1 Introduction

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.” —C. A. R. Hoare.

ACM Turing Award Lecture, 1972

1.1 Relevance of Software

Only a few decades ago, a very small number of industrial products integrated microprocessor systems. Some commercial vendors offered personal computers that were only available to a handful of experts in larger companies or research institutions. In the meantime, powerful personal computers have become part of virtually all consumer and industry products. Companies such as GE, SIEMENS or Philips that offer different product lines for industry domains such as health, telecommunication, automation, automotive, or power have employed huge numbers of software developers over the last years, because software turned into one of the most important assets in almost every product, be it a mobile phone, a control system, or an automation system.

Developments such as the arrival of the World Wide Web initiated another trend, the migration from isolated computer islands to fully connected networked systems. A decade ago, only a few experts knew how to access the Internet and how to set up local or wide area networks. Networking and Internet access have become commodity solutions in
recent years. Today, most regular computers users are familiar with home and mobile networks. In industry, communication networks have gained high importance as well. For example, the electronics system of a modern car typically contains dozens of CPUs that are connected using industrial network solutions such the CAN bus.

Nowadays, we are in the middle of another evolution: the large-scale network-based integration of various devices and computer systems. Terms such as Pervasive Computing or the Internet of Things were coined to emphasize this evolution. Traffic management is a good example for this: car electronics devices such as navigation systems are increasingly integrated with telematics infrastructures to exchange information and consume services such as toll collection services or traffic information. Another future vision anticipates cars capable of detecting traffic jams and automatically notifying other cars about this problem so that immediate actions can be taken. This approach is called ad-hoc networking.

Some experts believe that with respect to importance of software systems we have only seen the tip of the iceberg.

There are several observations and conclusions we can draw from these developments:

- Software engineering has become one of the most important disciplines, even for industries that formerly considered themselves pure hardware manufacturers.

- Due to growing complexity of products, software systems also become more complex. This problem even increases due to additional network and integration requirements.

- Global competition has lead to a dramatic reduction of development cycles and increase of quality requirements. The problem with this development is that the evolution of tools and technologies required for software development is also accelerating. Thus, software engineering is required to provide appropriate answers.
One of the most important implications is that software engineering faces a high degree of complexity so that unsystematic software and product development is doomed to fail. As software systems tend to become larger and more complex, the same holds for the size of development teams. Therefore, it is important to introduce and apply systematic and effective development processes, methods and tools. One of the core assets in this context is software architecture which introduces a sound structure and set of constraints for implementing software systems. When everything else is constantly changing, at least the internal structure of a software system should be as stable as possible and serve as a fix point without unnecessarily limiting flexibility aspects such as extensibility or changeability.

Software engineering alone is not sufficient. Especially, the design of distributed software systems such as healthcare infrastructures requires the availability of distribution technologies which enable connectivity between different parts of the software system. Distribution middleware does provide those kinds of connectivity solutions. Thus, the art and craft of software engineering consists of balancing different forces, e.g., methods for designing software architecture and technologies such as distribution middleware.

1.2 What is Software Architecture

The whole history of software engineering, e.g., the evolution of programming languages and programming paradigms, is a continuous attempt of rising abstractions, because abstractions help master complexity. In this context, the complexity of a system depends on the number of its constituents as well as the dependencies between them. The higher the number of constituents or dependencies, the less a human is capable of understanding the system. Let us take the human body as an example.

