CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Architectural Styles for Early Goal-driven Middleware Platform Selection

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

von

Dipl.-Inform. Simon Giesecke

Gutachter:

Prof. Dr. Wilhelm Hasselbring
Prof. Dr. Ralf H. Reussner (Univ. Karlsruhe)

Tag der Disputation: 2008-09-26

Contents

Summary

Business information systems are middleware-intensive applications, i.e. their structure and behaviour is significantly influenced by the chosen middleware platform. The selection of a suitable middleware platform is a task in the development process that is critical in fulfilling the system's quality requirements, e.g. availability or time efficiency. A systematic selection should be done as early as possible within a development project, i.e. on the architectural level. Up to now no adequate method for selecting a middleware platform in a concrete project context exists, thus often ad-hoc decisions are made.

As a solution, the MidArch approach was developed, which is presented here. The main constituent is the MidArch Design Method, which uses a repository of MidArch Styles and a MidArch Style Taxonomy for supporting the systematic selection of a middleware platform and to guide the specification of software architectures.

A middleware-oriented architectural style (MidArch Style) is a model of a middleware platform, which captures the structural constrains that are imposed upon the application architecture by the underlying platform. From another point of view, a MidArch Style characterises the family of software architectures that can be implemented on a given middleware platform.

Within the MidArch Design Method, a MidArch Style Taxonomy is used as the basis of a stepwise selection process of candidate MidArch Styles, which match the given quality requirements. Consequently, software architectures are specified that conform to these styles, and the architectures are evaluated against the quality requirements. The evaluation results are lifted from the level of individual architectures to the level of the MidArch Styles by comparison. This information is stored in the repository as annotations to the respectives MidArch Styles. In subsequent applications of the MidArch Design Method, these are used to refine the selection process.

For evaluation, two case studies using industrial systems are presented. Furthermore, the thesis contains contributions in the areas of software architecture foundations and the definition of architectural styles and patterns.

Zusammenfassung

Betriebliche Informationssysteme sind Middleware-intensive Anwendungen, d.h. Softwaresysteme, deren Struktur und Verhalten durch die Middleware wesentlich beeinflusst werden. Bei der Entwicklung ist die Auswahl einer Middleware-Plattform eine kritische Aufgabe in Hinblick auf die Erfüllung der gestellten Qualitätsanforderungen, die beispielsweise Verfügbarkeit oder Zeiteffizienz betreffen. Eine fundierte Auswahl sollte möglichst früh in einem Entwicklungsprojekt erfolgen, d.h. auf der Ebene der Softwarearchitektur. Bisher existieren jedoch wenige Richtlinien oder systematische Vorgehensmodelle für die Auswahl einer Middleware-Plattform in einem konkreten Projektkontext, so dass oftmals Ad-hoc-Entscheidungen getroffen werden.

Als Lösung wurde der MidArch-Ansatz entwickelt, der hier vorgestellt wird. Hauptbestandteil ist die MidArch-Architekturentwurfsmethode, die ein Repository von Entwurfswissen über MidArch-Stile und Taxonomien solcher Stile verwendet, um die systematische Auswahl einer Middleware-Plattform zu unterstützen und die Spezifikation von Software-Architekturen zu leiten.

Ein Middleware-orientierter Architekturstil (kurz: MidArch-Stil) ist ein Modell der Middleware-Plattform, das die strukturellen Bedingungen verkörpert, die die Plattform an die auf ihr realisierten Anwendungen stellt. Demnach charakterisiert ein MidArch-Stil die Familie der Softwarearchitekturen, die auf der betreffenden Middleware-Plattform realisiert werden können.

Innerhalb der MidArch-Architekturentwurfsmethode wird eine MidArch-Taxonomie als Grundlage für die schrittweise Auswahl von MidArch-Stilen genutzt, die für die gestellten Qualitätsanforderungen geeignet sind. Anschließend werden Softwarearchitekturen spezifiziert, die diese Stile einhalten, und gegen die Qualitätsanforderungen evaluiert. Die Evaluationsergebnisse werden durch Vergleiche von der Ebene einzelner Architekturen auf die Ebene des MidArch-Stils gehoben. Diese Informationen werden als Annotationen zu den betreffenden MidArch-Stilen im Repository gespeichert und ermöglichen in weiteren Durchführungen der MidArch-Methode eine verfeinerte Auswahl von MidArch-Stilen.

Zur Evaluation werden zwei Fallstudien mit Systemen aus der industriellen Praxis vorgestellt. Weiterhin enthält die Arbeit Beiträge zu Grundlagen der Softwarearchitektur und zur Unterscheidung von Architekturstilen und Architekturmustern.

Acknowledgements

Many people have contributed to me being able to complete this thesis in various ways, and I cannot name all of them here. Those I have omitted unjustly may be indulgent to me.

I would like to thank my parents for laying the foundation to my education and supporting me through my studies and dissertation.

My friends needed to bear with my tempers that have been strained from time to time by the ups and downs of working on the dissertation. Among my friends, I am deeply grateful to Sören Wehrmann, with whom I had the honour to spend many blissful hours. I do not know if I would have had the strength to complete my work without his support and encouragement, in particular in the final phase of writing and preparing the defense of my thesis.

I would like to thank my colleagues in the Software Engineering Group and later in the Business Information Management Division at OFFIS for the enjoyable atmosphere and interesting and thought-provoking discussions. I wish to extend this to the members of the Palladio Group, most notably Steffen Becker. I worked most closely with Matthias Rohr and André van Hoorn, who also provided valuable feedback on drafts of this dissertation. I shall not forget to mention Manuela Wüstefeld, who did a great job as secretary of the Software Engineering Group. She always had a sympathetic ear for any kind of concern.

I had the opportunity to attend numerous workshops and conferences, where interesting discussions took place. Among these, I need to mention Paris Avgeriou and Uwe Zdun, who organised a session at EuroPLoP 2008, where I received the most valuable feedback.

Several students contributed to the goals of this thesis by their bachelor, master and diploma theses, which includes Jürgen Englisch, Rainer Hilbrands, Michael Gottschalk, Reiner Jung, Angela Eiben and Jürgen Ulbts. In particular, I would like to thank Johannes Bornhold and Florian Marwede who actively helped in writing papers and getting them published in addition to their obligations.

I would also like to thank those senior researchers and professors, who took on providing scientific, professional or personal advise to me, which includes Hans-Jürgen Appelrath, Susanne Boll, Claus Pahl, Matthias Riebisch, Michael Sonnenschein, Ulrike Steffens and Ute Vogel.

Last, but not least I would like to thank Ralf Reussner, who not only offered to act as an examiner of my thesis, but was also a personal and

scientific mentor to me, and Willi Hasselbring who supervised my thesis and was always available to me in the need of advise.

1 Introduction

Within this chapter, we first lay out the motivation for conducting the research that constitutes this PhD Thesis, the early selection of middleware within a development project (Section 1.1). Then, we describe our approach to solving the problem which builds upon the concept of middleware-oriented architectural styles (Section 1.2). The scientific contributions of the thesis are summarised in Section 1.3. The chapter ends with an overview of the outline of the thesis (Section 1.4).

1.1 Motivation

In the development process of a *business information system*, one critical task is the *selection* of a suitable *middleware product* that enables the final system to fulfil given quality requirements. These quality requirements usually concern conflicting characteristics for which a trade-off must be made. For example, different distribution middleware products have varying influence on the availability and the performance of the system.

The selection task is critical because business information systems are *middleware-intensive* software systems, i.e. systems whose structure and behaviour are significantly influenced by the choice of middleware products. Therefore, a well-founded decision on using a middleware product should be made as *early* in the development process as possible: In later phases of the development project, significant investments into the design or implementation of the software system have already been made. If the decision conflicts with the existing design, substantial costs for revising and adapting the design to the middleware product may result. For this reason, an architecture-level selection of the middleware product is advisable. However, few guidelines or specific techniques exist on how to select a middleware product in a given project context, which leads to ad-hoc selection decisions [Ploski and Giesecke, 2005].

Most current development projects do not start from scratch but are confronted with a set of existing applications. These applications must be modified on the basis of middleware products that are already used, integrated with newly developed components using some middleware product, or the existing applications are to be migrated towards a new underlying middleware product. We assume that the middleware product itself already exists, i.e., we regard it as a COTS product even though it may actually be custom-made.

The selection of a middleware product must be complemented with the determination of a specific approach to using the middleware product consistently, as most middleware products allow different modes of use [Liu et al., 2006]: Different usages correspond to design alternatives that may influence the structure and behaviour of the application to the same extent as do different products. We refer to such a combination of a middleware product and its usage as a *middleware platform*. The choice a developer has to make is between a set of such middleware platforms, not between products on the first hand. For example, the applications using the Java Enterprise Edition product can choose to use message-based communication (Message-driven Beans) rather than call-and-return communication (RPC). Different usages should not be mixed arbitrarily to ensure the comprehensibility and maintainability of the system.

1.2 Solution Approach

To improve the selection of a middleware platform that is best-suited to achieve given quality requirements for the resulting system, we propose the MidArch Approach in this PhD thesis. It consists of the MidArch Design Method which uses MidArch Styles and a MidArch Taxonomy for the *systematic selection* of middleware platforms and for guiding the specification of software architectures. It enables an engineering approach to the selection of middleware.

Software architecture is defined by ISO/IEC Standard 42010 as:

> Software architecture is the "fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [ISO, 2006]

A software architecture in this sense is linked to a single system.

Common features of a family of related software architectures can be captured in architectural patterns or *architectural styles* (some authors [e.g. Jones, 1994] refer to these common features as "architectures" in opposition to our use of that term, which may be confusing). These artefacts specify constraints for component and connector types, and rules for the composition of components and connectors into an architecture [Shaw and Garlan, 1996]. Architectural styles are typically specified in a formal Architecture Description Language (ADL) such as Acme [Garlan et al., 2000], which allows the specification of both architectural styles and software system architectures. The formal specification allows to analyse specifications of software architectures for their conformance to an architectural style.

Middleware-oriented Architectural Styles (MidArch Styles) are a specialisation of architectural styles. They capture the structural constraints imposed

by a middleware platform. A MidArch Style characterises the family of software architectures that can be implemented on the associated middleware platform. It acts as a guidance for the process of modelling a software architecture for a specific system.

We first introduce MidArch Styles as a language-independent concept and then define mappings to the ADL Acme and to UML Component Diagrams. All known MidArch Styles are organised within a MidArch Taxonomy, which identifies refinement and other relationships among the MidArch Styles.

Within the MidArch Design Method, a MidArch Style Taxonomy is used as the basis for the stepwise selection of a suitable middleware platform. Software architectures are specified according to a MidArch Style and are evaluated against given quality requirements. The evaluation results are then stored as annotations to the related MidArch Style in the MidArch Repository. These annotations can be used in later middleware platform selections to improve the selection process.

For modern complex middleware products, it is non-trivial to extract the architectural rules that are necessary for an effective use of the middleware product, as various software development projects have shown [Garlan et al., 1995; Giesecke and Bornhold, 2006; Ploski and Giesecke, 2005]. Even when these rules are known, they must be applied consistently in the design of the architecture and in the actual implementation. The MidArch Approach ensures the consistent application of these rules within the architecture, and also supports their enforcement in the implementation.

Demarcation We are concerned with the choice among multiple middleware products that differ not only in their implementation, but in the structural constraints they impose on applications. If two middleware products impose the same constraints upon applications, they support the same middleware platforms and are thus considered equivalent in our approach, e.g. multiple Java Enterprise Edition Application Server products. The task of making a choice between multiple such products is addressed by middleware benchmarks.

1.3 Scientific Contribution

The PhD Thesis makes the following contributions to the scientific knowledge:

MidArch Style Modelling Approach The notion of architectural styles is refined by the definition of the MidArch Metamodel. While several ADLs exist that support modelling of architectural styles, the MidArch Style Modelling Approach focuses on the conceptual basis of middleware-oriented architectural styles independent from a concrete notation. The conceptual MidArch Metamodel can be mapped to existing notations, which is demonstrated in the thesis by providing mappings to the ADL

Acme and to the UML. By providing the UML mapping, we provide the first approach for modelling both architectural styles and style-based architectures in a widely used notation that also has the potential of integrating the component-and-connector viewpoint of architectural styles with other architectural viewpoints.

MidArch Design Method The MidArch Design Method is provided as a systematic method for the early selection of middleware in a development project, which enables an engineering approach to middleware selection. It makes use of the MidArch Style Modelling Approach.

The MidArch Design Method is the first approach that combines the work on middleware selection and on architectural styles by employing middleware-oriented architectural styles for modelling architectures and as a vehicle for storing architectural knowledge.

Evaluation The MidArch Design Method and the two mappings of the MidArch Style Modelling Approach (Acme and UML) are evaluated through case studies using real-world examples. While the case studies are performed in a lab setting, this is an improvement with respect to the state of knowledge, since the evaluation of prior work on middleware-oriented architectural styles used artificial examples only [e.g., Di Nitto and Rosenblum, 1999].

In addition, tool implementations for mapping style-based architectural descriptions to implementation concepts (ArchMapper) and for supporting the knowledge management around MidArch Styles (MidArch Repository) are provided to show that the method can be supported by tools which is a prerequisite for industrial use.

The contributions are explained in more detail when the research design and the single work packages are described in the following chapter (Chapter 4).

1.4 Overview

The PhD thesis consists of four parts and an appendix.

The first part is concerned with foundations that underlie the work done in the PhD project. We structured these foundations into two chapters, one dealing with software quality and software quality evaluation (Chapter 2), and one dealing with software design and software architecture (Chapter 3).

The second part presents the MidArch Approach that was developed within the PhD project. It begins with a description of the research design, which determines which research methods are used in the PhD project (Chapter 4). Then we list basic definitions and assumptions that underlie the approach (Chapter 5). These are necessary to mention explicitly, since no unanimously accepted definitions exist in the literature. Then, the taxonomy

of architectural style usages (Chapter 6) is presented, which formed the basis for developing a new usage of architectural styles. The approach taken for modelling middleware-oriented architectural styles is presented in Chapter 7. It is used as one component of the MidArch Design Method, which is the main contribution of the thesis (Chapter 8).

The third part is concerned with the evaluation of the MidArch Approach. It starts with an overview of opportunities for evaluating the approach (Chapter 9). The evaluation performed within the thesis was done using two case studies (Chapter 10), development of tools supporting the MidArch Design Method (Chapter 11) and discussion of related work (Chapter 12).

The fourth part contains the conclusions (Chapter 13) and a discussion of opportunities for future work (Chapter 14).

The appendix provides the definition of the MidArch Design Method (Appendix A).

Part I

Foundations

2 Software Quality and its Evaluation

*This chapter is partly based on a previous publication [*Giesecke*, 2006b].*

In this chapter, foundations required for understanding software quality and its evaluation are discussed.

This chapter is structured as follows: First, we give an introduction to software quality (Section 2.1). Then, basic terms regarding software and its quality are defined (Section 2.2). Software quality can be discussed with reference to different realms (business, system, architecture). The realms and the corresponding quality characteristics are discussed in Section 2.3. Quality models are used to model the entirety of relevant quality characteristics (Section 2.4). Finally, concepts and methods for software architecture evaluation are discussed in Section 2.5.

2.1 Introduction to Software Quality

Software quality has many facets, one of which is reliability: the ability of a software system to deliver correct service continuously. Software quality often refers to emergent properties. *Emergent properties* are such properties of a software system that cannot be derived from the component properties, but result from effects of interaction among the components. Different notions of "emergence" assume different degrees of non-derivability: These include a) properties that are not trivially (e.g. additively) derivable, but may be effectively calculated; b) properties that are practically infeasible to be derived due to inefficiency; c) properties for which no derivation procedure is known; d) properties for which it is known that it is theoretically impossible to derive. Some software system properties—even if they fall into the last category—may, however, be predicted in sufficient accuracy using compositional techniques, which is a major field of current research.

Historical Roots of Software Quality Dependability of mechanical devices, later of electronic devices has long been a major issue [cf. Avizienis et al., 2001]. Research on mechanical dependability was primarily focused on reliability, and was based on the "weakest link" approach: If some component failed, then the whole system failed. After World War II, modern reliability theory was developed which focused on electronic hardware. Early computer components were very unreliable—perhaps they would be considered unusable

by today's standards—, so much work focused on constructing more reliable systems based on less reliable components by using these redundantly. Only later the idea of software as an artifact that is of interest independently from the underlying hardware platform arose.

As the lack of software dependability became apparent, first attempts were started to transfer the ideas used to cope with hardware faults to the software domain. A major proponent of this early work was Brian Randell [Randell, 1975]. His work was later complemented with work on N-version programming [Avizienis and Chen, 1977]. However this approach later turned out to be ineffective to overcome software unreliability in general [cf. Rohr, 2005]. The reasons for this phenomenon can probably be found in the different nature of hardware and software, which is the focus of the next section.

Later, during the early 1990s, aspects of security gained more importance and were integrated into the conceptual framework of dependable computing, thereby also increasing interaction of the previously distinct scientific communities of dependability and security.

2.2 Basic Terms

In this section, we discuss basic terms regarding systems (Section 2.2.1), their behaviour (Section 2.2.2), and their quality properties (Section 2.2.3). The latter is based on the terminology used in the ISO standards that deal with software systems and their quality.

2.2.1 System

In the following, we will use the term *software-intensive system* for a software/hardware computing system, in which the influence of the software on the delivered service dominates that of the hardware. We will use the term *software system* to emphasise the software aspect, and the term *computer system* to emphasise the hardware aspect.

In ISO/IEC 9126-1 [2001], a distinction between system and software is used: Taking reliability as an example, the reliability of the system considers any failure, while reliability of the software considers only failures originating from faults located in the software. However, the user cannot observe where a problem originates, and it is not possible to ultimately decide on the source of a fault, so the distinction is not definitive. Additionally, software may be used to overcome deficiencies of the hardware (and the other way round), so the distinction is not helpful either.

2.2.2 Specification of Behaviour

We need to distinguish a system's *actual*, *specified*, and *expected* behaviour. Of particular relevance is the actual behaviour:

Definition 2.1 (service)**.** The *service* delivered by a system is the actual behaviour as perceived by its user(s), which may be a person or another system.

Avizienis et al. [2001] define the function of a system as the behaviour specified in its functional specification. A system delivers *correct service* if its service matches its function, i.e. the actual behaviour matches the specified behaviour. This reduces the service (and behaviour) to functional aspects, and matches the classical notion of correctness as used in theoretical computer science. Thus, a system whose service matches its functional specification but does not match the specified performance requirements, could still deliver correct service.

2.2.3 The ISO Conceptual Framework

The ISO/IEC 14598-1 [1999] standard distinguishes different levels of quality properties:

Definition 2.2 (characteristic, sub-characteristic)**.** A *characteristic* is a high-level quality property of a software system which is refined into a set of *sub-characteristics*, which are again refined into quality attributes.

