- **Lifecycle Management:** for scalability reasons remote objects as well as their implementations should be activatable and de-activatable on demand.

- **Resource Management:** for scalability also resource optimization strategies are required such as leasing strategies, caching, or lazy evaluation. Resource management is often interwoven with lifecycle management. Thus, both often represent two sides of the same coin.

- **Dynamic Invocation and Reflection:** for ensuring runtime extensibility remoting middleware may offer generic functionality for retrieving meta-information on remote objects at runtime and using this information to dynamically invoke remote objects.

- **Asynchrony:** many application contexts require asynchronous handling of remote communication, e.g., a client should not block until the remote object returns the response.

- **Concurrency:** a remote object should allow concurrent processing of multiple threads.
• **Extensibility**: remoting middleware should be extensible by application developers so that they are capable of integrating out-of-the-band functionality not anticipated by the middleware developers.

• **Context Information**: remoting middleware needs to support piggy-packing additional context information for advanced services such as transactions and location-aware services.

• **Versioning (optional)**: In all non-trivial applications continuous evolution is a driving factor. Thus, clients need to cope with remote object versions.

• **Configuration**: application developers need support to configure the remoting middleware according to their needs. This functionality could be provided programmatically or declaratively using configuration files.

### 2.2.3.4 Additional Services

In addition to the more remoting-middleware-specific requirements introduced in 2.2.3.2 and 2.2.3.3 advanced middleware technologies such as CORBA integrate various services. Some of these services might require the existence of certain extensibility mechanisms in the middleware, because they represent cross-cutting concerns.

• **Security**: all functionality required to offer authentication, authorization, non-repudiation, confidentiality, integrity, prevention of attacks.

• **Persistence and Transactions**: if an application’s functionality is distributed across different distributed entities, then this also holds for the application’s state. Thus, functionality is required for keeping the distributed state persistent. The distributed state must also remain consistent when activities (workflows) access different remote objects.
• **Availability and Fault-Tolerance**: To enhance availability, replication of remote objects is a possible approach. If errors occur, some middleware functionality must take responsibility to detect fault situations and react appropriately. Clients should be unaware of strategies used for this purpose.

• **Load Balancing**: If multiple remote objects provide the same service to clients, then load balancing functionality could transparently switch between these remote objects to prevent overload situations and optimize throughput and responsiveness.

• **Integration**: application developers should be able to integrate COTS components and legacy applications.

• **Management**: If an application consists of distributed objects spread across multiple network nodes, then management functionality provides command and control functionality.

In remoting middleware most of these services are provided as combinations of middleware extensions and remote objects running on top of the middleware which, however, results in a significant increase of complexity, as developers are forced to deal with service integration themselves. As a consequence, component-based containers have evolved that take responsibility for all infrastructural integration problems. The thesis won’t cover component containers but instead I’d like to refer to existing literature [106].

### 2.2.3.5 Next Step

For the issues raised in 2.2.3.2, 2.2.3.3, 2.2.3.4 the analysis method proposed will try to find existing patterns that already address the challenges or, if not applicable, uncover new patterns. As there might be patterns that solve multiple requirements, the pattern mining process provides an additional activity to look for higher level abstractions. And
as there might be patterns for solving sub-problems of the analyzed problems, an appropriate divide-et-conquer strategy has been integrated.