We could try to understand the human body using a bottom-up approach by considering our body as a mere aggregation of atoms or molecules. All atoms in this naïve physical approach belong to the same hierarchy level. That comes to no surprise as there only is one level of hierarchy in our atomic model. This approach, however, must fail due to the
virtually infinite number of atoms involved and the huge amount of emergent behavior caused by the interactions between these atoms. Hence, in practice, we need to increase the number of hierarchy levels and reduce the number of parts and inter-part dependencies by aggregating low-level concepts like atoms to higher level abstractions. In other words, we should better rely on a top-down approach. For instance, a human body could be structured by its different kinds of organ systems such as the nervous system, digestive system, circulatory system, muscular system, or immune system as well as its external features such as head, skin or spine. With this approach it is also much easier to analyze typical “workflows” of human bodies such as the interactions and parts involved when moving a leg. Depending on what details we are interested in, it is possible to move in this hierarchical model from the top layer (the human body as a unit) to the bottom layer (single atoms or molecules in a body cell). Due to the tree like organization of abstraction hierarchies, we only have to deal with a restricted number of parts in each layer respectively each node in the tree hierarchy. Another observation with this hierarchical abstraction-driven model is that depending on our concrete subject of interest we can take different viewpoints. For example, if we are just interested in the interaction between a leg and the brain, we could focus on the nervous system instead of considering also other parts such as the digestive system. In addition to all static and dynamic relationships between parts, it is also important to analyze the guiding principles. In the human body example one guiding principle could be described as symmetry. For example, there are two eyes, two ears, two legs, and two arms. Guiding principles show the “why” of a system, while parts and interrelationships often are constrained to reveal only the “what”.

But how does the example of the human body map to software systems? Software architecture is also about introducing abstractions, interrelationships and guiding principles. “Software architecture” as a term was introduced in the 1960s by famous computer scientists such as Edsger W. Dijkstra. In the literature various definitions of Software Architecture have been published:
According to [17] software architecture “is the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.”

In [14] it is defined as follows: “A description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system.”

According to Wikipedia (http://en.wikipedia.org/wiki/Software_architecture) “software architecture or software systems architecture is a representation of a software system, as well as the process and discipline for effectively implementing the design(s) for such a system. It is a representation because it is used to convey the information content of the related elements comprising a system, the relationships among those elements, and the rules governing those relationships. It is a process because a sequence of steps is prescribed to produce or change the architecture, and/or a design from that architecture, of a system within a set of constraints. It is a discipline because a body of knowledge or a general set of principles of (software) architecture is used to inform practitioners as to the most effective way to design the system within a set of constraints.”

Although there are various definitions of Software Architecture, it is not much of a challenge to extract some commonly agreed properties that also closely resemble what we learned in the human body example. Thus, we could simply remove “Software” from the following property descriptions and apply the same principles to non-software systems:

- Software Architecture is about the parts of a software system as well as their relationships.
- Software Architecture deals with the underlying principles that drive the design of a software system.
• Software Architecture introduces different views that reveal the relevant functional and non-functional (i.e., operational and developmental) properties of a software system.

• Software Architecture is also about the process how to design, maintain, and evolve a software system.

So, does software architecture by its mere existence represent a quality guarantee for a software system? In the beginning of software development, many software programs were developed for particular target machines by single (or small groups of) individuals using native machine language or early programming languages. Even these software systems inherently contained a software architecture. However, most systems were developed in an ad-hoc approach without explicitly taking functional and non-functional properties into account, nor did they follow any guiding principles. Thus, software architecture can only ensure the quality of a software system, if a systematic software engineering process maps the required and desired properties of a software system to appropriate software architecture and eventually to a concrete implementation using a piecemeal approach and following guiding principles. A lot of ingredients are available for this systematic development of software systems: development processes such as RUP (Rational Unified Process) or Scrum define the steps, activities, roles, phases, tools, technologies and best practices necessary to develop a software system using a predefined process (model). Notations such as UML (Unified Modeling Language), ADLs (Architecture Description Languages) and DSLs (Domain-Specific Languages) introduce the means to express a software system from different perspectives either textually or graphically. Such rather generic technologies support development phases such as software design and implementation in an abstract, domain-agnostic way. They, however, don't provide any help when software engineers are facing a concrete problem in software design or implementation. For instance, how can we structure a software system so that, whenever its graphical user interface changes, there is no impact on other domain-specific parts? To address those kinds of challenges, software scientists came up with the notion of software patterns more than 10 years ago. A software pattern describes
how a recurring problem that arises in a specific context can be addressed by a generic solution scheme. With other words, if, for example, a software architect faces a problem she or he has already solved before, she or he will try to apply the same solution recipe instead of wasting time by reinventing the wheel over and over again. Thus, software patterns are particular useful when engineers are designing the software architecture of a software system under development. Architecture patterns represent software patterns that are applicable to define the core architecture of a software system. Examples include Pipes and Filters, Layers, or Broker [14]. Patterns are not restricted to software design or even software systems. There are different flavors of patterns for different phases in the development process such as analysis patterns as well as for different granularities such as design patterns [33] or architecture patterns – originally also referred to as architectural styles such as in [16].