Definition 2.3 (attribute, metric, measurement)**.** Quality *attributes* are detailed quality properties of a software system, that can be measured using a quality metric. A *metric* is a measurement scale combined with a fixed procedure describing how measurement is to be conducted. The application of a quality metric to a specific software-intensive system yields a *measurement*.

In the ISO/IEC 9126-1 [2001], a specific hierarchy of characteristics, sub-characteristics and attributes is defined, which claims comprehensive coverage of all relevant aspects of software quality. Above the level of characteristics, the standard distinguishes internal and external quality of software and quality in use (see also Section 2.3).

For each characteristic, the *level of performance* can be assessed by aggregating the measurements for the identified corresponding attributes. Here, "performance" does not refer to time efficiency, but to the accomplishment of the specified needs.

2.3 Realms of Software Quality

Bass et al. [1998, ch. 4] distinguish three *realms of software quality*: System quality, business quality and architectural quality. System quality is

again distinguished into the quality that may be discerned by observing the executing system and the quality which can only be discerned by looking into the static constituents of the system. Essentially the distinction into external and internal quality made by others [cf., e.g., ISO/IEC 9126-1, 2001; Ghezzi et al., 2002, ch. 2] corresponds to this distinction of system quality characteristics. Quality characteristics may be assigned to these three realms.

Quality characteristics of the architectural realm are discussed in Section 2.3.2, and those of the system quality realm are discussed in Section 2.3.1. Business quality characteristics are outside the focus of our work and are not discussed further.

System quality characteristics may be refined into those aspects that can be or are influenced by the architecture (architecture-variant) and those that cannot (architecture-invariant). Architectural aspects are not decoupled from the implementation, but must of course also be obeyed by the implementation (architectural conformance, see Section 11.2). Most system quality characteristics have both kinds of aspects.

Often, architecture-variant system characteristics and genuine architectural quality characteristics are not clearly distinguished [see, e.g., Richter, 2005].

Concerning the run-time characteristics, *performance* is both influenced by communicational and computational aspects. While computational aspects are local to one component and thus mostly architecture-invariant, communicational aspects are architecture-variant. Often, communication is more time-consuming than computation. The time consumption varies significantly for different architectural solutions. Thus, architecture-level consideration of performance is advisable, in particular but not only for distributed software systems. *Security* is very often addressed by introducing special components that isolate security functionality from the "main" functionality of the software system by shielding the communication and interaction of those components. Security solutions are thus mainly architecture-variant. When regarding *functionality* as a quality as well, one can find that functionality alone is essentially architecture-invariant. Under the unrealistic assumption that no (other) quality than function were relevant, a monolithic software system would be able to serve the same function than a componentised software system. *Usability* needs to be enabled by architectural means, but is mainly architecture-invariant, since the detailed (visual) design of user interfaces is local to software components.

Concerning the static characteristics, *modifiability/changeability* is largely an architectural issue. Based on a specific architecture, modifications can be classified into three categories: those that are local to one component, those that concern multiple components, and those that require the whole architecture to be reevaluated and possibly changed, e.g. changes to the architectural style. *Portability* is usually achieved at an architectural level by introducing a portability layer. However, portability is a more complex issue in distributed, heterogeneous software systems which are in the focus of our

work. *Reusability* is the flip-side of modifiability: "Building systems to be modifiable yields the gains of reusability" [Bass et al., 2003]. *Interoperability* within the system boundaries (integrability) is a major concern in our work and is largely architectural, since it concerns external properties of the components. Interoperability across the system boundary is another issue.

Losavio et al. [2003] discuss the architecture-variance of the ISO 9126 system quality characteristics and sub-characteristics and propose metrics for evaluating the sub-characteristics on the architectural level. Bratthall and Wohlin [2000] discuss how system quality characteristics can be expressed as part of architectural descriptions.

2.3.1 System Quality Characteristics

ISO/IEC 9126-1 [2001] defines six characteristics along with several sub-characteristics:

Functionality "A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs."
- Suitability
- Accuracy
- Interoperability
- Conformance
- Security

Reliability "A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time."
- Maturity
- Recoverability
- Fault Tolerance

Usability "A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users."
- Learnability
- Understandability
- Operability

Efficiency "A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions."
- Time Behaviour
- Resource Behaviour

Maintainability "A set of attributes that bear on the effort needed to make specified modifications."
- Stability
- Analysability

- Changeability
- Testability

Portability "A set of attributes that bear on the ability of software to be transferred from one environment to another."

- Installability
- Replaceability
- Adaptability

In addition, for each characteristic the sub-characteristic "conformance" is defined, which refers to conformance of the characteristic to industry standards or legal requirements.

Quality of Service The TrustSoft framework [Hasselbring and Reussner, 2006] views the three characteristics availability, reliability and performance jointly as *Quality of Service*. They have in common that they can be observed by the user of a service and several applicable metrics are available, which operate on a ratio level scale. However, their prediction before the software system has been built remains a major research issue.

2.3.2 Architectural Quality Characteristics

Architectural quality characteristics are of major importance in our work. In the same way system quality is distinguished into internal and external quality, also architectural quality can be distinguished into internal architectural quality [cf. Teeuw and van den Berg, 1997] and external architectural quality [cf. Biemans et al., 2001].

Conceptual integrity of an architecture is considered to be the central consideration in (architecture-level) design by Brooks [1995]. Architectural style (see Section 3.4.2) is a concept specifically addressing conceptual integrity of a software architecture. Other architectural quality characteristics are *correctness* and *completeness* with respect to the stated requirements (i.e., validity of the architecture). Finally, Bass et al. [1998] list buildability, perhaps better termed feasibility[1], but it is not simply restricted to cost issues[2]. An important aspect of buildability is the understanding of the concepts used in the solution, which is also addressed by providing rigorous architectural styles.

The use of architectural constraints in architectural development and description is an important contribution to achieving architectural quality [Dueñas et al., 1998]. Adequate properties of the software architecture enable that the resulting software system meets the requirements with respect to

[1] "Buildability" is not a good term in our opinion since it is apt to confusion with a characteristics related to build management.

[2] Ultimately, any deficiencies in buildability may be solved by increasing the investment, but not without enhancing the project scope, e.g. by building project-specific development tools, execution platforms, or hardware.

behaviour, quality and life cycle [Bass et al., 2003]. The use of architectural constraints contributes to all these aspects of architectural quality.

The support for architectural quality concerning the development life cycle of a software system is central to this contribution. During the analysis and design phase a software architecture helps to make appropriate decisions and to reduce risks related to a specific set of requirements. Architectural constraints enhance these effects by providing well-established solutions. Furthermore, they reduce the complexity of a solution by improving the *conceptual integrity* and *consistency* of an architecture [Brooks, 1995, p. 95]. With respect to validation and verification, architectural quality encompasses both the internal consistency of the architectural description *(architectural consistency)* and the external conformance of a software system's implementation with its architectural description *(architectural conformance)*. The use of architectural constraints contributes to these aspects of architectural quality as well. It reduces the effort necessary to create and maintain an architectural description, and to maintain architectural conformance during the software life cycle.

Architectural conformance is attained through *traceability*. Traceability includes *requirements traceability*, i.e. the possibility to relate software artefacts (at any level) to the requirements they contribute to, as well as *architectural traceability*, i.e. the possibility to relate implementation artefacts to architectural artefacts.

A proper level of architectural conformance is a prerequisite for useful architecture-based predictions of system quality attributes [Jung et al., 2004], concerning the internal and external software quality. Tool support for these activities is desirable.

2.4 Software Quality Models

A software quality model specifies a set of software metrics. It defines *what* should be measured and *how* this should be done, but it comprises neither measurements of a specific software process or artefact nor the specification of required levels of performance for each metric. It is thus a class model of measurements, but not an instance model. It is typically used as a normative, prescriptive model, which guides a measurement process. A software quality model is *applied* to a software process or artefact to yield measurements. A software quality model may also be used for the specification of levels of performance for each metric, i.e. a quality model based requirements specification.

An alternate term for a software quality model is *software measurement model*. A software quality model is essentially a special case of a quality evaluation model, which again is a special case of a *measurement model*. However, we restrict the further discussion to models referring to software

process and artefacts.

We consider two dimensions of software quality models: One internal dimension which concerns the structure of the quality model, and one external dimension which concerns the application domain of the quality model.

Regarding the *structure*, a software quality model may be *flat* (i.e., the quality model is simply an unstructured set of metrics) or *hierarchical*. Hierarchical software quality models may be *goal-oriented*, when the hierarchy is organised around goals.

Regarding the *application domain*, we distinguish *universal* quality models and *context-specific* quality models. Universal quality models claim applicability to any setting, while context-specific quality models are designed within a specific context and do not claim applicability beyond that context. A hybrid approach is conceivable, where a context-specific overall quality model is composed from universal fragments.

While there are several variants of goal-oriented, context-specific quality models [see, e.g., Cantone and Donzelli, 1999], we focus on the established goal/question/metric paradigm.

The Goal/Question/Metric Paradigm The goal/question/metric paradigm (GQM) [Basili et al., 1994; van Solingen and Berghout, 1999] is an approach to defining hierarchical, context-specific software quality models, which consist of three fixed levels: goals, questions, and metrics. The goal/question/metric paradigm not only defines the structure of software quality models, but also a method for their definition.

As with all goal-oriented approaches to defining quality models, the goal/question/metric paradigm was conceived to overcome unfocused data collection efforts in software development, which were conceived as overly costly and unproductive since a large number of metrics were applied without a systematic and shared understanding on how the collected data could be interpreted. While the goal/question/metric paradigm does not provide the means to objectively interpret measurements, it provides guidance for the interpretation process. Through the way the quality model is defined, which involves all stakeholders in the software process, the intersubjectivity of the quality model and its interpretation is assured.

GQM goals are typically software process improvement goals. While the goal/question/metric paradigm was developed to allow more systematic evaluation of software processes and artefacts, it can be applied to any measurement context. In fact, it has been used extensively in scientific experimental settings in empirical software engineering.

2.5 Software Architecture Evaluation: Goals and Methods

The terms "evaluation", "analysis" and "assessment" are often mixed up. We follow the terminology of Kruchten [2005] translated to software architectures: The generic term "assessment" is not used as a technical term. It must be noted that in software engineering, "analysis" is sometimes used as a shorthand for "requirements analysis", which has a completely different meaning. "Evaluation" in general is the assessment of the gap between expected properties and actual properties.

Then, *software architecture evaluation* is the assessment of the gap between the specified requirements and the actual properties of a software architecture. A *software architecture evaluation method* is a specific procedure to perform such an assessment. One class of software architecture evaluation methods that can be used within the MidArch Design Method are scenario-based software architecture evaluation methods, which are briefly presented in Section 2.5. A quality model may be used within a software architecture evaluation as a prescriptive means which guides measurement.

Software architecture analysis, on the other hand, refers to techniques and tools that are used to derive properties from a software architecture description. Software architecture analysis is used within software architecture evaluation to obtain data for the final assessment.

Software architecture analysis techniques and tools are specific to certain quality characteristics. A software architecture evaluation method, on the other hand, is not specific to a certain set of quality characteristics. However, some specialised methods may be more suitable for software systems that have specific requirements for quality characteristics, e.g. safety-critical software systems. In addition, a clear distinction between architecture evaluation and architecture analysis is not always made, so that there are architecture analysis techniques that include the evaluation aspect.

A general problem of software architecture analysis is that many design decisions are still left open by the software architecture, so a large variety of implementations may correspond to a single architectural description. The quality characteristics of these software systems may also vary significantly. This problem cannot be overcome by any software architecture analysis technique, so one must accept the large degree of uncertainty in architecture-based predictions. Exceptions to this general property exist only with respect to genuine architectural quality characteristics, the foremost of which is flexibility. Flexibility is an important quality characteristic in long-living software systems in contexts with fluctuating requirements. This typically applies to business information systems, where the requirements depend on a large number of stakeholders which are remote to the development team.

Scenario-based Software Architecture Evaluation Methods Many of the well-known scenario-based software architecture evaluation methods have

been developed at the Software Engineering Institute of Carnegie Mellon University:

- The Scenario-based Architecture Analysis Method (SAAM) [Kazman et al., 1994] can be applied to the analysis of several quality characteristics, while a certain focus on modifiability or architectural flexibility exists. Modifiability is determined based on change scenarios. Other types of scenarios include "use case" scenarios, for which the satisfiability is determined. The scenarios are elicited, prioritised and evaluated by the relevant stakeholders. The subjectivity of the evaluation is controlled by a tight involvement of the stakeholders in the overall process.

- The Architecture Trade-off Analysis Method (ATAM) [Kazman et al., 2000] is based on the SAAM and has a more comprehensive approach to the evaluation of software architectures. While the SAAM focuses on a single architectural characteristic, the ATAM considers all relevant quality characteristics and classifies architectural design decisions based on their relationship to the architecture-level quality characteristics. Classification dimensions include the *risk* of the decision and the number of quality characteristics (one or more) that are affected by the decision. A decision is considered risky if the effect of the decision on the software system's quality cannot be determined with confidence.

- The Cost Benefit Analysis Method (CBAM) [Kazman et al., 2001, 2002; Moore et al., 2003] adds an economic dimension to the ATAM by also covering cost. The benefit is determined through suitable elements of the ATAM, and cost predictions are made using existing software cost prediction methods.

- The Architecture-level Modifiability Analysis (ALMA) [Bengtsson et al., 2004] is focused only on modifiability and does consider change scenarios only.

For a more detailed discussion of scenario-based software architecture evaluation methods, refer to Babar and Gorton [2004]; Dobrica and Niemelä [2002]; Grunske [2006].

In the sense of the above definition, the ATAM and CBAM can be considered genuine architecture evaluation methods, while the other presented methods, SAAM and ALMA, have more of the character of architecture analysis techniques.

3 Software Design & Architecture

In this chapter, we present foundations regarding software design and architecture. We will first concentrate on basic definitions of software architecture and its general characteristics in Section 3.1. At this point, we will also discuss commonalities and differences of software design and software architecture. We do not attempt an exhaustive discussion of this issue, and will only refer to "software architecture" for the most part of the remainder of the thesis. Next, we discuss general concepts that are used in architectural description (Section 3.2), which are abstract in the sense that they outline a framework for specific architectural *views*. Viewpoint and view are core notions introduced in this section. One architectural view is the component-and-connector view which is discussed in more detail in Section 3.3. It receives the greatest attention of software architecture research, and also is the focus of architecture descriptions that are the subject of this thesis. Following that, we will discuss architectural design constraints, which include architectural design patterns, architectural styles and reference architectures (Section 3.4). Notations that define a concrete syntax for describing software architectures, i.e. Architecture Description Languages, are described in Section 3.5.

3.1 Basic Definitions

Software architecture as a discipline within Software Engineering has been emerging since the seminal paper on "Foundations for the study of software architecture" by Perry and Wolf [1992] and the book by Shaw and Garlan [1996]. However, the term "software architecture" is much older, tracing back to Sharp [Randell and Buxton, 1970, p. 12] in the 1960s [Mahoney, 2004].

The software engineering literature refers to a variety of definitions of software architecture. Several of these definitions may be viewed as mutually incompatible, and concrete architectures modelled with reference to diverse definitions of software architecture will probably be incommensurable. Therefore, we decided to select the definition of software architecture in the Recommended Practice for Architectural Description of Software-Intensive Systems [ISO, 2006; Maier et al., 2001] as an approved reference definition:

> Software Architecture: "The fundamental organization of a system embodied in its components, their relationships to each

other, and to the environment, and the principles guiding its design and evolution." [ISO, 2006]

3.1.1 Commonalities and Differences of Software Design & Software Architecture

Software design and architecture both refer to models of a software system. Thus, they need to be distinguished from the software system itself (reduction feature [Stachowiak, 1973]). On the other hand, they are sometimes distinguished from each other. Both distinctions cannot be clearly made in a general way yet, which is one of the challenges for the maturation of the software engineering discipline. In the literature, terms such as design, architecture, system, and implementation are mapped in diverse ways to the concepts mentioned here. An exhaustive discussion of this terminological problem cannot be given within this thesis. However, we want to make our view of these terms and concepts clear insofar it is required for this thesis.

We use the following terminological convention: A *software system* has a *architecture description* and an *implementation*[1]. The implementation is the set of source code items and similar artefacts as well as the executable artefacts that are generated from them former. The implementation may or may not conform with the architecture described in the architecture description.

3.1.1.1 Common Properties of Software Design & Software Architecture

Executability Descriptions of architecture and design of a system are not intended to be executable, while the system itself must be executable. Some authors take the view that the design is a complete declarative and executable description of the system, and the actual implementation is merely performed for optimisation purposes. Similarly, model-driven development approaches focus on a series of transformations that add information to a design-like "model" until an implementation-like "model" is reached. It is questionable whether these models are actually models of the software system in the sense of general model theory, or merely incomplete precursors of the system itself.

Distinctness from System Implementation In our view, software design including architectural design is an activity that is distinct from implementing the system. Thus, the derivation of software design representations from the system implementation itself is not possible: for example, a UML class diagram that is generated from the source code by a tool is not a design document, unless it is manually checked that it reflects the design intent.

[1] The term "implementation" does not occur in the ISO Standard 42010, where the system is identified with its implementation.

It may be used as an aid for creating the design documentation, but by itself it is only a visualisation of the source code. In terms of general model theory, the automatically generated diagram misses the objective feature [Stachowiak, 1973], which can only be warranted by a dedicated design activity.

3.1.1.2 Differences between Software Design and Software Architecture

Differences between software design and software architecture are difficult to define in general. In the literature, sometimes the terms are used interchangeably, but if they are distinguished, software architecture is usually considered to be on a more abstract level than software design. Software design is then often referred to as *detailed design* to emphasise this fact. The conceptual distance of software architecture, software design and the system varies significantly within the literature. It appears that in software development practise, documentation that is conceptually far apart from the system can seldom be found.

In the software architecture literature, often the point is made that there are design decisions that are deliberately left open, because they are too technically detailed and it would impair the flexibility. The object of such a decision is thereby deemed an implementation issue, and it is distinguished from architectural issues [see, e.g., Mehta et al., 2000a]. However, it cannot be said in general (or at least no consensus has been found) which kind of issues belong to which category.

Middleware selection Within this thesis, the most important question of this kind is whether choosing a middleware is an architectural or an implementation issue. Depending of the context, both answers can make sense. Regarding the middleware as an implementation issue warrants the flexibility in this respect: No dependence on a particular supplier of a middleware is introduced at the architectural level. A suitable middleware product can be chosen during implementation when sufficient design detail is known to make an appraisal of quality of service characteristics.

On the other hand, making middleware selection an architectural issue is an obvious choice when some middleware is required to be used due to higher-level decisions, e.g., an organisational strategy requiring an organisation wide use of a particular middleware. We focus on middleware-intensive systems which have the defining property that the middleware significantly influences the structure of the system. Ignoring this property on the architectural level incurs the risk that the architecture does not fit well to the implementation concepts that are offered by the middleware. Thus, it would move the burden of adapting the architecture to the middleware to the architecture-to-implementation mapping (cf. also Section 11.2).

By making a distinction between "middleware platform" and "middleware product" (see Section 5.2.3) and choosing only the platform on the architectural level, an additional argument for making middleware platform selection an architectural issue is presented.

Flexibility In fact, *flexibility* is a central quality characteristic of the overall system that is thought to be supported by using an explicit architectural level of system description. This fact is also central in scenario-based architecture evaluation methods (see Section 2.5). This line of thought inherently conflicts with methods that try to *predict* properties of the resulting system based on an architectural description. In the first view, a software architecture description is more a tool for project management and a framework for making further decisions, while in the second view it is a technical document that has a close formal link to the resulting system. These views lead to conflicting requirements for the form and content of architectural descriptions. The focus of the first view is on internal properties (such as maintainability) of the system, while the focus of the second view is on external properties (such as quality of service) of the system (cf. Section 2.3).

In the development of business information systems, flexibility is often a requirement of such a great importance that using the more technical view of software architecture is beyond debate. Requirements on the system often change during all development phases and ignoring this fact would possibly lead to a system that is of better quality with respect to the explicitly stated requirements, but of less quality-in-use, as perceived by the users, and often a system that is unusable in practise. Making the flexibility manageable is thus of prime importance.

3.1.2 Scope of Software Architecture

The (outer) scope of a software architecture has two aspects: The first aspect is *vertical* and concerns whether *only software* or a *software-intensive* system that also includes the underlying hardware is modelled. The second aspect is *horizontal* and concerns the coordination of the development of the system: The system can either comprise only a *centrally coordinated* development and evolution effort or a system of systems that are developed and evolve *separately*.

Both aspects are not original properties of the system but they are design decisions of the system modeller, at least to some degree. This certainly applies to the first aspect, since any software system will finally be run on a hardware platform. In many cases, it also applies to the second aspect, except in cases where there is only one software system that is not contained in a larger context. However, most larger organisations maintain complex *application landscapes* ("Anwendungslandschaft" [Humm et al., 2006]), whether or not they are explicitly managed and modelled as such.

The scope of the software architecture is determined by the *system boundary*[2] that is chosen by the modeller. The system boundary demarcates the border between the system and the outside world, or the *environment* of the system. The system's internal structure and the relationships between its elements are modelled in detail, while the environment is only captured to the extent needed to describe its interactions with the system.

On a larger scale, software-intensive systems are considered socio-technical systems [Bostrom and Heinen, 1977], when the users and their sociological interactions are considered part of the system itself.

Vertical Scope A software system ultimately needs a hardware platform on which it is deployed and executed. However, it does not need to be deployed directly on a physical hardware platform. On the one hand, it may be deployed on a virtual machine, which resembles a physical hardware platform from the point of view of the software system. On the other hand, additional software layers that provide abstractions from the hardware platform can be situated between the subject software system and the hardware platform, which include middleware (see Section 5.2) and operating systems. It usually makes sense to generalise the notion of a hardware platform to that of a *deployment platform*. A software architecture then models some software layers which are considered inside the system boundary and considered the remaining layers as the environment, which is only modelled through some deployment mapping.

While hardware is different from software in that it is material rather than immaterial, from the point of view of the software system that is deployed on either type of platform, these platform share an important characteristic: Their internal structure is not considered relevant to the deployed system. It is only necessary to describe externally exposed properties of the platform. The exact list of relevant properties differs for different types of software and hardware platforms. A more rigorous definition of "platform" will be discussed in Section 5.2.3.

Horizontal Scope The distinction between software architectures that concern a single system and a system of systems corresponds to the distinction between the tactical focus within some defined development project and the strategic focus on some organisational level that transcends project limits.

3.1.2.1 Abstract and Concrete Platforms

Almeida et al. [2004] define the notion of an *abstract platform* in the context of the MDA approach. They refine the notion of a "generic platform", which

[2] Note that "system boundary" is used with a more restricted meaning in some modelling approaches, particularly within UML use case diagrams.

is mentioned in the MDA Guide [OMG, 2001], but not rigorously refined. The relationship of abstract and concrete platforms parallels the relationship of platform-independent (PIMs) and platform-specific models (PSMs): The platform-independent model relies on an abstract platform, which represents certain common characteristics of the concrete platforms that refine the abstract platform. To transform a PIM into a PSM, a transformation representing this refinement is chosen and applied. So, typically an abstract platform is refined by several alternative concrete platforms. Is it not generally possible to apply an arbitrary transformation to any PIM. A PIM-to-PSM transformation depends on both the abstract and the concrete platform. Almeida et al. define:

Definition 3.1 (Abstract platform)**.** "An abstract platform defines an acceptable or, to some extent, ideal platform from an application developer's point of view; it represents the support, as comprehensive and direct as possible, that is assumed by platform-independent models of a distributed application." [Almeida et al., 2004]

An indirect characterisation of concrete platforms is also given:

Definition 3.2 (Concrete platform)**.** "An abstract platform defines characteristics that must be mappable onto the set of concrete platforms that are considered as potential targets in a development project." [Almeida et al., 2004]

In a typical MDA approach, neither the abstract nor the concrete platforms are explicitly modelled. Almeida et al. propose a specific MDA approach which uses explicit models on abstract and concrete platforms.

3.2 General Concepts of Architectural Description

In this section, we discuss basic concepts for the description of software architectures. The basis for the presentation is the ANSI/IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, which is also an ISO Standard (ISO/IEC 42010:2007 [ISO, 2006]). Its scope are architectural descriptions, not the systems that are described (or their architecture), the development projects creating the systems, or the processes used within the projects or the organisations conducting software development. Only an architectural description can thus be said to comply or not to comply with the standard.

However, two points must be noted in this respect: First, the standard is fairly general and provides basic conceptual foundations rather than immediate practical guidance. For example, the concept "viewpoint" is defined in the standard, but no specific viewpoints. Second, the standard does not immediately reflect current architectural description practise. For

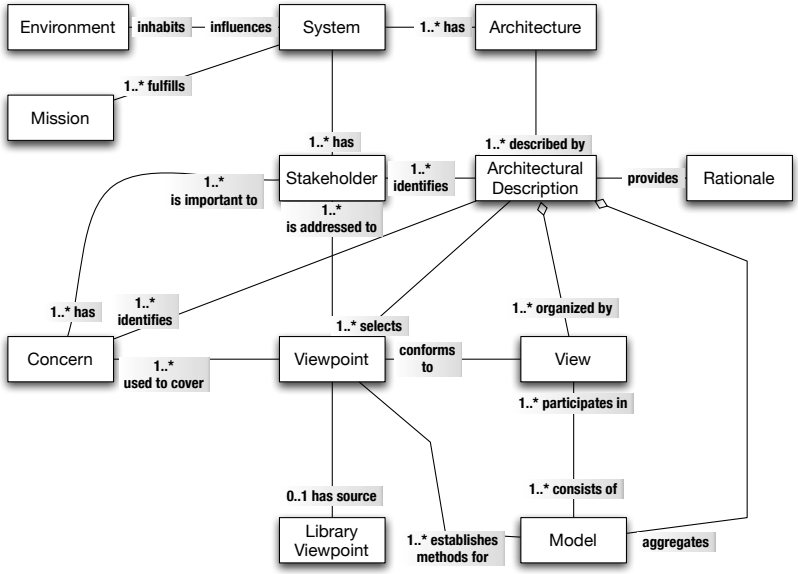

Figure 3.1: Conceptual metamodel for architectural description (after ISO [2006])

example, the distinction of viewpoints and views is quite specific to the standard. Thus, different degrees of compliance of terminology with the terminology of the standard may be observed in practise.

The standard defines a conceptual metamodel for architectural description. An overview of this metamodel is shown in Figure 3.1. However, the standard itself notes that this model is not authoritative with respect to the relationships of the represented concepts, but should merely serve as an illustration of the standard's intentions. Most importantly, the cardinalities of the associations are not entirely consistent because they assume that only a single architectural description is considered.

The two most fundamental distinctions made in the conceptual metamodel are the following: First, the (software) *system* (or the implementation) is distinct from its architecture. Second, the *architecture* of a system is distinguished from its description (*software architecture description*). *Software architecture description* is both the process and the result of explicitly modelling a software architecture using suitable notations, most importantly Architecture Description Languages (ADLs). The architecture itself is intangible, it is changed only indirectly through changes to the system. Every

system has an architecture, but it is not necessarily documented explicitly in an software architecture description. If there is a software architecture description, it may not be complete, it may be outdated or it may be simply incorrect with respect to the actual architecture of the system.

However, for complex software systems it is advisable to maintain an explicit software architecture description which is kept conforming with the actual architecture, i.e. when the actual architecture and the documented architecture deviate, conformance is reestablished as part of the software development process. The software architecture description can then be used for conformance analyses throughout the lifecycle of the system, for example [Sefika et al., 1996]. Modifications to the software architecture description (resp. the system) may be regarded as a trigger to adapting the system (resp. the software architecture description) such that conformance of the software architecture description and the actual architecture is reestablished.

Viewpoint A *viewpoint* provides principles for representing related concerns that are determined by stakeholder interests. A viewpoint also determines modelling techniques and notations to use in views that correspond to the viewpoint. A *view* describes a (concrete) system from a certain (abstract) viewpoint. Views structure the overall *software architecture description*. The term "view" should not be confused with "visualisation". The concept of views is only a means to structure the information in an abstract manner, and a view has not an immediate material representation[3]. The actual information is contained in *models* that are related to views in a $m : n$ relationship: A view refers to information that may be distributed over multiple models, while a model may contain information that attributes to multiple views (which is necessary to relate multiple views to each other).

A viewpoint has a name and is related to stakeholders it addresses and specific concerns of these stakeholders. Furthermore, the viewpoint specifies modelling techniques and languages to be used for views that comply with the viewpoint. In addition, it may specify analyses that can be applied to the models within views that comply with the viewpoint, e.g., consistency and completeness checks or performance analyses.

The standard specifies that guidelines aiding modelling views that comply with the viewpoint may be specified as part of the viewpoint (cf. Section 5.7 for a more refined discussion of this point).

3.2.1 Application and Technology Domains

The term "domain" is often used for classifying systems with similar properties, i.e. similar systems are said to belong to the same domain. We see

[3] E.g., Medvidovic and Taylor [2000] use the term "view" with the meaning of "visualisation".

the need to distinguish two kinds of properties in this context. We thus use two more specific terms, which we will handle in very different ways in our conceptual framework: The application domain of a system and the technology domain of a system. The term "application domain" is often used as a substitute for "domain", but even then technology domains are subsumed under the term, which may be even more confusing. A system may belong to multiple application and technology domains.

Application Domain The application domain denotes the subject area of the problem the system ought to solve, i.e. the application domain is a property of the problem space and comes up during requirements engineering. Systems from the same application domain are similar in that they aim to solve related problems. Application domains can be considered at different levels of granularity, and can be specific to a single project, organisation, or they might to a whole industrial sector. Examples of application domains are systems for e-commerce, accounting systems or, as a very general application domain, business information systems. The specific fields of domain engineering or domain modelling are concerned with application domains.

Technology Domain The technology domain of a system denotes the technological basis used to implement the system, i.e. the technology domain is a property of the solution space. While specific requirements may specify or constrain the technology domains that may be used, the technology domain primarily comes into play during the definition of the software architecture, after requirements have been formulated. Examples of technology domains include web-based information systems (very general) or Cocoon-based web information systems (specific to a particular product). In our conceptual framework, the technology domains are reflected by the architectural layers below the application layer and are represented by abstract platforms. A significant difference between technology domains and abstract platforms is that while a system may belong to an unstructured set of technology domains, abstract platforms are considered to be layered. Choosing the abstract platforms that underlie a system's application-level architecture thus requires more refined design decisions than simply enumerating technology domains.

Distinction Some topics can be both an application domain as well as a technology domain. For example, in the case of software engineering tools, the application domain can be a software technology, but it plays a different role than the technology domain of that system: A tool for developing Java EE applications (Java EE as application domain) does not need to be implemented as a Java EE application (Java EE is not the technology domain).

3.2.2 Typical Architectural Viewpoints

In principle, viewpoints may be individually defined for a single development project [Koning and van Vliet, 2006]. But stakeholders and their concerns are shared among multiple projects, even across organisations, so it is economical to reuse viewpoint definitions. The ISO Standard 42010 indeed mentions such viewpoints, and refers to them as *library viewpoints*.

Viewpoints are seldom defined in isolation, but a set of viewpoints must be coordinated to ensure that different viewpoints do not overlap or conflict. Largely independent from the ISO Standard 42010, several sets of viewpoints have been defined[4].

Well-known viewpoint sets are defined by [Clements et al., 2002; Hofmeister et al., 2000; Kruchten, 1995]. The terminology differs between these viewpoint sets, but roughly the following common viewpoints can be identified:

Module Viewpoint The Module Viewpoint describes the system from the point of view of the software developers who need to maintain an overview of the source code structure at the time of programming/development. It is also known as the development view [Kruchten, 1995] or the source code view [Hofmeister et al., 2000].

Component-and-Connector Viewpoint The Component-and-Connector Viewpoint describes the system in terms of the logical structure of components and connectors at deployment time and the basic structure during runtime. It establishes an "allocation of function to structure" [Kazman et al., 1994]. It is also known as the structural view, logical view [Kruchten, 1995] or module view[5] [Hofmeister et al., 2000].

Process Viewpoint The Process Viewpoint is an alternate view of the software system which focuses on issues of concurrency. Thus, it is not relevant for systems that operate essentially sequentially. However, for highly concurrent systems it may be a critical viewpoint. It is also known as thread architecture [Sandén, 2003] or execution viewpoint [Hofmeister et al., 2000] (while the latter also includes aspects of the Allocation Viewpoint).

Allocation Viewpoint The Allocation Viewpoint describes the allocation of the elements of one of the former viewpoints, most importantly the Component-and-Connector viewpoint to elements of the execution platform, particularly hardware components. It also includes configuration

[4] While the authors do not refer to them as viewpoints and do not make the distinction between viewpoints and views, they may be considered viewpoint definitions.

[5] Despite the term "module view", it refers to the component structure.

information for the execution platforms. It is also known as the deployment viewpoint, physical viewpoint [Kruchten, 1995], or distribution viewpoint.

The term "dynamic viewpoint" or "behavioural viewpoint" is problematic, as it is unclear what they exactly mean. It may refer to both dynamic changes within the structures described by the component-and-connector viewpoint or to the interactions of the components and connectors. In both cases, it is tightly bound to the component-and-connector viewpoint and can thus be considered to be part of extended information within that viewpoint (cf. also Sections 3.2.3 and 3.3.3).

In addition to these viewpoints, a *data viewpoint* [Hilliard, 1999] is often used, which contains conceptual and logical data models that are used within the system.

3.2.3 Architectural Decompositions

A conceptual framework that is not entirely in line with that of the ISO Standard 42010 is described by Ran [Jazayeri et al., 2000; Ran, 1998, 1999]. It builds upon the assumption that a software architecture is described by several *architectural decompositions* which are associated additional *views*. An architectural decomposition is based on a *component domain*[6]. Each of the component domains allows to decompose of a system into a graph/diagram of computational elements. Examples such as the run-time domain (where computational elements are threads), the structural domain, or architectural structure domain (where computational elements are components), and the deployment domain (where computational elements are the units of deployment).

In an attempt to map this framework to the ISO Standard 42010, one could say that both architectural decompositions and views represent viewpoints. However, these are necessarily distinguished into primary and secondary viewpoints: An architectural decomposition is defined through a primary viewpoint, which has its additional views corresponding to secondary viewpoints.

However, this does not reflect the assumption that each architectural decomposition corresponds to a set of concerns, while the views are merely structuring information within one set of concerns (see below). The idea of general architectural decompositions historically evolved from the observation of the need to distinguish the modular structure of a software system from its runtime structure. We discuss this specific distinction in more detail, as it is important for understanding today's approaches to software architecture.

[6] Despite the terminological proximity to the terms "application domain" and "technology domain" introduced above, the concept is not directly related to them.

Modular Structure vs. Runtime Structure Weinberg [1998] states, in an overinterpretation of Conway's Law [Conway, 1968], that the structure of a software system is determined by the organisational structure of the developing organisation. In our view, this notion is misleading for two reasons: First, like virtually any social phenomenon, it is not a law of nature, but—if true at all—merely a tendency that could probably be diverted from if specifically addressed. Second, it is based on a view of software, which puts great emphasis on the *modular* structure of software, i.e. the structure of the source code. This view is in conflict with architecture-centric approaches to software development, which places the component structure before the code structure. One must keep in mind that when Weinberg originally wrote his book in 1973, software architecture was far from becoming prominent (in today's sense) and the complexity of software systems was much smaller than of today's software systems. This view of software, despite the introduction of the notion of software architecture, nonetheless represents widespread practise today [see, e.g., Bass et al., 1998, ch. 4.2]: It is the view that a single decomposition of a software system is sufficient to adequately describe it. On a different level, this view is reflected in the discussion of the nature of software components, concerning whether they are more like code modules or more like runtime processes [Cheesman and Daniels, 2000; Szyperski, 2002].

Some multi-viewpoint models, like the 4+1 View Model of Software Architecture Kruchten [1995], acknowledge that multiple views on a software architecture are necessary, but still refer to one primary decomposition of the system, which is the static view or decomposition. Implicitly, there are other decompositions, for example for the allocation to hardware elements, but these are considered subordinate to the primary decomposition. The ISO Standard 42010 discussed in Section 3.2 is indifferent to this issue, since it does not prescribe to use specific views, viewpoints or models.

The practise noted above thus derives the primary structure from organisational issues, interpret this as the static modular structure, and further technical issues get subordinate to these, i.e. run-time and non-run-time system characteristics (see Section 2.3). This is considered bad practise in modern software architecture. For example, Bass et al. [1998, p. 89] state:

> "It is an axiom of this book that assuming that the two structures [i.e., static and run-time] are the same is a fundamental design mistake, since they are optimized to meet completely different criteria."

Of course, with the acknowledgement of multiple equally important decompositions of a software system the problem of mapping their elements onto each other gets more complex.

3.3 The Component-and-Connector Viewpoint

Some authors equate software architecture implicitly or explicitly with the component-and-connector viewpoint [e.g., Kazman et al., 1994]. As a result, the component-and-connector viewpoint is the viewpoint that receives most attention in software architecture research. This is due to the fact that the views that correspond to some variant of this viewpoint are referred to by many other viewpoints. Thus, such a view needs to be included in virtually any software architecture description, even if it is not always the most important viewpoint (for example, for highly concurrent systems a variant of the runtime or process viewpoint may be even more important). The only viewpoint that is still more obvious to consider is a variant of the module or code structure viewpoint. However, in many software systems, this viewpoint is so straightforward that it does not need to be extensively documented. Alas, ADLs historically evolved from Module Interconnection Languages (MILs). Implicitly this evolution represents a change of the viewpoint that can make use of the language, from the module viewpoint to the component-and-connector viewpoint. Most ADLs today focus on the component-and-connector viewpoint. According to Medvidovic and Taylor [2000], it is a defining characteristic of ADLs to provide distinct modelling constructs for representing both components and connectors. However, in the software architecture research community, the question whether connectors deserve first-class status (rather than being consider secondary to components) is subject to an intense debate with strong proponents on both sides. For this thesis, we concur with those proponents that see a first-class status of connectors. The arguments in this discussion are summarised in Section 3.3.2.