1.3 Software Product Lines

Many companies offer variants of the same product for different customer groups or system environments. Prominent examples include consumer products such as a series of mobile phones, car manufacturing, and community/professional/enterprise versions of standard software products. Developing all variants of the same product from scratch would lead to high development and production costs as well as unacceptable development time. Especially, the car manufacturing industry has illustrated how lean and just-in-time production can be efficiently leveraged to address this problem. Re-using components in all products definitely represents a possible and important means. However, opportunistic or accidental re-use of parts does not provide an efficient and effective solution as [11] has proven. Instead, reuse and development of these product variants must be based upon a systematic and efficient approach. For this purpose, software product development typically groups similar products to software product families and explicitly
defines common features\textsuperscript{1} for all members of the product family as well as features that vary across members. The development of these members is then based upon a Product Line Engineering approach which, by the way, is the same approach other industries have introduced for manufacturing hardware.

CMU SEI (http://www.sei.cmu.edu/productlines/) defines a software product line (SPL) as “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.

Very often, the collection of core assets shared by all members of a product family is not just a loose collection of components, but provides a whole platform. A platform is an application framework that implements a runtime system to host applications. In program families platforms typically represent the most important core asset. Such a platform is often built as a framework which denotes an architectural skeleton instead of providing a complete solution for a specific application domain and can be configured for a concrete application.

In Product Line Engineering the following artifacts will be defined\textsuperscript{[77]}:

- All products that will be part of the product line, the so-called scope.
- Common assets of the program family members (derived from common functional and non-functional requirements).
- A production plan that specifies how each product family member can be created from the common assets.

\textsuperscript{1} [11] defines a feature as a logical grouping of requirements which should be semantically grouped together from a customer perspective.
Chapter 1 - Introduction

Fundamental artifacts of SPL include the common software architecture as well as commonly used components or subsystems. Software architecture represents a fundamental issue for product line engineering because the SPL architecture specifies how to systematically re-use common assets (see [77]) by introducing the software architecture of each family member as well as all relevant interfaces\(^2\).

One important prerequisite for successful product line engineering is the specification of variabilities and commonalities within a SPL, which is determined using a Commonality/Variability analysis. For instance, each car in a specific product line might use the same engine (commonality). But depending on the type of car (convertible, cabriolet) the different product line members might use different types of chassis (variability).

1.4 Middleware – Foundation for Distributed Systems

Since the Internet has turned from an infrastructure only a few experts were able to handle to a commodity solution everyone, even casual and non-expert users can leverage, the relevance of distributed software systems has dramatically increased. Today, most software systems are either internally based upon networked environments such as the Internet or can at least integrate with such networks. Examples for distributed software systems are E-Commerce systems, telecommunication solutions such as Skype or Peer-to-Peer file sharing networks, standard platforms such as SAP R/3, or electronic business process and employee portals. The core property of a distributed software system is that all or some of its parts reside in different nodes of the network. This might be as simple as a database management system that runs on a dedicated server on the network and is accessed by the other parts of the software system. This might also be as complex as a Grid or Peer-to-Peer solution where the software system consists of a large number of (mostly small-grained) entities that are distributed across the network.