A further topic of this section are the particular problems of modelling component-and-connector views in structurally dynamic systems (Section 3.3.3).

3.3.1 Definition

A characterisation of software architecture restricted to the component-and-connector viewpoint is given by Moriconi and Qian [1994]:

> "A software architecture is represented using the following concepts:
>
> 1. Component: An object with independent existence, e.g., a module, process, procedure, or variable.
> 2. Interface: A typed object that is a logical point of interaction between a component and its environment.
> 3. Connector: A typed object relating interface points, components, or both.

4. Configuration: A collection of constraints that wire objects into a specific architecture.
5. Mapping: A relation between the vocabularies and the formulae of an abstract and a concrete architecture. The formula mapping is required because the two architectures can be written in different styles.
6. Architectural style: A style consists of a vocabulary of design elements, a set of well-formedness constraints that must be satisfied by any architecture written in the style, and a semantic interpretation of the connectors.

Components, interfaces, and connectors are treated as first-class objects, i.e., they have a name and they are refineable. Abstract architectural objects can be decomposed, aggregated, or eliminated in a concrete architecture." [Moriconi and Qian, 1994]

There are many definitions that focus on components and connectors as two distinct first-class entities. The definition cited above is among them and additionally covers architectural style, which is the most important entity in our research. For this reason, we chose this definition as the basis for our work.

The distinction between components and connectors is important, as the operational semantics of a component are not considered on the architectural level, while those of connectors are. This question is detailed in the following subsection.

3.3.2 Connector as a First-class Entity?

We will first present arguments for and then against regarding the connector concept a first-class entity in modelling component-and-connector views. Proponents of either view do not always provide arguments for their view but simply assume that it must be taken.

A *first-class entity* is a modelling construct that may be modelled independently from any other entity that can be considered its owner. For a connector, this more specifically means that it can be modelled independently from the components it connects. A consequence of not being a first-class entity is that the entity cannot be typed, since describing types requires independence from the component instances that are connected. Modelling solutions that include the independent modelling construct of connector types, but make connector instances subordinate to the components are conceptually inconsistent, which reflects a hidden first-class entity status of connectors. Finally, we discuss alternative solutions to modelling connectors as first-class entities.

Arguments for regarding connectors a first-class entity The most prominent argument for regarding connectors as a first-class entity centres around the fact that the distinction allows to increase the separation of concerns, which is a generally accepted software engineering principle. In this context, computational aspects are separated from aspects of communication and coordination [Egyed et al., 2000; Medvidovic and Rosenblum, 2000]. Connectors are technical entities (and are influenced by the platform), while components are application-oriented entities (and are influenced by the application domain).

Another argument is that connectors as a first-class entity allow a better analysability of quality characteristics [Mehta et al., 2000b].

Arguments against regarding connectors a first-class entity The arguments against modelling connectors a first-class entity can usually be brought down to the doubt that there is a need for an additional modelling construct, and thus an argument for confining the complexity of the modelling language. Often, these arguments are discussed from the point of view of some automated tools that are meant to process the architectural description, which do not require the explicit modelling of connectors.

Another argument against an additional first-class construct for components is that commercial component platforms do not offer an abstraction that connectors can be uniformly mapped to. Thus, connectors do not have a direct counterpart in the implementation: complex connectors are always mapped to components, while simple connectors may not be easily identifiable at all [Abi-Antoun and Medvidovic, 1999]. However, the mapping of connectors into the implementation has not been investigated in detail yet [Mehta et al., 2000b]. In the context of the architecture-to-implementation mapping approach developed in the context of this thesis, we address this issue (see Section 11.2).

Non-first-class Modelling Alternatives for Connectors Connectors clearly play some role in describing component-and-connector views. So, when connectors are not regarded as first-class entities, the question arises what takes their place instead. The solution may be different for simple and complex connectors. Essentially, there are three possible alternative solutions, which may be combined:

Second-class entities Connectors can be modelled as second-class entities, i.e. subordinate properties of the components they connect. A consequence of this solution is that they are not typed, which adversely affects their reusability. For this reason, this solution only applies to simple connectors.

Regular components Complex connectors can be modelled as regular components, i.e. when two components need to be connected by a connector

that cannot be represented as a simple connector, an additional component is introduced between the two original components. In the modelling language, this component is not distinguished from the other components. This solution reflects the fact that most component platforms only offer one kind of component and is the most common solution for complex connectors.

Interaction components The third possibility is similar to the previous, with the difference that the newly introduced component is designated a special interaction component [Clements et al., 2002, p. 112] or connector component [Medvidovic and Taylor, 2000]. Depending on the point of view and the exact context, "interaction component" can be regarded just another term for "complex connector". However, this solution, like the previous solution, usually assumes that interaction components are indeed implemented as components, while in general, complex connectors may or may not be mapped one-to-one to components.

3.3.3 Component-and-Connector Views for Dynamic Systems

As noted in Section 3.2.2, the component-and-connector viewpoint is often considered to be static as opposed to some other dynamic viewpoint. This type of separation of viewpoints may lead to two conflicting consequences:

1. A "dynamic" view describes the interaction behaviour of the elements of the component-and-connector view. In this case, the dynamic view is clearly subordinate to the component-and-connector view. Implicitly, this understanding assumes that the underlying structure remains static. We prefer to view the information in the dynamic view to be considered part of the component-and-connector view. In the context of architectural styles, this applies in particular to the description of the connectors, since the behaviour of connectors is considered to be determined by the style (cf. Section 7.6.1). The extent and rigour to which their behaviour is actually modelled remains in the discretion of the architect in any case. This is actually reflected by most architectural description languages, which allow the description of components and connectors and place varying emphasis on the description of the interaction behaviour of connectors and the interaction points of components.

2. A "dynamic" view describes possible (run-time) changes to the structure described in the component-and-connector view. Again, the dynamic view is subordinate to the component-and-connector view. In addition, the value of the component-and-connector view becomes very questionable: it merely describes some snapshot of the system architecture, typically an initial configuration. Again, the separation of this dynamic

viewpoint from the component-and-connector viewpoint does not make much sense.

If the system's structural dynamism is considered that important, an integrated description of the structure and its dynamism should be provided that describes all possible configurations. Exemplary or typical static snapshots can then be provided as auxiliary documentation. However, there are no architectural description languages that provide a notation suitable for this purpose. There are two lines of research that head into this direction: On the one hand there are approaches to the description of architectural reconfigurations. However, these approaches focus on the operations that transform the system rather than the characterisation of the resulting system structure. On the other hand, there are approaches that describe system families, most importantly in the form of architectural styles. However, the idea of architectural styles is not to describe different states of a single system, but multiple distinct systems. Architectural styles could be used to describe the valid states of a single system throughout its runtime, but should then be combined with an approach to describe the transformations between snapshots. The construct of "architectural modes" [Hirsch et al., 2006], which has been defined as an extension for the Darwin ADL, is also meant to model the set of allowed architecture-level configurations of a single system.

As this discussion shows, the description of component-and-connector views remains an open research issue. This issue will not be tackled further in this thesis (but see Section 14). It has previously been addressed by Medvidovic [1996], for example.

3.3.4 Abstraction Levels

While the general notion of architectural layers is difficult to capture in a formal manner, because the layers may be different in different architectural views, we will discuss a more refined notion for the component-and-connector viewpoint, that of abstraction levels in component-and-connector configuration. The underlying idea is that a system can be described with different degrees of proximity to the implementation. For an object-oriented system, the closest possible configuration is isomorphic to the structure of the classes or objects. When the system is implemented on some component platform, each object and class is confined in a component, and the structure of the implementation components induces an architectural component configuration as well. In general, the closest possible configurations are isomorphic to the structure imposed by the elements of some specific kind (classes, objects, implementation components) that can be found genuinely within the implementation. This dimension can also be understood as a refinement

of the distinction of "architecture" and "design" (cf. Section 3.1.1).

There are several variations of this distinction:

1. Mehta et al. [2000a] distinguish "conceptual architecture" and "concrete architecture". In their view, conceptual architecture is an informal box-and-lines representation, which does not allow reasoning since it misses relevant information that is usually considered to be on the implementation level. The concrete architecture adds this information, in particular on the properties of the connectors, and is a more rigorous representation of the system.

2. Zendler and Schwartzel [1998] distinguish "logical architecture" and "physical architecture".

3. Engels et al. [2008] distinguish the same terms for enterprise-wide architectures. "Domains" are elements of the logical architecture, while applications as groups of components are elements of the physical architecture. In an ideal situation, each component is constrained to one domain, but in practise deviations from this principle may occur. Thus, the mapping of logical to physical architectures may be complex.

The distinction of platform-independent and platform-specific models, which is made in the context of the Model Driven Architecture approach [OMG, 2001] is similar in spirit as well. However, the models that are considered in the context of MDA are not genuinely descriptions of component-and-connector views but have strong aspects of the data view. Furthermore, in a typical instance of the MDA approach, there are several model levels where each model level is considered as both platform-independent and platform-specific with respect to different platforms.

Still more into the direction of the data view are different levels of data models, which are distinguished by database researchers. Usually, three levels are distinguished here: Conceptual, logical and physical data models [see, e.g., Atzeni et al., 1999]. Here, a conceptual data model is completely independent from the technology used to implement the database and is indifferent to whether a relational or object-oriented database management system (or any other type of DBMS) will be used. The logical data model then is specific to the general type of DBMS, i.e. relational or object-oriented. The physical data model is a description of the internal representation of the database within the database management system. It is mainly of interest for optimisation purposes and is analogous to design documents that are generated from code. The terminology in the context of software architecture described above is obviously inspired by this distinction.

3.4 Architectural Design Constraints

Architectural constraints form part of the rationale used to establish, maintain and employ an architectural description. Their rationale has a strong influence on the way of making decisions during the architectural design process. Terms referring to architectural constraint concepts encompass "architectural style", "architectural pattern", "architectural metaphor", and the like. In this section, we discuss several types of *codified* constraints [Riehle and Züllighoven, 1996] on software architecture descriptions, i.e. such constraints which have some explicit description separable from the software architecture of a specific system under consideration.

When an architectural constraint is selected, it restricts the design space that needs to be further considered by capturing a set of related architectural design decisions. Additionally they pre-structure the decision process of dependent decisions, which may be regarded as options enabled by the decision for a constraint. While these restrictions are useful for improving the design process in general, they particularly help less experienced designers in reusing approved design knowledge for architecting high-quality software systems.

In this section, we will discuss *architectural rules* (Section 3.4.1), architectural styles (Section 3.4.2), and design patterns and architectural patterns (Section 3.4.3). The section closes with a brief overview of other types of architectural constraints (Section 3.4.4).

3.4.1 Architectural Rules

An architectural rule [Becker-Pechau et al., 2006] specifies an isolated unit of architectural knowledge as a provision for a software architecture. They are often informally specified, but tools for checking formally specified architectural rules exist as well (e.g. Sotograph [Bischofberger et al., 2004]).

Examples Assuming that an enterprise-wide business information system has been partitioned into separate domain units based on functional (rather than technical) characteristics, an architectural rule might be: Communication that crosses domain unit borders must take place through a dedicated Enterprise Service Bus [Engels et al., 2008, ch. 5.2].

3.4.2 Architectural Styles

An architectural style groups a set of related isolated architectural rules to a named entity. Instead of considering a large number of fine-grained design decisions which generate a huge design space, in which many points may be excluded due to conflicts of the corresponding alternatives of different

decisions, a single decision is made between a smaller number of better-known architectural styles.

There are several incompatible definitions of architectural styles. The definition that we use is closest to that developed at the SEI by David Garlan and others [Abowd et al., 1993, 1995; Allen, 1995; Garlan, 1995; Garlan et al., 1994, 2002; Monroe and Garlan, 1996]. It builds upon the notion of a family of software architectures:

Definition 3.3 (Family of Software Architectures)**.** A family of software architectures is a set of software architectures that share some common property. It is called a *formal* family of software architectures if the common property is decidable.

Distinction A family of software architecture should not be confused with a software product family. While the architectures of the members of a software product family form a family of software architectures, there are other families of software architectures that are not related to software product lines. Furthermore, a product family architecture may describe all instances of the product family within a single architectural description which includes a specification of all variations within the instances of the product family. Such descriptions are not related to architectural style description.

Definition This leads us to the following definition of architectural style:

Definition 3.4 (Architectural Style)**.** An architectural style defines a formal family of component-and-connector views[7] by providing a vocabulary of component and connector types and a set of declarative configuration rules. The common property of the formal family is the conformance to the configuration rules.

In other words, an architectural style is a model of the corresponding family of component-and-connector views. Since architectural styles are related to component-and-connector views, the term "architectural style" is somewhat misleading and should possibly be replaced by "component-and-connector style" or a similar term, but we keep the established term.

Conformance to an architectural style ensures internal conceptual coherence and consistency of an architecture. Moreover, it may establish specific quality properties of the resulting system, e.g. good scalability. Thus, checking an architecture for conformance to some architectural style is an important architectural analysis. Performing conformance checks is a distinguishing property of architectural styles in comparison with design patterns, which embody the same concepts but are defined archetypally only (see Section 3.4.3.1).

[7] To be even more exact: a formal family of models of the component-and-connector view.

Elements of an Architectural Style Garlan [1995] discusses several different approaches to the definition and use of architectural styles, but assumes several common properties of any view of *architectural style*:

a. The provision of a vocabulary of design elements, i.e. component and connector types. Connector types determine interactions between components, while component types determine the packaging of functionality into components [Shaw, 1995b]. Thus, the architectural style does not only affect connectors but also the number of components and the allocation of functionality to them.

b. The definition of a set of configuration rules.

c. The definition of a semantic interpretation, which gives some well-defined meaning to all configurations of design elements that satisfy the configuration rules.

d. The definition of analyses for configurations of that style. Examples include schedulability analysis, deadlock analysis, code generation, and conformance checking.

General-purpose Architectural Styles Architectural styles encompass *general-purpose* architectural styles as well as *platform-oriented* architectural styles. Catalogues of well-known and widely used general-purpose architectural styles have been discussed in the literature for some time [Abowd et al., 1993; Monroe et al., 1997; Perry and Wolf, 1992; Shaw, 1995a,b, 1996; Shaw and Clements, 1997; Shaw et al., 1995]. Important general-purpose architectural styles include:

Pipes-and-filters Architectures in the pipes-and-filters architectural style use filters as components and pipes as connectors, which often assumes a stream-based processing (as in Unix process pipes), but other interaction styles are conceivable. Loops are not allowed in this style, while forks are only allowed in specific variants of this style.

Blackboard The blackboard architectural style is a data-centred style, which makes use of a central data repository, which is accessed directly by the components. Architectures in the style thus have a star topology.

Event-based Architectures in event-based architectural styles employ independent computational processes as components, which are implicitly invoked on the basis of event matching specifications.

Client-server The client-server style distinguishes client and server components as component types, which interact via transactions. Shaw and Clements [1997] refer to this as the full client-server style.[8]

[8] Shaw and Clements [1997] identify different flavours of the client server style, which

3.4.3 Design Patterns and Architectural Patterns

Software design patterns were made popular by Gamma et al. [1995]. The idea of "design patterns" as expressed by the "Gang of Four" [Gamma et al., 1995, ch. 1.1] is close to Alexander's idea of a pattern [Alexander et al., 1977] in building architecture. A pattern is a "solution to a problem in a context" [Gamma et al., 1995, p. 3], and consists of a name and the description of the problem, solution and consequences.

A distinction of design patterns and architectural patterns has been made by Buschmann et al. [1996]. However, we avoid the term "architectural pattern" and refer to architecture-level design patterns instead, if it is necessary to emphasise the architectural level.

Although Gamma et al. are not concerned with architecture-level design patterns, but with "descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context" [Gamma et al., 1995, p. 3], many of their patterns can also be interpreted from an architectural point of view, e.g. Mediator or Observer.

Design patterns are documented in some variant of a *pattern template*. A pattern template provides a structured frame for natural language text, which is usually amended by design diagrams.

In addition, there has been work on the formalisation of design patterns [Caporuscio et al., 2004; Eden et al., 2004; Raje and Chinnasamy, 2001; Taibi, 2007].

Since architectural styles and architecture-level design patterns are often confused, we provide a detailed discussion on the differences between the two concepts in Section 3.4.3.1. Afterwards, we discuss sets of patterns or pattern collections (Section 3.4.3.2). Design patterns are often briefly referred to as "patterns" only, which may lead to confusion with other concepts called "patterns" as well, so we briefly note these other uses in Section 3.4.3.3.

3.4.3.1 Architectural Styles vs. Architecture-level Design Patterns

In Table 3.1, we give a summary of the comparison of several aspects of architectural styles and design patterns. In the following, all of these aspects are discussed in detail. Since both terms are not used consistently by many authors, and only rarely an explicit definition is given at all, the discussion refers to an idealised view of both concepts, which aims at exposing their differences, rather than being faithful to all uses of the terms.

Protagonists and Communities There are two different scientific communities that deal with architectural styles and design patterns, which is prob-

assume different execution models: the naive client-server style has call-and-return interactions, the process-based naive client-server style has concurrent interacting processes, and the full client-server style is data-centred with transaction operations.

	Architectural Style	Design Pattern
Protagonists	Garlan (SEI), Medvidovic (USC)	"Gang of Four", Buschmann
Content	common repertoire of ideas	
Notation	Formal Language (ADL)	Natural Language + Design Fragments (Pattern Template) incl. Code Fragments
Role in Design Process	Basis for Architecture Specification	Basis for Implementation
Analysis	conformance	identification, recovery
Combination	specifically designed	arbitrary
Origin	varies	from specific existing systems
Precedents	not required	required

Table 3.1: Comparison of Architectural Styles and Design Patterns

ably the most apparent feature of the two concepts. On the one hand, there is a *pattern community*, which was inspired by the work of Alexander et al. [Alexander, 1964, 1979, 1999; Alexander et al., 1977], and widely established through the publication of Gamma et al. [1995]. On the other hand, there is a part of the community of Architectural Description Languages (ADLs) which discusses the representation of architectural styles in ADLs (we will call this the *ADL/Style community*).

While rarely the ADL/Style community refers to the term "design patterns" [e.g., Shaw and Clements, 1996] and more often the pattern community refers to the term "architectural styles" [e.g., Buschmann et al., 1996], an in-depth discussion of the other community's concepts is virtually non-existent on both sides[9].

In a wider view, the pattern community is mainly concerned with two of the discussed aspects as their defining characteristics: The notation and the existence of precedents. The content of the patterns is not limited to design knowledge in this wider view. However, our discussion at this point is focused on design patterns only, other patterns are briefly discussed in Section 3.4.3.3.

Content Both architectural styles and design patterns use the same repertoire of ideas, e.g. the idea of connecting components processing streams in a way pertaining to the "pipes-and-filters" metaphor, which exists in formulations as an architectural style and a design pattern. Sometimes, the question is raised whether any design pattern could be expressed as architectural style

[9] The author of this thesis undertook an attempt to change this in the presentation of Giesecke and Hasselbring [2006].

(or vice versa). Due to the vague definition of what a design pattern is, this question is probably impossible to answer. Similarly, one might ask whether known design patterns would be regarded as forming a valid architectural style by architectural style proponents (and vice versa)—a question that could be assessed empirically but not objectively. However, this question is not further considered in this thesis.

Notation Architectural styles are formulated in special ADLs (cf. Section 3.5) which allow the specification of architectural styles and architectural description that are instances of an architectural style. The aspects that are specified vary depending on the ADLs, e.g. Acme focuses on structural aspects, while Wright emphasises interaction aspects.

Several researchers work on formalising design patterns [Caporuscio et al., 2004; Eden et al., 2004; Raje and Chinnasamy, 2001; Taibi, 2007], but this is not the main view of design patterns. Similarly, architectural styles are sometimes considered in a less formal way, particularly in early work referring to architectural styles [Perry and Wolf, 1992].

Role in Design Process Design patterns are used to affect the implementation directly and only have an indirect effect on the design process, while architectural styles guide the architecture design process: the selection of an architectural style does not already manifest itself in the architectural views [Giesecke et al., 2007b]. They guide the development of a software architecture, since they constrain the vast architectural design space and thus ease design decisions.

In the case of design patterns, the embedding into the overall design process is essentially ad-hoc: the architect decides when to consider the use of a design pattern based on their experience. It remains controversial whether explicit documentation or internalised use of design patterns with them being implicitly applied results in better software [May and Taylor, 2003; Odenthal and Quibeldey-Cirkel, 1997; Prechelt et al., 2002]. The explicit documentation is independent from the explicit selection of design patterns: The documentation may happen concurrently to the selection and adoption, or it may happen afterwards through analysis of the source code (see next section).

In the case of architectural styles, various kinds of usages within the design process have been proposed. This aspect is discussed in more detail from a slightly different point of view in Chapter 6.

Analysis When an architectural description is created based on the description of an architectural style, it is possible to check the specified architecture for conformance to the architectural style.

One could think of trying to match an arbitrary architectural description, i.e. one that does not reference an architectural style, against a catalogue of architectural style descriptions, to identify the style (or styles) the architecture conforms to. There are several problems with such an endeavour:

- Names of component and connector metatypes must be matched. Even if the names match exactly, this does not ensure that the semantic connotation matches. Typically, architectural styles do not specify all aspects formally (e.g., in Acme only structural/topological aspects can be specified) that are connoted with them.

  When the names of such metatypes do not match, matching might become very arbitrary: Architectures that are topologically homomorphic may nevertheless conform to very different architectural styles.

- In practise, an architectural style is not applied strictly. For this reason, an approach for defining architectures based on architectural styles should allow the explicit specification of deviations.

For design patterns, there are tools that attempt the recovery or identification of known design patterns in program code [Antoniol et al., 2001; Huang et al., 2005; Philippow et al., 2003]. Since the usual description of design patterns is considered to be archetypal only, such tools need to refer to more restricted models of the design patterns. Therefore, the accuracy and recall of such tools is limited.

Due to the same reason (pattern description being archetypal in nature), it makes no sense to analyse a program for "conformity" to a design pattern description in the sense we discussed above for architectural styles.

Garlan [1995] also regards code generation as an analysis. Style-specific code skeleton generation from an architectural description is possible, which also supports architecture-to-implementation traceability. Similarly, there are tools for the instantiation of patterns.

Combination In a software system, usually not a single elementary architectural style or architecture-level design pattern is used, but a combination of multiple styles or patterns. The approach to handle such combinations is different for styles and patterns.

In the case of architectural styles, it is usually assumed that for a given partition and layer of a software architecture, only a single style can be used. However, this does not suffice for all practically relevant situations, in particular because a system cannot be partitioned into unconnected subsystems that use different styles, but there needs to be some interface between these subsystems. For example, one might think of a pipes-and-filters system that makes use of a client/server database access. The component serving as database client needs to be a filter as well. To use architectural

styles in such a situation, a combined architectural style must be defined. No calculus for combining architectural styles is provided, but the combination must be specified manually. It may be possible to specify the combination as a formal specialisation of either or both of the combined styles. In general, many variants of combinations of the same styles are conceivable.

For architecture-level design patterns, the situation is different. While it is possible to define combinations of architecture-level design patterns manually and describe them as a distinct pattern [see, e.g., Riehle, 1997], multiple architecture-level design patterns may be instantiated subsequently and independently from each other. Conflicts that occur are handled during instantiation. This is compatible with the architecture-level design pattern concept, since the pattern description is considered to be only archetypal.

Origin The origin of architectural styles varies. In principle, a valid architectural style does not need to have a known origin, it may be specified by merely using the syntax of the ADL used for its specification. Typically, generally known design expertise or design rules for a specific platform are captured in the form of architectural styles. On the other hand, architecture-level design patterns are necessarily specified on the basis of existing software systems. More specifically, a number of precedents are required, which are discussed in the next paragraph.

Precedents Authors on design patterns often emphasise that patterns must capture recurring, but non-obvious design practise (Gamma et al. [1995] and with particular emphasis in Coplien [1996a]). Thus, there is no such notion of the "invention of a design pattern", but the pattern can only be discovered through the analysis of existing software systems, where instances of the patterns are identified. These instances are documented in the pattern template as "known uses" [Gamma et al., 1995] of the pattern, or *"precedents"*, i.e. instances of a pattern that precede the formulation of the pattern description.

There is no comparable claim held in the ADL/Style community.

A consequence of the reliance vs. non-reliance on precedents is that architecture-level design patterns are always known to be good for some purpose, i.e. when writing them down an evaluation of the presented solution has already been done[10]. Architectural styles are neutral to their suitability for some purpose at first. They are first specified, and can be evaluated afterwards. In fact, a method for evaluating architectural styles is presented in this thesis (cf. Section 8).

[10] Often this evaluation is purely anecdotal and not systematic, but it needs to exist in some form nonetheless.

3.4.3.2 Pattern Collections

Multiple patterns, for example for a specific technology domain, are organised into pattern collections. Several types of pattern collections are distinguished. The patterns within a pattern collection are related to each other in different ways.

Pattern Catalogues, Systems and Languages Buschmann et al. [1996] distinguish three types of pattern collections, which are presented here in the order of increasing coherence:

Pattern catalogues A pattern catalogue describes a collection of patterns, and possibly provides a coarse categorisation of them.

Pattern system A pattern system is more coherent than a pattern catalogue by systematically identifying and documenting relationships of the contained patterns.

Pattern languages A pattern language is a pattern system that aims to be complete for some family of systems, e.g. a family of systems of a certain application or technology domain. By "complete" Buschmann et al. understand that any system from the family can be constructed merely by combining a subset of the patterns and exploiting their documented relationships within the process.

Relationships between Patterns One part of the description of a design pattern are relationships to other design patterns. Several types of such relationships are distinguished, e.g. usage or variant relationships. Design pattern relationships are discussed by Alpert et al. [1998]; Buschmann et al. [1996]; Dyson [1997]; Lorenz [1997]; Meszaros and Doble [1997]; Noble [1998]; Zimmer [1995]. Englisch [2006, sec. 2.1.3] presents a taxonomy of these relationships. An overview of this taxonomy is shown in Figure 3.2. It identifies identity, specialisation and inversion relationships between pattern relationship types. The relationships of several patterns from Gamma et al. [1995] are shown in Figure 3.3.

3.4.3.3 Software Patterns other than Design Patterns

Besides patterns that concern software design or architecture, there are other patterns in software development. These include units of knowledge that have a different topic, but are documented in pattern form as well. These include workflow patterns [Van Der Aalst et al., 2003], requirements patterns [Withall, 2007], analysis patterns [Fowler, 1997], which are concerned with domain analysis, and reengineering patterns [Demeyer et al., 2003]. Additionally so-called anti-patterns have been described, some of

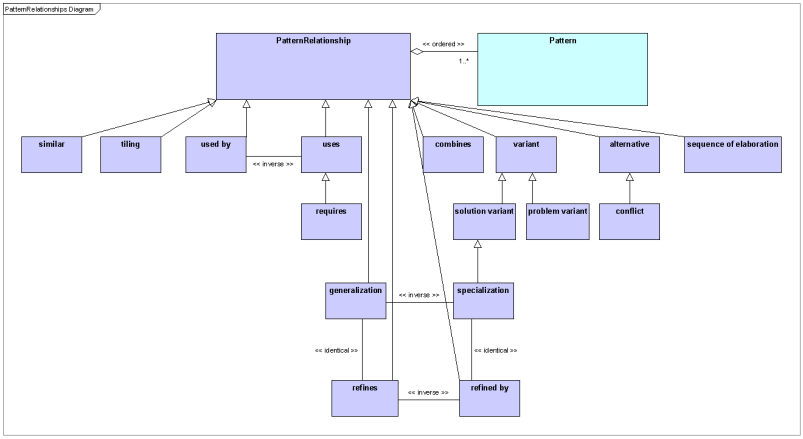

Figure 3.2: Types of pattern relationships [from Englisch, 2006]

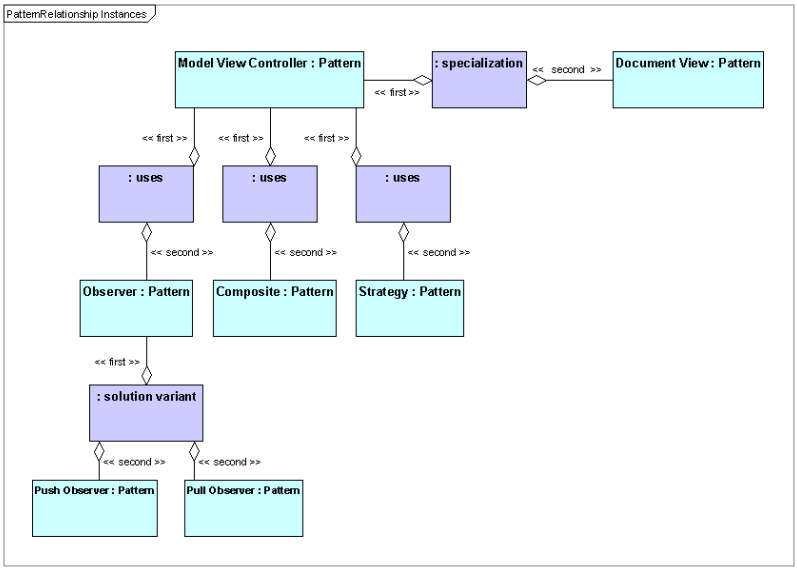

Figure 3.3: Example of pattern relationships [from Englisch, 2006]

which concern software design, but are not written in pattern form [Brown et al., 1998].

3.4.4 Other Architectural Constraint Types

Other types of architectural constraints include *generic architectures* such as reference architectures or architectural skeletons, which exist at different levels of scope, e.g. for a technology domain or for an application domain.

A variant of generic architecture are parametrised architectures such as *product-line architectures* (or product family architectures).

3.5 Architectural Description Languages

An Architectural Description Language is a formal language [Rozenberg and Salomaa, 2004] that specifically targets the representation of one or more architectural views. Of particular interest to this thesis are Architectural Description Languages that allow the representation of some variant of the component-and-connector view. All known existing Architectural Description Languages allow this. In fact, Medvidovic and Taylor [2000][11] regard the provision of distinct modelling constructs for components and connectors as a defining property of Architectural Description Languages. Architectural Description Languages have emerged from research on Module Interconnection Languages (MILs) [Prieto-Diaz and Neighbors, 1986].

In this section, we first describe basic ideas that underlie Architectural Description Languages (Section 3.5.1) and then describe several individual Architectural Description Languages (Section 3.5.2). Finally, we describe the use of the UML as an ADL (Section 3.5.3).

3.5.1 Basic Ideas of Architectural Description Languages

A survey that classifies Architectural Description Languages is provided by Medvidovic and Taylor [2000]. There are some earlier surveys such as Clements [1996], but these are not as comprehensive. A comprehensive survey of more recent Architectural Description Languages is not available, although many specialised Architectural Description Languages have emerged since the earlier surveys.

The survey [Medvidovic and Taylor, 2000] proposes a classification scheme for Architectural Description Languages with several categories: For each of the two main basic modelling constructs, i.e. components and connectors, the scheme includes categories regarding the ability to describe interfaces, types, semantics, constraints, evolution and non-functional properties. For configurations, the categories understandability, composition,

[11] Building upon a definition in Tracz [1993].

refinement/traceability, heterogeneity, scalability, evolution, dynamism, constraints and non-functional properties are offered. These categories are related to a discussion of the architectural viewpoints that are covered by the respective ADL, however Medvidovic and Taylor [2000] do not refer to this notion or any model of multiple architectural views (cf. Section 3.2.2).

Typically, an ADL distinguishes component types and component instances. Component types specify interaction points, often called ports. When component types are instantiated within a configuration, interaction points of components and connectors are bound. Which types of interaction points may be bound together, depends on the ADL.

An important aspect of ADLs for this thesis is the support for the definition of architectural styles, either by an explicit style modelling construct or some other means to define families of configurations, generic configurations, or refineable configurations.

Most ADLs are primarily textual, some offer a well-defined graphical representation in addition, but this is typically not complete[12]. The layout information is, on the other hand, usually not contained in the textual representation.

In addition, tool support is analysed regarding interactive specification, multiple views, analysis, refinement, implementation generation and dynamism. Inverardi et al. [2005] provide a more recent overview of tool support available for different Architectural Description Languages.

3.5.2 Examined Architectural Description Languages

Several Architectural Description Languages have been examined regarding their suitability for modelling Middleware-oriented Architectural Styles during the course of this thesis. UniCon (Section 3.5.2.1), Rapide (Section 3.5.2.2), Darwin (Section 3.5.2.3), Acme (Section 3.5.2.4) and Wright (Section 3.5.2.5) are earlier Architectural Description Languages, which have been discussed by Medvidovic and Taylor [2000]. We follow Medvidovic and Taylor closely in the discussion of these languages. Medvidovic and Taylor provides a discussion of C2, MetaH, SADL and Weaves in addition to these, which were not considered in more detail within this thesis. The Aesop system was also discussed by Medvidovic and Taylor [2000], but it does not match our definition of an ADL because it is not really a language on its own. However, in addition to the Architectural Description Languages discussed in Medvidovic and Taylor [2000], we present more recent ADLs: xADL (Section 3.5.2.6), and Alfa (Section 3.5.2.7).

A more detailed discussion of these ADLs in the context of this dissertation project has been done by Bornhold [2006]; Hilbrands [2006]; Marwede [2007].

[12] This even applies to the UML, which is usually used only in its graphical form.

ADL	Style modelling construct
UniCon	predefined styles only
Rapide	none specific, POSETs?
Darwin	none
Acme/Armani	invariants and element types via “family” construct
Wright	invariants via “style” construct
xADL	none
Alfa	five groups of primitives for modelling styles
UML	none

Table 3.2: Style Modelling Constructs in ADLs

Table 3.2 shows an overview of the availability of style modelling constructs in the examined ADLs. Details for each ADL are discussed in the following sections.

3.5.2.1 UniCon

UniCon [Shaw et al., 1995; Zelesnik, 1996] was developed at Carnegie-Mellon University between 1992 and 1997. It is an early ADL and has still much resemblance to MILs in that it requires a high degree of fidelity of the architecture to its implementation. Still, it is an ADL according to the definition of Medvidovic and Taylor and an explicit configuration ADL. Its primary purpose is to allow the interconnection of existing components and the generation of glue code connecting these components.

Components in UniCon define interaction points called *players*, which are typed. A number of predefined players are available in UniCon, as are a number of predefined component types. Adding component or player types is not supported by UniCon. Connectors define roles as interaction points. Again, only a predefined set of role types is supported. In addition, specification of implementation mappings is supported, which is used for code generation. A configuration in UniCon is simply a composite component.

Language mechanisms that can be used for modelling architectural styles are not available in UniCon. However, through the predefined component, player, connector, and role types, several predefined architectural styles can be understood to be modelled within the language definition, e.g. the pipes-and-filters style. UniCon supports specific architectural analyses for some styles, e.g. a schedulability analysis for rate-monotonic processes.

Tool support is available for graphical, syntax-driven modelling and C language code generation.

3.5.2.2 Rapide

Rapide [Luckham et al., 1995; Rapide Design Team, 1994] was developed from 1990-1998 at Stanford University. It differs from most other ADLs by its strong focus on the behaviour of the described configuration. Rapide does not assume a close resemblance of the architecture and its implementation (implementation independent language [Medvidovic and Taylor, 2000]). Rapide is a complex language consisting of several sub-languages: Constraint Language, Reactive Programming Language, Pattern Language and Type Language.

What is called "component" in most other ADLs finds its counterpart in Rapide's *interfaces* which define *constituents* as interaction points. Rapide explicitly distinguishes *provided* and *required* constituents on the one hand, and *synchronous* and *asynchronous* constituents on the other hand. They are interconnected in *architectures* by *connections* in an in-line manner (i.e., Rapide is an inline configuration ADL). Connections are thus untyped in Rapide. Complex connections can be refined by *connector interfaces*. The behaviour of interfaces, connections and architectures are constrained through POSETs, partially ordered event sets. These can be modelled using Rapide's Pattern Language.

Rapide considers dynamic architectures, and requires the configuration to specify all allowed run-time configurations. Special events allow rewiring of connections at run-time.

A language mechanism in Rapide that could be used for modelling architectural styles is that of POSETs, which allow to define constraints that are global at the configuration level. In addition, it allows the refinement of architectures.

Rapide provides a graphical modelling environment. Moreover, Rapide tools (Simulator, Animation Tools, Poser Browser) allow the simulation and visualisation of the behaviour of a configuration without having an actual implementation available.

3.5.2.3 Darwin

Darwin [Magee et al., 1995] was originally developed at the Imperial College London between 1991 and 1997. The operational semantics of Darwin is based on the π-calculus. The focus of the language are highly distributed systems, whose behaviour needs to be formally specified.

Components specify *portals* as interaction points. *Bindings* are specified in-line in *configurations*. Similarly to Rapide, complex bindings can be refined by *connector components*. As in UniCon, a configuration is not distinguished from a composite component. Darwin offers the possibility to specify arrays of components. Provided and required services are explicitly distinguished, and the correct binding can be checked.