\(^2\) It is important to mention that in Software Product Lines, there are, at least from a logical viewpoint, two different kinds of development teams, one for the development of the common core assets used by all program family members as well as a team responsible to develop one concrete product instance.
By the way, such Peer-to-Peer systems in some respects are very similar to a population of ants, for instance when facing the large variety of complex communication relationships between all parts. As in Software Architecture, the first distributed software systems ever deployed on a network were built with ad-hoc approaches using low-level communication APIs (programming interfaces). Very soon, the liability of this approach became obvious. For instance, software engineers face the following problems when using such a low level approach:

- Low-level communication APIs offer only a basic set of rather primitive functionality for establishing and maintaining connections between distributed components and for sending data packets between these components. If more sophisticated functionality is required such as mapping data from one format to another, this has to be provided manually by programmers.

- Low-level communication APIs often depend on a concrete operating system, programming language or software infrastructure. If a distributed software system consists of components running on different operating systems or infrastructures, low level approaches do either not work or require injecting a whole bunch of complexities and dependencies into the software system which badly impacts properties such as maintainability and extensibility.

- Low-level communication APIs lead to a mixture of domain-specific and communication-specific aspects. The developer can not concentrate on providing a specific solution for a specific problem such as writing a software system that implements a Web shop, but must additionally deal with communication aspects.

In order to address these challenges, computer science introduced middleware solutions. According to Wikipedia (http://en.wikipedia.org/wiki/Middleware) “Middleware is computer software that connects software components or applications. It is used most often to support complex, distributed applications. It includes web servers, application servers, content management systems, and similar tools that support application development
“and delivery”. Starting very simple as plain vanilla Remote Procedure Calls, today’s middleware provides much more capabilities. Middleware is traditionally located in the middle between the operating system and the applications, which is the reason for its name. It typically provides advanced communication styles as well as additional services such as security or transaction functionality. Instead, of directly using low level APIs, an application may integrate middleware that shields the application from all details of network communication. As middleware technologies have rapidly evolved into very efficient communication platforms, most distributed systems nowadays integrate distribution middleware. Middleware solutions provide a stack of hierarchy layers and abstractions that offer different kinds of transparency to application developers. Transparency comes in different flavors. For example, when it is invisible to an application where a remote peer resides in the network, the remote peer can even be migrated to another network node or replicated across different nodes without impact on the application, thus providing properties such as location transparency, scalability, or fault-tolerance.

1.5 Challenges of Middleware and Distributed Software

My own research on middleware and software architecture started in 1996 after POSA 1 [14] arrived and during work on POSA 2 [85]. The idea was initiated by a colleague, Norbert Portner, from the Automation & Drives business group at Siemens AG. Norbert Portner complained about the fact that all these middleware solutions such as Microsoft COM/DCOM required a steep learning curve and initially made the whole development team less productive. This was due to the fact that middleware manuals were either written by system-oriented software engineers (e.g., Microsoft COM/DCOM) or as standards specifications (e.g., CORBA). It was thus my proposal to Norbert Portner to come up with middleware manuals that focused more on the architecture of these middleware technologies. For the POSA books I had already figured out that all these different remoting platforms at the core level revealed identical or at least very similar design strategies.

---

3 i.e., making remote procedure calls appear as if they were local machine calls within the same address space.
Hence, they were also a good place for pattern mining. Patterns such as Broker, Proxy, Forwarder Receiver and Client-Dispatcher-Server [14] were extracted from these platforms and then abstracted as pattern descriptions. Other patterns such as Layers or Reflection [14], Adapter, Abstract Factory (both in [33]) had also been applied to the middleware infrastructures. Of course, the middleware developers that time didn't know they were using “patterns”. Thus, my original idea consisted of leveraging patterns as a means to understand these platforms and provide this knowledge to other people. Uncovering the architecture of middleware solutions offers the opportunity to obtain both a high level abstraction of the middleware (the what) as well as a glimpse on the design rationale (the why).