Darwin, again similarly to Rapide, allows to explicitly model dynamic structural changes of the configuration, which are specified using *scripts*.

The Darwin "Software Architect's Assistant" is a support environment for modelling Darwin configurations. Darwin tools support the generation of code skeletons in C++.

3.5.2.4 Acme & Armani

Acme [Garlan et al., 1997, 2000] was originally designed as an language for interchange between other ADLs, which provides only basic structural modelling constructs for the component-and-connector viewpoint by itself and allows the integration of other viewpoints by including fragments from other ADLs. Later, this design goal of Acme was no longer placed special emphasis on, probably because the syntactic and semantic integration of fragments from different ADLs proved harder than expected.

Independently from Acme, another language called Armani [Monroe, 2000b] was designed with the purpose of providing an ADL with means to declaratively specify design rules. Later the design rule sub-language of Armani, which is based on first-order predicate logic, was merged into Acme.

Components in Acme specify *ports* as interaction points, while *connectors* provide *roles*. Ports and roles are bound to each other in *systems*. Hierarchical systems can be modelled through representations and representation mappings (rep-maps). All design elements can specify named properties with an arbitrary content that is not interpreted by Acme tools. The idea was to allow to include fragments of other ADLs within an Acme specification. Acme does not allow to specify the behaviour of components and connector in its own language. Support for specifying dynamic structural changes of an architecture is also not included in Acme.

Acme is the only of the earlier ADLs that allows the definition of architectural styles explicitly through its *family* construct. For this purpose, it provides constructs for the definition of components and ports as well as connectors and roles, and of types for each of these architectural elements. By the Armani sub-language design constraints for member architectures of a family can be specified.

A graphical modelling environment for Acme is available, called AcmeStudio [Schmerl and Garlan, 2004], which is based on the Eclipse platform in its current version. In addition, to programmatically access Acme specification, a Java programming library called AcmeLib is supplied.

3.5.2.5 Wright

Wright [Allen and Garlan, 1997] was developed at Carnegie-Mellon University. Its main purpose is modelling the dynamic behaviour of concurrent

systems, and to allow formal analysis of that behaviour, particularly deadlock analysis. The modelling of behaviour is based on CSP (Communicating Sequential Processes, a process algebra).

Components in Wright specify *ports* as interaction points, while *connectors* provide *roles*. Both port interaction semantics as well as connector glue semantics are specified in CSP. Ports and roles are bound to each other in *attachments*, and the compatibility of attached ports and roles can be analysed by evaluating their CSP specifications. Both composite components and connectors can be modelled: Then, the behaviour respectively glue is again an architectural specification, and the overall semantics is evaluated through composing the behaviour/glue of the constituents. Wright allows the specification of dynamic architectures through *control events*, that are distinguished from *communication events*.

Wright allows the specification of invariants that define an architectural style, and provides a *style* modelling construct for this purpose.

Since an analysis that is well-supported for CSP is deadlock analysis, this analysis is also the main focus of Wright tools.

3.5.2.6 xADL

xADL 2.0 [Dashofy et al., 2005] is an XML-based ADL which evolved from a traditional line of ADLs at the University of California at Irvine. xADL is a collection of extensions to the xArch [Dashofy et al., 2006] core ADL, which is meant to be a "standard, extensible XML-based representation for software architectures" [Dashofy et al., 2006]. xADL was designed in a modular and extensible fashion which is based on the modularity and extensibility of XML and XML-Schema [World Wide Web Consortium, 2006].

In xADL, both *components* and *connectors* have *interfaces*, which are attached to each other in *structures* through *links*. xADL distinguishes the type and structure level at design time from the instance level at run-time. E.g., there are component types and components at design time, and component instances at run-time. Besides the core language, modules are provided for specifying implementation mappings, and security properties, for example.

Neither xArch nor xADL provide specific support for modelling architectural styles or families. While it is easy to add constructs to the language through its extension mechanisms, tool support for the new constructs must still be added.

Tool support is available on different levels. On the syntactical level, xADL benefits from its XML basis. Generic tools can be used out of the box, e.g. XML validators can validate xADL and also custom extensions. Another example are syntax-based editors, which can understand the schemata and adopt to custom extensions automatically. Specific xADL tools [University

of California at Irvine] are a data binding library and a generator which automatically generates a custom data binding library from an XML Schema. Additionally some higher-level tools are available. A comprehensive modelling environment, ArchStudio 4, which is based on Eclipse, is available.

3.5.2.7 Alfa

Alfa [Mehta and Medvidovic, 2002, 2003] is an ADL developed at the University of Southern California. It embodies higher-level architectural primitives that have been distilled from common properties of several architectural styles. These architectural primitives are thought to be suitable to model arbitrary architectural styles, and indirectly architectures.

Five groups of primitives, or characteristics are provided: Data, Structure, Interaction, Behavior and Topology. Data characteristics describe the data that can be exchanged through a system's structure. The most important structural primitive is a *particle*, which subsumes both components and connectors. These provide *input* or *output* ports, which are connected through primitive *ducts*. Interaction primitives describe the communication between particles, while behaviour primitives describe the internal behaviour of particles. Topology primitives describe structural dynamism.

Two tools are provided for Alfa, a modelling environment called ViSaC and a compiler, Alfaac, which generates code from an architecture description.

3.5.3 Using UML for Architectural Description

The Unified Modelling Language [OMG, 2007b] was originally designed by the Object Management Group as an object-oriented modelling notation, which is distinct from architectural description languages [Garlan et al., 2002; Medvidovic and Taylor, 2000]. It was clearly targeted as a notation for lower-level design documentation (cf. Section 3.1.1). However, with the definition of UML 2.0, several extensions were made to retarget the UML as a general-purpose modelling language. One of the goals of the extensions was to better allow to model software architectures. The question whether this goal has been achieved is highly debated. This is in part due to the fact that there is no common understanding on what software architecture is, which leads to even more opinions on how software architectures should be described. The UML merely defines the syntax and a basic semantic interpretation[13], but it does not prescribe a specific method for using it in software development.

Garlan et al. [2002] state three requirements for such methods:

[13] There is much criticism on the fact that the UML does not define a rigorous semantics, but explicitly leaves several "semantic variation points" unspecified.

Semantic match The syntax and semantics of the UML and "intuitions" of UML modellers should be respected, so that models are usable by UML tools and modellers.

Visual clarity "The resulting architectural descriptions in UML should bring conceptual clarity to a system design, avoid visual clutter, and highlight key design details." [Garlan et al., 2002]

Completeness The method should include all relevant architectural concepts. According to Garlan et al., these include components & ports, connectors & roles, configurations ("systems"), extra-functional properties, and architectural styles.

Several methods for using the UML for architectural description (in the component-and-connector viewpoint) have been proposed, which can be classified into three categories [cf. Abi-Antoun and Medvidovic, 1999], which satisfy the requirements of Garlan et al. to different degrees:

- Methods that use plain UML
- Methods that use light-weight extensions to the UML
- Methods that use heavy-weight extensions to the UML

Architecture modelling methods using plain UML are often informally described in textbooks on software architecture or the UML, for example by Posch et al. [2004, p. 144f]. These approaches best fulfil the semantic match requirement, at the cost of the completeness requirement. The missing completeness is in particular due to the fact that the UML does not offer a first-class connector construct, neither does it provide a style construct (however, the latter is not supplied by many "true" ADLs either).

Light-weight extensions to the UML are defined through UML Profiles. UML Profiles can be interpreted by UML Tools, and thus UML Tools can be used for modelling software architectures with these methods [cf. Ivers et al., 2004]. Light-weight extensions offer a compromise of semantic match and completeness, but it may be awkward to achieve true completeness, so that visual clarity is still often impaired when trying to accurately model connectors and roles (see Section 7.7.2).

Heavy-weight extensions, on the other hand, modify the UML metamodel in a way such that resulting models are no longer conforming to standard UML. Referring to these methods as "extensions to the UML" is a kind of euphemism, since the resemblance to the UML can be arbitrarily small. Thus, the semantic match with the UML is neglected, but the completeness could be most easily achieved.

There are several approaches for mapping specific ADLs into the UML on the one hand, and on exploiting light-weight UML extensions for specific

architectural modelling approaches. These are related work to our UML-based modelling approach that is described in Section 7.7.2, and we will discuss these in Section 12.1.

Part II

The MidArch Approach

4 Research Design and Methods

In this chapter, we present the research design of this dissertation project. We structure the overall project into work packages. For each work package, research questions are posed, and the research methods that ought to be used to answer the research questions are defined.

In Figure 4.1, the work packages of this PhD research project are shown as rectangular boxes, along with dependencies between them, the primary results they are intended to produce (boxes with curly bottom). The dependencies must not be understood as time dependencies, but as completion dependencies, which are weaker: If WP2 depends on WP1 this means that WP2 cannot be completed before WP1 is completed, but WP2 may be started before WP1.

The major work packages are shown in Table 4.1. For each of the work packages, several research *questions* are identified, the research *methods* [Hasselbring and Giesecke, 2006] used to gain answers to the questions (partially based on the classification given by Vogel and Wetherbe [1984]), the type of the expected *outcome* is characterised. Not all of the research questions are handled in the same depth within this PhD thesis, some questions will primarily lead to hypotheses that can be dealt with in future work.

These work packages are explained in detail within the following sections.

4.1 WP1: Analysing Architectural Styles and their Usage

Architectural styles have proved to be a useful guidance for software development. "Architectural pattern" is a term for essentially the same concept, but usually used within a different community. While the sets of patterns and styles overlap, the two communities usually have a subtly different view on the use of the patterns/styles. The specific type of platform-oriented architectural styles is closer to the ADL-based use of architectural styles, thus we will use the term "styles" within this thesis. Within this work package, we analyse the state of the art in defining architectural styles (Q1.1), which is reported in Chapter 3. Then, the a taxonomy of the state of the art in using (Q1.2) architectural styles is modelled (Chapter 6). Based on the answers to these first two questions, a vision of a new usage of architectural styles is devised (Q1.3), which lays the foundation for the remainder of this PhD research project.

Work Package	Research Questions	Research Methods	Results
WP1: Analysing Architectural Styles and their Usage	**Q1.1** What is the nature of architectural styles and how are they related to similar software architecture concepts? **Q1.2** Which usages of architectural styles exist and how are they related to each other? (or: What is the state of the art in using architectural styles?) **Q1.3** How can these usages be made more effective, e.g. by combining their features?	Literature Survey, Interpretive	○ Taxonomy of Architectural Styles and related Concepts ○ Taxonomy of Usages of Architectural Styles ○ Report on the Conduct and Results of the Survey
WP2: Extending ISO Standard 42010 to Represent Architectural Rationale	**Q2.1** How do architectural styles relate to architectural rationale? **Q2.2** How can architectural styles be reflected within a standardised conceptual framework for architectural description?	Metamodelling, Argumentative	○ Extended Reference Model for Architectural Description ○ Documentation of the Rationale for the Reference Model Extension
WP3: Modelling Middleware-oriented Architectural Styles	**Q3.1** Which ADLs are suitable for modelling MidArch Styles and architectural descriptions exploiting MidArch Styles? **Q3.2** Can formalised MidArch Styles be induced from actual middleware platforms? **Q3.3** Is it possible to organise MidArch Styles for a given set of middleware platforms in a taxonomy including specialisation and other relationships?	Argumentative, Deductive, Empirical (Case Studies)	○ ADL Suitability Report ○ Documentation of Style Modelling Approach ○ Style Descriptions in an ADL ○ Documentation of Style Relationship Types ○ Taxonomy resp. Type Hierarchy
WP4: Developing a MidArch Style-Based Middleware Selection Method	**Q4.1** How can architecture-level software design be supported by a systematic method exploiting a MidArch Style Taxonomy for selecting an appropriate middleware platform? **Q4.2** Is the proposed method applicable? **Q4.3** How does the proposed method compare with existing Software Engineering practises in terms of productivity and predictability?	Method Engineering, Argumentative, Empirical Validation through Case Studies	○ Description of the Proposed Method ○ Report on the Case Studies ○ Summary Validation Report ○ Discussion of Related and Fundamental Work
WP5: Providing Tool Support for the MidArch Design Method	**Q5.1** How can the proposed method be supported by tools? **Q5.2** How do such tools contribute to productivity and predictability of applying the method?	Tool Development (Demonstrating the method's implementability); Empirical (Tool Evaluation)	○ Tool Prototype and Documentation of its Design ○ Experience Report

Table 4.1: Work Packages Overview

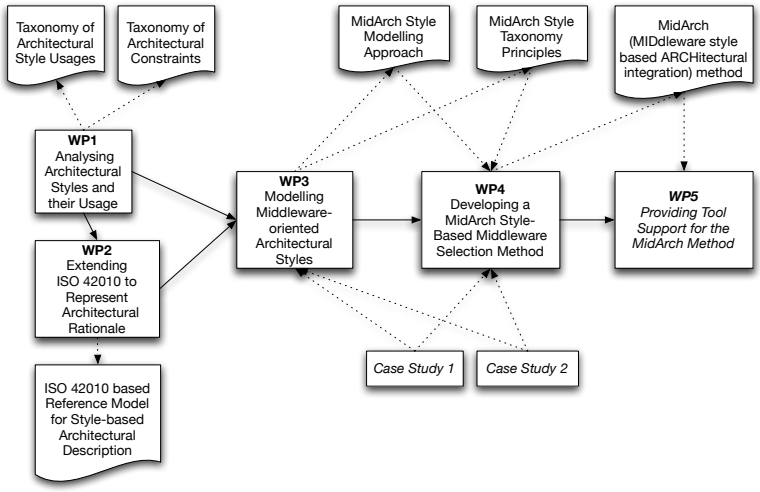

Figure 4.1: Work Packages of this PhD Project

4.2 WP2: Extending ISO Standard 42010 to Represent Architectural Rationale

ISO Standard 42010 [ISO, 2006] refers to the rationale or underlying principles of an architecture in the definition of software architecture. However, this is not equally reflected in the proposed reference model. Rationale occurs only as a monolithic block that is provided by an architecture description. We extend and refine the standard's reference model such that it encompasses the concepts necessary for representing architectural rationale including architectural styles. In our view, architectural styles form an important aspect of the architectural rationale. The relationship of styles and rationale is elaborated within this work package (Q2.1), before the actual extension of the reference model is defined (Q2.2) (Section 5.7).

4.3 WP3: Modelling Middleware-oriented Architectural Styles

Over the past 15 years, a large number of ADLs have been proposed. Some of these ADLs already provide specific modelling constructs for modelling architectural styles. However, the ability to model real business information systems and the styles they employ has not been studied in depth for most of these languages. In addition, the type of styles we intend to use might

deviate from the conception of architectural styles that was conceived in the development of the ADLs. Thus, it is necessary to evaluate the ability of ADLs to describe MidArch Styles within this research project (Q3.1). Due to feasibility considerations, no new ADL for modelling systems and their styles should be developed within this PhD research project. If the existing ADLs prove inadequate for modelling MidArch Styles, future work may include the development of an ADL specifically targeted at modelling MidArch Styles.

In addition, the concept of styles as understood by us (see WP1) is an informal one in the first place. ADLs, on the other hand, are formal languages. Thus, it is presumably impossible to represent all aspects of a style in an ADL, most importantly with respect to the pragmatics of its intended use. It should be evaluated which aspects may and which may not be formalised and which consequences for modelling and using MidArch Styles arise from the answer to this question (Q3.2).

Architectural Styles Well-known *general-purpose* architectural styles include the pipes-and-filters and layered styles (see Section 3.4.2). We consider a somewhat different type of architectural styles, i.e. those oriented towards a middleware platform. Target or *implementation* platforms may be distinguished from *modelling* platforms. A modelling platform is a modelling language or notation, for example an ADL. There are ADLs that are specifically designed to support a specific architectural style, e.g. C2SADL [Medvidovic et al., 1999] for the C2 style, but even ADLs that claim to be style-independent are usually biased towards some architectural style [Di Nitto and Rosenblum, 1999]. A middleware platform is one type of implementation platform. Since the term "middleware" is used in varying definitions, we use an artificial name to refer to the styles induced by middleware platforms (in our sense), i.e. *MidArch Styles*.

Style-based Taxonomy of Middleware Platforms Our approach does not focus directly on the selection of a specific middleware *product*, but to make a decision between architectural *styles* that are induced by middleware platforms. The motivation for this approach has two aspects.

First, the architecture should be independent from technology-specific issues to the greatest extent possible. The architecture will necessarily be biased towards some implementation platforms (through the architectural style it uses), but the tight binding towards one specific product should be avoided by all means. One of the main motivations for introducing an explicit architectural level of design is to ensure portability of the system, which would be impeded by a platform-specific approach to architecting the system.

Second, the set of available middleware products needs to be structured in some way. We believe that the styles they induce are a suitable basis

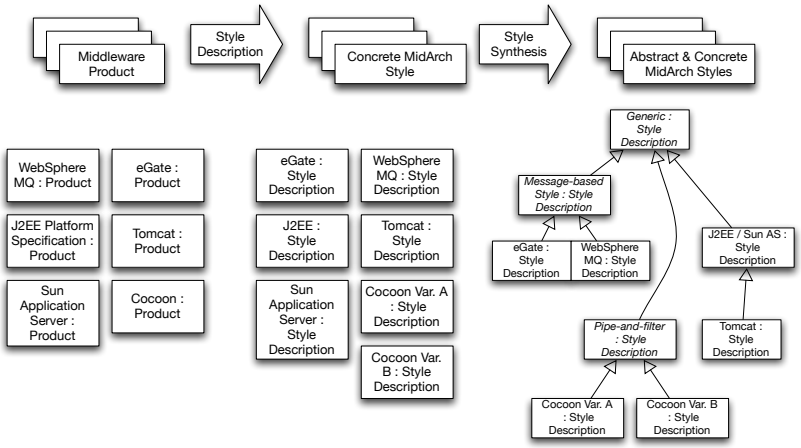

Figure 4.2: Derivation of a Platform Taxonomy

for creating a taxonomy of middleware products. One middleware product may support the use of several architectural styles. This work package shall provide an answer to the question whether it is possible to derive such a taxonomy from a given set of industrially relevant platform products (Q3.3). In creating the taxonomy, *abstract* platforms may need to be introduced in addition to actually *implemented* platforms, which is shown in Figure 4.2.

In our view, middleware is a software layer between the application and operating system layers. This characterisation does not fix the middleware layer in a universal way. Distributed object infrastructures (such as CORBA [OMG, 2004]) are only one type of middleware. In particular, one system's middleware may be another system's application. For example, when developing a new web server, the web server is on the application layer, and implementation technologies like the component platform used are on the middleware layer. However, when designing web applications that rely upon the web server, the web server is moved to the middleware layer. Likewise, the borderline between middleware and operating system is not always clear. Typically, we consider virtual machines (such as the Java Virtual Machine) as part of the operating system. In any case, the application and middleware layers must be specified for any particular project. The specification is influenced by the project's context, but is necessarily a design choice.