It also turned out very soon in various software development projects where I worked as software architecture consultant that software architects had very little knowledge about middleware and considered middleware as only one of many components in software development projects. Unfortunately, this resulted in products that failed to meet their requirements. Especially operational issues such as performance or scalability proved to be critical. The reason for these inadequate designs was transparency. Remoting middleware platforms are the attempt to encapsulate system details of communication and distribution behind object-oriented facades. For a software engineer the development of remote objects and clients should be almost identical to the development of local objects and clients. If, however, software engineers rely on this kind of transparency their software architectures will fail to perform well in terms of operational qualities, because meeting operational qualities needs consideration of the cost of communication and distribution. What is the reason for this fact? Let us zoom into some details to answer that question. Actually, each remote invocation requires a whole bunch of activities:

- On the client side, the request parameters as well as additional attributes must be marshaled into a network protocol conformant format. If not already available, a connection needs to be established to the remote object.
• On the server side, the request needs to be received, the appropriate servant activated, the data demarshaled, and the request dispatched to the servant.

• For the response sent from the servant to the client all of these steps need to be performed in the opposite direction.

It is essential to know in this context that additional artifacts such as proxies are also involved. Nonetheless, this is just the tip of the iceberg. Add to this, additional processing activities such as security-related or transaction-related processing steps. In the end, each remote method invocation that appears as an atomic activity to clients and remote objects in reality requires a large number of processing steps. That kind of transparency does not only hide infrastructural issues but also important details such as the cause and origin of error conditions. Thus, the best strategy for building distributed systems is not to communicate at all and as this is infeasible in any distributed software system to at least reduce communication to a minimum. For this purpose, software architects and developers require knowledge about the internal architecture of the middleware. This knowledge is important for coping with operational and developmental requirements, while transparency is still important, because it allows separating business logic from cross-cutting concerns. In summary, patterns help to efficiently and effectively leverage middleware platforms.

Analyzing middleware solutions by uncovering and mining their underlying principles and patterns enables software engineers not only to understand the middleware domain but also to compare different middleware solutions with each other. For instance, most remoting middleware solutions apply the same patterns in their core design, but also vary in some details or add additional properties or functionalities. Actually, the domain of remoting middleware can be considered a product line or architectural framework with the base assets consisting of reusable design artifacts (patterns) and reusable components. Concrete middleware solutions such as CORBA are then instantiations of the product line. Hence, patterns help to express the domain of remoting middleware in a kind of idiomatic language. For the same reason, architectural paradigms such as Remoting (Dis-
tributed Objects) or Service-Oriented Architecture can be described using patterns as it will be illustrated in this thesis.

Knowing the patterns of a software system implies knowing how to change, refine and evolve it. The same holds for the evolution of architectural paradigms. If for example, proxy objects are placed in the processing between client and remote object, then a whole pipeline of processing steps can be transparently added between these proxies using the Pipes & Filters pattern [14]. As this pipeline remains hidden from clients and remote objects, it enables easy extension and configuration of the middleware. For example, security tokens can be added, processed, and removed without any impact on clients and remote objects. Even the whole transport protocol and marshaling layer may be exchanged. Windows Communication Foundation is a prominent example for this kind of approach. Therefore, architecture patterns such as Broker [14] help specify the strategic base line architecture while finer-grained design and analysis patterns provide its tactical refinement.

Last but not least, the knowledge of the middleware-related software patterns enables software engineers to develop their own middleware. This smells like reinventing the wheel in enterprise contexts where many COTS middleware products are already available such as CORBA, RMI, EJB and Windows Communication Foundation, to name just a few examples. However, in the area of embedded systems there still is a significant lack of standard middleware. Despite of standards like AUTOSAR (Automotive System Architecture) or Real-time CORBA many projects implement their own middleware to cope with stringent requirements and constraints. Instead of building such middleware from scratch, it is far more effective to learn from other middleware experts. Patterns are the right way for that purpose as they capture well proven experience and expertise. The author has been involved in some Siemens-internal projects where proprietary middleware had to be developed which went particularly well using a pattern-based approach.

In summary, patterns turn out to be a universal tool for software architects. They are not constrained to forward engineering of new software systems, but also as a means for
analyzing and understanding existing software systems. Pattern-based software architecture analysis for a single application is useful when the application reveals high pattern density, at least in its design center. Analyzing program families and platforms for a specific domain, however, offers a much better opportunity to mine software patterns which often represent the core assets of a program family or platform. The reason lies straight at hand: developers of program families and platforms must perform a Commonality/Variability analysis in the beginning, explicitly address domain-specific core concepts, and instantiate re-usable architecture and design artifacts. Hence, understanding program families and platforms by analyzing their architecture concepts in terms of patterns and proto patterns seems to be a reasonable approach.

1.6 Research Questions

The overall objective of the underlying thesis can be described as: It is commonly accepted that software patterns help construct solutions using proven design artifacts. Are patterns also beneficial in the opposite direction, especially for pattern-based software architecture analysis? In other words, do patterns provide a useful abstraction and adequate means to better understand and analyze existing software architecture?

To be more concrete, this thesis focuses on program families and platforms in the domain of distribution middleware, in particular on remoting and service-orientation. These domains respectively paradigms are sufficiently complex. In addition, they are widely used in software applications. Thus, they represented a good source for the groundwork required for the thesis.

The individual research questions the thesis tries to address are as follows:

[Q1] Can we provide a scientific description and classification of software patterns that includes the usage of patterns for architectural analysis?
[Q2] In order to understand the key concepts of distribution middleware: can we systematically classify and describe middleware in general and remoting middleware in particular as well as Service-Oriented Architecture in terms of requirements (commonalities) and concepts?

[Q3] Understanding remoting middleware and Service-Oriented Architecture from an architectural perspective:

  a. Is it possible to introduce new or use existing patterns and proto patterns to describe the relevant core concepts, e.g., can remoting middleware being described with patterns?

  b. Are software patterns applicable to describe and compare the different remoting middleware technologies?

  c. Can software patterns even help describe paradigms such as remote objects or service-oriented architecture?

  d. How can software patterns be evolved and refined with more details using additional software patterns to further cover the domain of remoting middleware?

  e. Can best practice patterns be derived that help application programmers to leverage the remoting middleware?

[Q4] Does architectural coverage of a domain or software system necessarily imply complete coverage by a pattern language or is it sufficient to express only the core concepts using patterns?
1.7 Research Approach

Software Architecture is primarily about the hierarchical decomposition of a solution space into parts as well as about the static and dynamic relationships between these parts. For a single software system, there might be different perspectives on software architecture, e.g., a deployment view or a static view, which leads to different details and granularities of decomposition. While the process of organizing the software architecture into appropriate parts as well as the interdependencies between these parts might be formalized to some extent\textsuperscript{4}, a mathematical formalization is typically not possible in practice due to problem complexity. Thus, software practitioners need assistance in constructing software systems with high quality even in the absence of mathematically sound formalisms and methods. Moreover, formal methods require exact input parameters to produce exact results. Unfortunately, requirements for software systems – which represent the input parameters - are often vague, incomplete, inconsistent or contradicting, and, sometimes even worse, are subject to change during project lifetime. Yet another problem for software architects are the properties and limitations of existing tools and technologies required for creating the implementation. Usually, there is a huge gap between software architecture and its implementation - the process of turning a specification into an implementation is far from being straightforward. Nonetheless, the goal of software engineering is to produce code. Given all these facts, the question is: how can software architecture help to develop high quality software systems despite all uncertainties and vagueness in a normal software development project? Instead of trying to strictly follow a mathematical approach, software architecture methods heavily rely on empirical studies and experiences. For example, if a specific solution proves to be applicable to almost all problems in a similar context, this solution can be abstracted and expressed as a general architectural principle, mostly documented as a software pattern. The research for this thesis applied the empirical approach to analyze existing middleware solutions to understand the underlying architectural principles used. It turns out that different middleware