The results of this work package are reported on in Chapter 7.

4.4 WP4: Developing a MidArch Style-Based Middleware Selection Method

Based on existing techniques, a design method for conducting software projects exploiting MidArch Styles should be developed: the MidArch Design Method (Q4.1). The method targets middleware-intensive applications, and aims to improve the early selection of a suitable middleware. The applicability of the method is evaluated using case studies based on real systems from the domain of business information systems (Q4.2). When the applicability has been demonstrated, the question arises whether the method actually improves the realisation of software projects (Q4.3) with respect to the productivity and predictability of the process.

As a starting point, four top-level activities of the method are defined, whose enactment may overlap in time:

Definition In this activity, the scope of the project, i.e. the systems involved, and the goals to be achieved are defined.

Preparation In this activity, a project-specific quality model (in the sense of a goal/question/metric [Basili et al., 1994] quality model) is developed, and the current architecture is modelled, if an architecture description does not exist in a suitable form.

Exploration This activity is central to the proposed method and is assumed to consume the majority of the effort. It involves the preselection of MidArch Styles, modelling of candidate architectures that conform to the selected styles, and the evaluation of the resulting architectures. For evaluation, existing architecture evaluation methods can be used. Finally, the evaluation results are assessed to decide whether an architecture candidate (or a set of such) that allows to achieve the stated goals has already been found.

Implementation Based on the previous activity, the architecture that should actually be implemented must be developed, which may involve combining multiple candidate architectures, and the implementation must be performed.

Style Selection The central aspect of the proposed method, and the task to which the research project makes the most significant contribution is the preselection of candidate MidArch Styles. The selection of a middleware platform should be made based on its impact on extra-functional properties of the resulting system such as maintainability, availability, and time efficiency. The idea is that the selection process should be based iteratively on the hierarchical taxonomy of MidArch Styles, i.e. by iteratively making choices on properties of abstract platforms, finally implemented middleware platforms

are reached. The properties of the different styles should be analysed by first modelling operations to integrate existing legacy systems according to a style and then evaluating the effect on system characteristics. The selection of a style is based on guidelines for choices along the hierarchical taxonomy. Based on the taxonomy derived in WP3, the chosen style may be incrementally refined from an abstract style towards a concrete style in a software project.

The results of this work package are reported on in Chapter 8 (method definition) and Chapters 9 and 10 (evaluation).

4.5 WP5: Providing Tool Support for the MidArch Design Method

A striking question for any development method is whether it is possible to create tools that support the application of the method. Thus, for the MidArch Design Method, possible tool support should also be explored (Q5.1). After roles of tools in the application of the method have been hypothesised, the question arises whether their use actually improves the application of the method (Q5.2), i.e. whether it has an additional benefit with respect to the stated goals, i.e. to improve productivity and predictability of software projects.

The results of this work package are reported on in Chapter 11.

5 Basic Definitions and Assumptions

In this chapter, we lay out basic definitions and assumptions that underlie the thesis as a whole, which have not been already defined in the foundations part because they are new and rather specific for this thesis. In the following chapters, more assumptions that specifically apply to the material presented within a single chapter will be added. The chapter is structured into the topics software architecture (Section 5.1), middleware (Section 5.2), architecture exploration and evaluation (Section 5.3), and design knowledge (Section 5.5). The assumptions on the relationship of software development practises and software system quality are discussed in Section 5.6. In Section 5.7, we discuss an extension of the ISO Standard 42010 reference model for software architecture description that incorporates architectural styles.

5.1 Software Architecture

As discussed in Chapter 3, conflicting definitions of software architecture abound. We base our work on the definition in the ISO Standard 42010 presented in Section 3.2. The standard defines a reference model for software architecture description.

The standard's definition of software architecture is on an abstract level. Other definitions often have a specific viewpoint in focus [Software Engineering Institute, 2008]. We also focus on a specific viewpoint in our work, while keeping in mind that other views are also part of a complete software architecture description. The viewpoint we focus on is a variant of the component-and-connector viewpoint (see Section 3.3). In addition to the general assumptions of that viewpoint, we additionally assume that the components described in the viewpoint are not the physical software components that can be found as artefacts in the system but logical design components, which may or may not coincide with physical software components (cf. Section 3.3.4). The configuration of these components is described from a runtime point of view. We refer to the viewpoint as the *logical component structure viewpoint*.

In the following, for simplicity we use the convention that "software architecture" refers to a view conforming to the logical component structure viewpoint, if nothing else is said explicitly.

5.1.1 Application and Technology Domain

In this section, the application and technology domains of the software systems we chose as the subject of our research are discussed. The terms "application domain" and "technology domain" have been defined in Section 3.2.1. The application domain we consider is that of business information systems as explained in Section 5.1.1.1. Large business information systems are usually middleware-intensive software systems, which makes up the technology domain defined in Section 5.1.1.2.

5.1.1.1 Application Domain: Business Information Systems

In our work, we focus on software systems from the application domain of business information systems. We have a broad view of such systems, i.e. we do not restrict this to database-oriented applications, but mainly intend to exclude the application domain of embedded systems, which has disparate requirements and usually implies the use of specific implementation technologies. Most importantly, software for embedded systems has hard real-time requirements, while business information systems do not. Modelling techniques for embedded systems differ from those for business information systems as well.

Certainly, it is an interesting question whether the results of our research can also be applied to embedded systems, but this is out of the scope of this thesis.

5.1.1.2 Technology Domain: Middleware-intensive Software Systems

We define:

Definition 5.1 (Middleware-intensive software system)**.** A *middleware-intensive software system* is a software systems, whose logical component structure is significantly impacted by the choice of middleware technologies.

As a consequence, choosing different middleware technologies for modelling and implementing a software system for the same set of requirements would yield a structurally different system. More vaguely, Liu et al. [2006] state that middleware-intensive applications depend "on the mechanisms and services provided by the infrastructure".

While the statement above is a definition, an interesting question it raises is whether there actually are systems of commercial interest which are middleware-intensive in the sense of that definition. Our work does not address this hypothesis directly, but we rather make the following assumption, which is partly substantiated by our case studies:

Assumption 5.1. Large business information systems are middleware-intensive software systems.

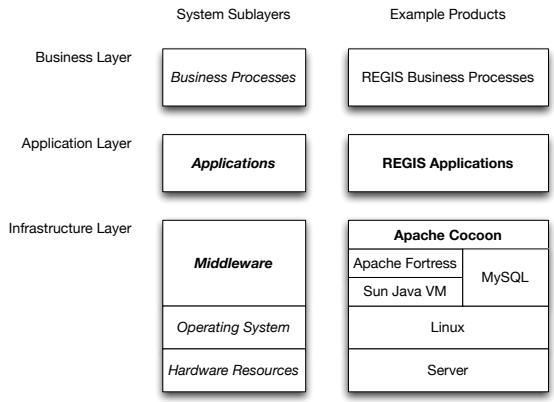

Figure 5.1: Architectural Layers and Example Technologies

5.1.2 Architectural Layers for Middleware-intensive Software Systems

The architecture of a middleware-intensive software system is considered to consist of three coarse-grained layers (cf. Section 3.1.2): business layer, application layer, and infrastructure (or technology) layer [cf. Hasselbring, 2000]. For our purposes, we split up the infrastructure layer into at least three sublayers: middleware, operating system and hardware resources (cf. Figure 5.1). We regard architectures at the application layer, which make use of primitives provided by the Middleware layer.

The distinction between the middleware and application layers depends on the project context, i.e. software components cannot be categorised into these layers outside a project context.

As an example, suppose the case of a development project for an business information system. The business information system is then on the application layer. It uses a web server, which serves as middleware for the business information system. Probably, this is the most suggesting situation. However, in a development project that develops the web server, the web server would reside on the application layer. A component platform that is used to implement the web server would then make up the middleware layer in that project. Therefore, two different views on the same software system exist: When taking an application-view on the web server, different component platforms might be evaluated with respect to their suitability for implementing the web server. On the other hand, when taking a middleware-view on the web server, it may be asked how the web server should be implemented to best support the design and implementation of a certain

type of business information system.

Middleware architecture Liu et al. [2006] use the term "middleware architecture", however they do not provide a definition, and do not use the term consistently. We deprecate the use of this term, since it is not clear to which architectural layer in the model presented above it refers. Literally, it suggests that it refers to the architecture of the middleware itself, i.e. a description of the architecture at the middleware layer. However, such a description would not be of major interest to application developers (if it were, the correct separation of concerns between the application and the middleware would be lost). The alternative interpretation is that the application architecture of a middleware-intensive system is meant, i.e. the same layer we consider in our work. Depending on the exact occurrence, Liu et al. seem to refer to either interpretation, or even a third one: An archetypal description of applications using a particular middleware. This understanding implies the assumption that all applications using a particular middleware share a similar structure. The layering notion described above is more refined, and does not require this strict assumption: Rather than assuming a common structure of applications using a middleware, it merely assumes that the concepts that make up the structure of applications are the same. These concepts are captured in architectural styles, which form the central construct of the architecture modelling approach taken in this thesis (see Chapter 7). In addition, different variants of architectural styles associated with the same middleware are allowed. In order to refine this understanding, we will introduce two new terms, middleware platform and middleware product, in Section 5.2.3.

5.2 Middleware

First, definitions of middleware in general are discussed (Section 5.2.1). After this general definition, a vertical classification of middleware is discussed (Section 5.2.2). In Section 5.2.3, we introduce the more specific terms "middleware product" and "middleware platform", which refine the single concept of "middleware".

5.2.1 Definitions of Middleware

Middleware is some piece of software. Coarsely, two views on the nature of middleware can be distinguished in the literature, which one may refer to as the *narrow* and the *wide views on middleware*. The narrow view restricts restricts middleware to distribution middleware, i.e. middleware that supports the implementation of distributed applications. The wide view, on the other hand, considers any software that is situated "in the middle"

between the application and the operating system as middleware[1]. While the operating system can be objectively identified in a computer system, the answer to what the application is depends on the point of view of the stakeholder. For example, assume there is a web-based information system, which employs a web server and a database management system. From the point of view of the vendor of the information system, the web server and the database management system are considered middleware, while the vendor of the web server views the web server as the application and thus regards the software frameworks that are used to implement the web server as middleware.

The wide view is thus *relativistic* as opposed to the narrow view, which may be considered *absolute*. The narrow view allows to discuss middleware as a distinct category of software products independently from the context of their use. Given a software product outside the context, the wide view does not allow to decide whether this product is middleware or not: In principle, any software product may be used as middleware. However, still, typical categories of middleware may be defined, which represent products that are (or could be) widely reused across organisational and project boundaries. These products are often explicitly marketed as middleware products. Most middleware products are thus standard software and sold as off-the-shelf software, but individual middleware products, for example for specific embedded systems exist as well.

For both views on middleware, the term can be explained historically: Originally, only applications and operating systems existed. Older programming languages made it difficult to introduce a reusable software layer in between. Thus applications were required to be implemented on the basis of the functional primitives provided by the operating system. Inserting a software layer in between was particularly required to make the implementation of distributed systems feasible, which added an additional degree of complexity.

Rigorous explicit definitions of middleware do not abound. Typical discussions of the nature of middleware provide characterisations of typical properties or uses of middleware. Goedicke and Zdun [2001] take the narrow view:

> "A middleware extends the platform[2] with a framework comprising components, services, and tools for the development of distributed applications. It aims at the integration, the effective development, and the flexible extensibility of the business applications."

[1] The concept of middleware thus at least implicitly implies a layered view on the software system. We will get back to this idea in Section 5.1.2.

[2] Note that "platform" refers to the computer hardware including the operating system here, as opposed to the use of the term in this thesis.

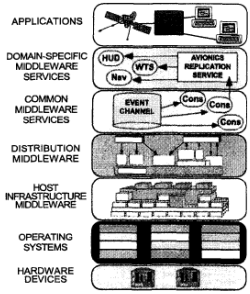

Figure 5.2: Middleware Layers [from Schmidt and Buschmann, 2003]

Emmerich [2000] also takes the narrow view, when defining:

> "Middleware [...] is layered between network operating systems and application components. Middleware resolves heterogeneity, and facilitates communication and coordination of distributed components."

It appears that the narrow view on middleware is more widely held than the wide view. The reasons are manifold, and include the commercial interest for this type of middleware, which again is due to the fact that most larger software systems are distributed[3], and the fact that its absolute character makes the distinction from other fields of research easier. However, we regard the wide view as more obvious and universal. It is held in Schmidt and Buschmann [2003]:

> "Middleware is reusable software that leverages patterns and frameworks to bridge the gap between the functional requirements of applications and the underlying operating systems, network protocol stacks, and databases."

While not explicitly mentioned it, this definition is clearly targeted towards the reusability of middleware, which was already mentioned above.

5.2.2 Middleware Platform Layers

Schmidt and Buschmann [2003] identifies four vertical middleware layers (cf. Figure 5.2): host infrastructure middleware, distribution middleware, common middleware services and domain-specific middleware services. A single middleware can span two or more of these layers. For example, typically

[3] Still, one needs to carefully distinguish between software systems that are distributed themselves, and software systems that are non-distributed but used in a distributed context.

an off-the-shelf distribution middleware also includes higher-level common middleware services. *Host infrastructure middleware* is confined to a single, i.e. non-distributed, computer system, and comprises virtual machines and enhanced mechanisms for implementing concurrent systems. *Distribution middleware* builds upon host infrastructure middleware and provides basic mechanisms for realising distributed systems. More complex distribution services are provided by the *common middleware services* layer. Taking Java EE as an example for the common middleware services layer, the corresponding middleware on the distribution middleware layer is the Java Remote Method invocation technology, and the Java Virtual Machine on the host infrastructure middleware layer. A concrete software system might employ additional middlewares on each of these layers. On the top layer, *domain-specific middleware services* are provided.

The model provided by Schmidt and Buschmann can be criticised since it assumes that domain-specific services necessarily build upon distribution aspects. Obviously, this is not always the case. For our purposes, such a strictly layered model is not entirely adequate. However, we do not see the need to provide a rigorous alternative, and will just discuss this issue for the concrete case studies in Chapter 10.

5.2.3 Middleware Platforms and Products

As noted above, we distinguish middleware products and middleware platforms. Colloquially said, one can buy (or download) a middleware product, but only in use it becomes a middleware platform. Since there are many ways to use a complex middleware product, there may be many platforms associated with a product. On the other hand, since a middleware platforms is an abstraction of the actual use, it may apply to multiple middleware products.

As Almeida et al. [2004] argue, architectural style is part of the relevant characteristics of an abstract platform (cf. Section 3.1.2.1). We focus on the style aspect in our work, since it is not our goal to provide automatic model transformations. However, with this restriction, the notion of an abstract platform is close to our understanding of a middleware platform.

The idea of abstract and concrete platforms is not necessarily bound to middleware products, but we consider only those that are related to middleware products directly or indirectly. In a series of MDA transformations (cf. Section 3.1.2.1), the final PSM is the implementation of the system, and its (concrete) platform corresponds to the actual middleware product that is used. The abstract platform, which we term middleware platform, represents the architecturally relevant characteristics of that middleware product and its usage. Therefore, the abstract platform (middleware platform) is an abstraction of the concrete platform (middleware product), but it also carries additional information: On the one hand, the uniform usage of the concrete

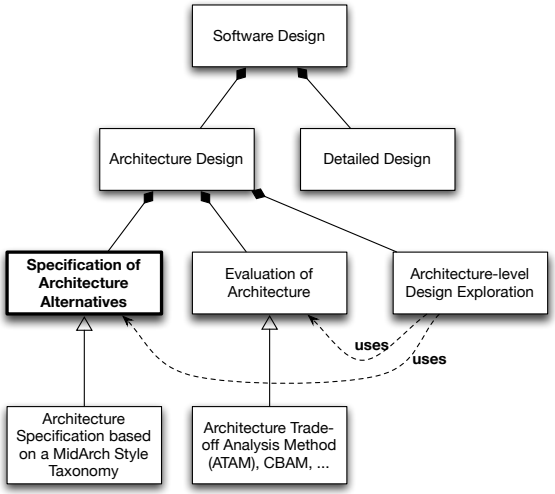

Figure 5.3: Activities in (Software) Architecture Design

platform is part of the abstract platform and implicit in the PIM, but it is dissolved into the concrete uses in the PSM. On the other hand, it captures only architecturally relevant aspects of the concrete platform, but not its internal implementation details.

5.3 Architecture-level Design Exploration

Figure 5.3 shows a hierarchy of activities in software design, which ought to illustrate the concept of architecture-level design exploration (in short: architecture exploration). Design exploration, also design space exploration, [cf. Bondarev et al., 2006] is a coordinated effort that involves the specification and evaluation of multiple architectural alternatives for a given system specification. Its goal is to bring insight into a representative set of points within the relevant portion of the design space that contains solutions for a given problem. Thus, beside the need for suitable techniques for (a) the specification of architectural alternatives on the one hand, and for suitable techniques for (b) evaluating architectures on the other hand, an overall method for architecture-level design exploration requires some techniques for (c) identifying such a *representative* set of points within the *relevant* portion of the design space.

The relevant portion of design space is made up by those architectural alternatives that provide an acceptable satisfaction of the stated requirements.

Any architecture-level design exploration technique involves a heuristic that judges which portion of the design space can be reasonably expected to contain acceptable architectural alternatives based on the existing architectural design knowledge. The quality of a architecture-level design exploration technique in this respect is thus determined by the degree to which the heuristic identifies this portion accurately, i.e. such that it does not contain bad solutions.

From this portion of the design space, a set of architectural alternatives must be identified, which should be representative in the sense that they are sufficiently distinct so that they represent different trade-offs to be made between the stated requirements. As [Jansen and Bosch, 2006] put it, "explicit design space exploration helps the architect in preventing from making obvious mistakes."

This thesis will not provide a contribution to part (b) of the problem, but relies on existing architecture evaluation methods. The contribution to part (a) and (c) on the problem builds upon the concept of middleware-oriented architectural styles: We assume that the component-and-connector structure of the systems we address is heavily influenced by the chosen middleware platform, and different middleware platforms yield architectural alternatives with distinct quality properties. Furthermore, we assume that middleware-oriented architectural styles are a suitable means to describe middleware platforms. Thus, we conclude that a representative set of points within a given portion of the design space can be enumerated by specifying architectures for different middleware-oriented architectural styles that correspond to that portion of the design space. Thus, we propose a method for specifying software architectures based on middleware-oriented architectural styles as a technique for part (a), which is described in Chapter 7. The identification of relevant styles is addressed by the overall MidArch Design Method, which uses middleware-oriented style taxonomies. It is described in Chapter 8.

Both techniques rely on a method for specifying middleware-oriented architectural styles, which is closely linked to both the specification of style-based architectures as well as the description of middleware-oriented style taxonomies. These methods are described collectively in Chapter 7.