\textsuperscript{4} At least in theory.
technologies apply the same or similar patterns because each of these patterns solves a particular problem in an appropriate way. This solution works in space and time, i.e. across different middleware solutions that have constantly evolved over the last decade. The problem with this analytical approach is how to systematically detect and abstract architectural principles, because the same patterns might be implemented in infinite ways. For analyzing existing software, the only sources available are often the API documentation and the implementation. Thus, the main challenge consists of proving that two different implementations have implemented the same architectural principles. Fortunately, software patterns have been used for a long time in middleware frameworks and platforms, even if they weren’t coined “patterns”. Thus, research can leverage the knowledge of existing pattern catalogs in order to detect whether specific patterns have been used in a concrete software system when a well-known is going to be solved. Thus, a kind of feedback loop can be instantiated: Existing patterns might be applied to develop new software systems (forward engineering). From existing software systems, a pattern mining process is applicable to extract new software patterns (backward or reverse engineering) which can then be applied in forward engineering.

In summary, the research approach in this thesis consists of

- Empirical analysis of existing solutions in a given domain (in our case that of distribution middleware). Extraction of commonly applied architectural principles (patterns).

- Defining an appropriate foundation of architectural principles (obtained in the analysis step) and their combination in order to cover the problems in the chosen domain as well as to set a basis for its further evolution. This also denotes an empirical approach.

- Implicit proof of concept: The results reveal inherent quality because all architectural principles are taken from existing high quality solutions. Since these solutions apply the same principles (i.e., patterns), the approach implies sufficient
qualitative evidence. Our experience was not constrained to analyzing existing middleware solutions built by third parties. We were also involved in projects where middleware solutions were developed using exactly the same patterns as introduced in this thesis. An example is the Open Source Framework ACE (Adaptive Communication Environment) as well as TAO (The ACE ORB) which is a CORBA system on top of ACE. Both solutions apply many patterns from [85][33][14].

1.8 Summary

In the beginning of this chapter I tried to illustrate why software engineering and middleware are key aspects in almost all product development projects. The subsequent sections then provided an overview of these aspects introducing software architecture, product line engineering, and distributed systems in more detail. A section about challenges of middleware discussed the problems software architects and engineers face when developing distributed systems. Then, the research questions were introduced this thesis is going to answer. Finally, the concrete research approach of the thesis was introduced that shows how the research questions were addressed during the work for this thesis.

1.9 Remainder of this Thesis

The remainder of this thesis is structured as follows:

- **Chapter 2** of the thesis covers two areas: First, a new systematic overview of software patterns and their role for software architecture is given. A pattern mining process for systematic pattern-based analysis of domains and applications is introduced. This sets the groundwork for the rest of the thesis. In part two of the chapter, the thesis introduces the domain of distribution middleware and all requirements to be addressed by a pattern-based analysis.
• **Chapter 3** then analyses the area of remoting middleware in more detail by introducing patterns which address the basic and some of the more advanced requirements described in chapter 2.2.

• **Chapter 4** adds the area of XML Web services and also applies the pattern-based approach to the domain of web-based distribution middleware.

• **Chapter 5** then introduces a new abstraction layer. Instead of regarding XML Web services an individual solution for Web-based remoting middleware; the thesis introduces core principles of Service-Oriented Architecture (SOA). SOA in this context denotes a paradigm consisting of a set of architectural principles, all of them expressible by patterns or proto patterns. It can be thus shown that even architectural principles can be described using software patterns.

• **Chapter 6** describes the Broker Revisited pattern. This pattern illustrates how the core concept of remoting, specified by the Broker pattern, can be subject to further modification and refinement, thus digging deeper into the internal architecture of remoting middleware.

• **Chapter 7** introduces the Activation patterns as an example for a more advanced issue in the domain of remoting middleware.

• **Chapter 8** goes even one level deeper and provides the Context Object pattern that offers some optimizations for the internal working of remoting middleware.

• **Chapter 9** summarizes the result of the thesis by answering the research questions raised in 1.6 and provides some possible topics for future research.