5.4 Formalisation

We specify central concepts of the MidArch approach using an Object-Z specification [Smith, 2000]. Object-Z is an object-oriented extension of the formal specification language Z, which is based on axiomatic set theory, lambda calculus, and first-order predicate logic.

Figure 5.4 shows the structure and relationships of the specification packages (i.e. sections of the thesis that contain Object-Z specification fragments). In the "basics" packages, only the external properties of the relevant entities

are defined, i.e. not their internal structure. These definitions are refined using a MOF metamodel in Section 7.6.1, since this is better suited to reflect the focus on a large number of meta-classes and their meta-relationships, which would render an equivalent Object-Z specification hard to oversee.

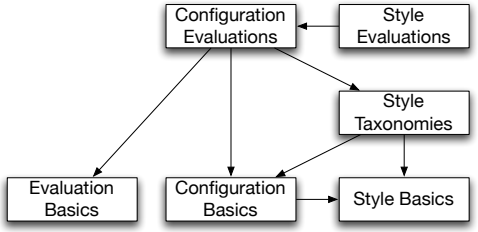

Figure 5.4: Object-Z Specification Package Structure

5.4.1 General Definitions

First, we define an abstract class for entities:

Entity

5.4.2 Style Basics

This package contains just the elements that are required for the definition of both MidArch Style Taxonomies and basics of configurations, as well as for defining the inner structure of MidArch Styles. This is just the declaration of the free set *Style*.

$[Style]$

5.4.3 Configuration Basics

A *Configuration* is a configuration in the logical component structure viewpoint. The MidArch Metamodel (Section 7.6.1) specifies what the internal structure of a configuration is. At this point, only the relationship between a configuration and MidArch Styles is relevant: A *Configuration* declares a style and it may conform to one or more styles. We assume here that every

configuration declares a style. Configurations that do not declare a style are not relevant for our considerations. In general, it is not guaranteed that a configuration conforms to the style it declares. However, at this level of abstraction, we are only concerned with valid configurations for which this is the case.

Configuration

Entity

$$declaresStyle : Style$$
$$conformsToStyle : \mathbb{P}\, Style$$
$$declaresStyle \in conformsToStyle$$

5.4.4 Evaluation Basics

We formally define *Metrics*, which are applied to *MeasureableEntites* in order to obtain *Measurements* (cf. Section 2.2.3). Measureable entities most importantly include measurable configurations which are defined in Section 5.4.5. A metric has a scale, which defines the values its measurements can take and the relationships between these values. The function *measure* describes the relationship between metrics, measurable entities and measurements.

$$[Metric, Measurement]$$
$$MetricMeasurement_1 == Metric \times Measurement$$
$$MetricMeasurement_2 == Metric \times Measurement \times Measurement$$

MeasurableEntity

Entity

$$scale : Metric \rightarrow \mathbb{P}\, Measurement$$
$$measure : Metric \times \downarrow MeasurableEntity \rightarrow Measurement$$
$$\forall\, m : Metric \bullet \forall\, e : \downarrow MeasurableEntity \bullet measure(m, e) \in scale(m)$$
$$\forall\, m_1, m_2 : Metric \bullet scale(m_1) \cap scale(m_2) \neq \varnothing \Rightarrow scale(m_1) = scale(m_2)$$

A quality model organises a set of metrics. An operation *measure* is defined for a quality model to apply all metrics of the quality model to a measureable entity. At this point, we only define a flat projection of a quality model, since for performing an evaluation only the set of metrics defined by a quality model is relevant. The structure of the quality model in terms of goals,

questions and metrics is discussed in Section 2.4.

QualityModel

$$metrics : \mathbb{F}\ Metric$$

measure

$$me? : \downarrow MeasurableEntity$$
$$measurements! : \mathbb{F}\ MetricMeasurement_1$$

$$\forall\, m_1 : Metric \mid m \in metrics \bullet \exists (m_2, mm) \in measurements! \bullet (m_1 = m_2 \land mm = measure(m_1, me?))$$

5.4.5 Configuration Evaluations

This package is very small and just defines a common subclass of *Configuration* and *MeasurableEntity* to define measurable configurations. It is provided as an auxiliary package to avoid mixing the definitions referring to configurations and measurements.

MeasurableConfiguration

$$ConfigurationMeasurableEntity$$

5.5 Design Knowledge

In this section, we discuss the roles design knowledge plays in this thesis and the methods proposed within it.

Design knowledge occurs in three roles here: acquisition, reuse and management of design knowledge. Design knowledge in the MidArch Design Method manifests in different types of artefacts, according to the classification by Jones [1994] (cited after Frakes and Terry [1996, Table 1]). There are ten types of reusable artefacts in software projects, which include source code, architectural styles, and designs. Frakes and Terry [1996] additionally consider processes and knowledge as reusable. Since these are usually intangible aspects of software development, their reuse is hard to trace.

An overview of the roles and the associated MidArch activities and artefacts is given in Figure 5.5. They are discussed in detail in the following. The discussion anticipates the presentation of the MidArch Design Method in Chapter 8 in part, but is given here nonetheless, in order to provide a framework for the following chapters.

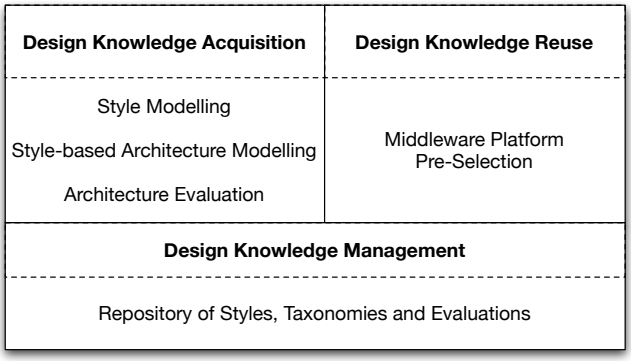

Figure 5.5: Roles of Design Knowledge

5.5.1 Design Knowledge Acquisition

Design knowledge is *acquired*, produced or generated (the latter term is close to the terminology of design analysis) in the form of style descriptions, style taxonomies, architecture evaluation results, and evaluation artefacts derived from architecture evaluation results and style taxonomies. Three tasks acquire design knowledge within the MidArch Design Method:

1. *Style modelling* generates style descriptions as well as style taxonomies.

2. *Style-based architecture modelling* generates architectures based on those styles.

3. These first two tasks are not directly in the focus of our attention, but they provide the input for *architecture evaluations*. These evaluations generate evaluation results related to individual architectures at first, but by comparing evaluation results of different individual architectures, style-related evaluation results are yielded. These evaluation results represent knowledge on the influence of the corresponding platform on the system quality (see also Section 5.6).

5.5.2 Design Knowledge Reuse

Design knowledge is *reused* primarily in the middleware platform selection task. The artefacts used in the task are the style-related evaluation results as well as style taxonomies. The style descriptions themselves are not used in the task, at least not in a formal sense. But reuse of style descriptions occurs in style-based architecture modelling, and partly in the modelling of styles, if a substyle of an existing style is modelled.

According to the definition of Frakes and Terry [1996], also reuse of the method definitions happens throughout the application of the method. The metamodels for styles and style-based architectures are also part of the definition of the MidArch Design Method.

Since reuse incurs an overhead on the software design process, not every form of reuse will lead to a cost reduction. A method promoting reuse might thus be criticised for incurring a greater overhead than the benefit it achieves. We will discuss current theoretical and empirical finding on the beneficiality of specific modes of reuse. In the discussion, we will argue that our method indeed implements a mode of reuse that can be expected to be beneficial to cost reduction.

As an alternative to "reuse", one could also speak of a "transfer" of design knowledge (between MidArch instances), or simply of the use of design knowledge. It may be argued, that in fact "use" would be more accurate, since the first "use" of a design knowledge artefact cannot already be called "reuse". However, we choose the term "reuse" nonetheless, since the benefits of the approach are only achieved when the same design knowledge artefacts are reused in multiple MidArch instances.

Literature on Reuse There has been some work on the link between reuse and productivity [Basili et al., 1996; Browne et al., 1990; Card et al., 1986; Chen and Lee, 1993; Gaffney and Durek, 1989; Lim, 1994; Nada and Rine, 2000], and on the link between reuse and software quality [Agresti and Evanco, 1992; Browne et al., 1990; Card et al., 1986; Chen and Lee, 1993; Gaffney and Durek, 1989; Lim, 1994; Mohagheghi et al., 2004; Nada and Rine, 2000]. Maturity models similar to the CMMI but specific to reuse have also been developed [Davis, 1993; Koltun and Hudson, 1991], but empirical validation of these models is also rare. There is also some work on the question which variants of reuse are most effective [Nazareth and Rothenberger, 2004; Selby, 2005].

For our work, we employ the following assumption, which is substantiated by Nazareth and Rothenberger [2004]; Selby [2005], even if it does not exactly apply to the same type of reusable artefacts that we use:

Assumption 5.2. Increased reuse causes an increase in software process efficiency and predictability, which contributes to increased product quality.

Since software development in general and reuse in particular are very complex activities, they may also be applied adversely, i.e. if they are used thoughtlessly, they may have a negative effect. Certain conditions in implementing reuse must thus be considered [Morisio et al., 2002].

Architectural Styles and Reuse Reuse at the architectural level is beneficial to support manageability of reuse.

The use of off-the-shelf middleware products, such as distributed component platforms, supports reuse at the architectural but are not sufficient to provide an effective reuse process. They provide an environment for component assembly [Fuentes et al., 2002; Szyperski, 2002], but usually do not provide guidance for specifying systems pertaining to specified quality requirements, or for analysing systems [Fuentes et al., 2002].

Application frameworks that specialise component platforms are a supporting element for architectural reuse [cf. Demeyer et al., 1997], but their mere implementation is not sufficient for their effective reuse [Fuentes et al., 2002]. It is a hard task to document them in a form that new developers are guided to using them correctly [Ploski and Giesecke, 2005; Sparks et al., 1996]. Architecture-level documentation could be one step towards good documentation of frameworks, providing guided access to lower-level documentation. Architectural styles are one form of such architecture-level documentation.

5.5.3 Design Knowledge Management

The design knowledge management role serves as an enabler and provides the common basis for the other two roles. It is implemented through the MidArch Repository, which stores styles, taxonomies of styles and evaluations of architectures and styles. Acquired design knowledge artefacts are put into the repository, and reused by querying the repository.

5.5.4 Summary

Within a given instance of the MidArch Design Method, the management role occurs necessarily, and either or both of the acquisition and reuse roles occur as well. However, the tasks within the MidArch Design Method are partitioned such that each can be uniquely associated with either the acquisition or reuse role, with the management role as a supporting role. This association will be discussed in detail for each task in Section A.3.

5.6 Software Development Methods and System Quality

The relationship of software development methods and system quality is very complex, which makes it difficult to predict and assess. To a large extent, software engineering research relies on assumptions shared by many researchers and/or practitioners rather than empirical evidence. The assumptions that build the basis for the claim that the approach presented in this thesis is sound in this respect are discussed in this section. In Section 9.3, we will discuss how the proposed method could be subject to a comprehensive empirical validation in principle.

The discussion in this section refines that of the introduction (Chapter 1).

5.6.1 Effectiveness of Software Development Methods

Grossly simplified, the goal of software engineering is to produce high quality software systems (benefit) under a low consumption of resources (cost). A software system is developed through a software development process. Conceptually, a software process is a series of concrete occurrences of activities. It does not represent their abstract description, which we refer to as the software development method. A software process may exist independently of a software development method (informal software process), or it may be an *instance* of a method (formal software process). It might be based only loosely on a specific software development method, while not formally following it. Similar to the assurance of conformance of a system to its architecture, the conformance of a process to its method may be addressed.

A cost-benefit tradeoff thus needs to be made. However, only points that are non-dominated with respect to cost vs. benefit in the universe of software development processes are of interest, i.e. those points that are not dominated by any points with either higher benefit at the same cost or with lesser cost at the same benefit. The software engineering challenge is to define generic software development methods, such that for a designated subset of the problem domain their instances are (a) non-dominated and (b) fulfil certain stakeholder-defined subsidiary conditions *(Subsidiary application condition)*. Examples for factors considered in subsidiary application conditions may be budgetary or organisational constraints. A method can be said to be *effective* for a problem if both application conditions are fulfilled, and the set of those problems can be called the *effectiveness set* of the method. It is not sufficient that the effectiveness set is non-empty, but it must also be decidable in advance. From a practical point of view, approximations to these conditions will suffice, i.e. a certain uncertainty in the decidability of the effectiveness set and a restricted fulfilment of the application conditions are acceptable. On a large scale, the state of software engineering can be measured through the degree of availability of effective methods covering the economically relevant subset of problems.

Formalisation We now formalise this matter. First, we define:

- the set of requirements specifications R,
- the set of software systems S,
- the set of software development processes P,
- the implementation function $im : R \to \mathcal{P}(S)$.

A software system is the *outcome* of a software development process, determined through $o : P \to S$. *Cost* is a function $c : P \to \mathbb{R}$, while *benefit* is a function $b : R \times S \to \mathbb{R}$. For a given requirements specification $r \in R$, only

those processes p are relevant that yield systems fulfilling those requirements, i.e. $o(p) = im(r)$.

For a problem $r \in R$, a software development process $p \in P$ with $o(p) = im(r)$ is *non-dominated* if

$$\forall\, p' \in P : o(p') \in im(r) \Rightarrow c(p) \leq c(p') \vee b(r, o(p)) \geq b(r, o(p'))$$

A non-dominated process does not need to be unique, but if there are multiple, their cost and benefit are the same.

A *software development method* is a (partial) function $m : R \to P$, i.e. it maps a problem onto a process.

Given a software development method m and $C \subset P$ such that $\chi_C : P \to \{0, 1\}$[4] is a *subsidiary application condition function*, the *effective set* of m w.r.t. C is

$$\mathrm{eff}_{\chi_C}(m) := \{r \in R \mid m(r) \text{ non-dominated } \wedge\, \chi_C(m(r)) = 1\}$$

The quality of a software development method for a given problem can be measured by the degree to which the non-dominance condition and the subsidiary application condition are fulfilled.

We first define the degree of non-dominance by relating a process to the non-dominated processes it is dominated by. Let $r \in R$ be a problem and $p \in P$ such that $o(p) \in im(r)$. Then there are non-dominated processes $p', p'' \in P$ such that $c(p') = c(p)$ and $b(r, o(p)) = b(r, o(p''))$. The *degree of non-dominance* of p for r is then defined by

$$\mathrm{dPar}(r, p) := \frac{c(p'')}{c(p)} \cdot \frac{b(r, o(p))}{b(r, o(p'))}$$

We replace the binary subsidiary application function by a continuous function $\chi_C : P \to [0, 1]$ with $\chi_C(C) = \{1\}$.

Then, a *weak effectiveness set* may then be defined using thresholds for the quality values:

$$\mathrm{eff}_{\chi_C, t_1, t_2}(m) := \{r \in R \mid \mathrm{dPar}(r, m(r)) \geq t_1 \wedge \chi_C(m(r)) \geq t_2\}$$

By this definition, a true generalisation of eff_{χ_C} is yielded, i.e. $\mathrm{eff}_{\chi_C, 1, 1}(m) = \mathrm{eff}_{\chi_C}$.

Simplifications We assumed that software development methods are totally well-defined, i.e. the method uniquely prescribes one software development process for a given problem. Later, we will weaken this assumption and consider that applying a software development method to a given problem may yield various processes and thus various systems.

[4] We use the convention that for a set S, χ_S is the characteristic function of S.

Figure 5.6: Quality Impact Chain

The production of internally reusable artefacts is not explicitly considered in this model, as are any other effects across projects.

Furthermore, while cost may be easily regarded a one-dimensional concept, benefit is usually multi-dimensional. It can only reduced to a one-dimensional concept through the incorporation of subjective preferences that trade off the different dimensions. Such subjective preferences may not be stable over time. A generic benefit function needs to consider as parameters at least (a) either the problem or a quality model together with a set of required performance levels and (b) a model of the preferences. We will address the multi-dimensionality of benefit explicitly for concrete development methods.

5.6.2 Quality Impact Chain

As shown in Figure 5.6, we basically assume that there is an impact of (architecture-level) design reuse on the quality of the architecture (cf. Section 2.3.2), and an impact of the quality of the architecture on the quality of the resulting system. The relationship of these aspects is not monotonous, i.e. it is not the case that any increase in design reuse necessarily improves the quality of the architecture, and it it is not the case either that any improvement of architectural quality results in an improvement of the system quality. In fact, there is supposedly no such step to be made that guarantees a certain degree of system quality. Any architecture can be implemented in a way that the resulting system does not fulfil the requirements. The figure simplifies the overall situation to a large extent, as other influences on the qualities are deliberately omitted. For example, other kinds of reuse that might directly influence system quality are left out. However, certain measures can contribute to avoiding that the system resulting from a software development effort does not fulfil its requirements.

We make two assumptions, under which this condition is expected to hold:

1. First, the mapping of the degree of well-definedness of the architecture development method to the volatility of the architecture quality is assumed to be monotonically non-increasing. This assumption is also made (and has been validated to some extent) by process maturity models such as the CMMI [Chrissis et al., 2003]: The “managed” maturity level represents an improvement over the “initial” maturity level in that it requires a successful repeatability of a process (not necessarily an equivalent repeatability). It is not necessarily the case that the mapping

to the overall quality is monotonically non-decreasing: One can easily imagine a (even nontrivial) development method that produces bad solutions by systematically neglecting or violating software engineering principles such as the cohesion and coherence decomposition principle.

2. Second, we require that conformance of the implementation with the architecture can be checked. In general, this is not the case: Even when a formal representation of the architecture exists, no conclusions on the consequences for the implementation can be drawn from it. We consider conceptual architectures (cf. Section 3.3.4), which by definition are not necessarily homomorphic to the structure of the implementation. Thus, a procedure for checking the conformance of the implementation to the architecture needs to make additional assumptions on their relationship. It is beneficial, but not necessary, to use an explicit specification of the mapping of the architecture to the structural concepts provided by the implementation platform, since this allows a constructive transition from the architecture to the implementation. A conformance checking procedure can then be deduced from the mapping or it may use the mapping as an input.

In this context, it is important to note that an architecture is not a specification of the implementation, but a model of the implementation. Both the architecture and the implementation are part of the solution domain, not of the problem domain. Thus, checking the implementation for conformance with the architecture does not allow to draw any conclusions regarding the correctness of the implementation.

On the way from the architecture to the implementation, additional (design) decisions must be taken, which influence the quality properties of the resulting system. The lack of knowledge (ignorance) resulting from the freedoms of choice in this process cannot be eliminated in advance. However, it may be explicitly considered in models of the software process in the form of the volatility of the system quality for a given architecture.

Formalisation We define the set of architecture descriptions A and the set of implementations I. A software system is associated with an architecture description and an implementation (cf. Section 3.1.1), which are determined by functions $ar : S \to A$ and $im : S \to I$. Multiple distinct software systems may have the same architecture description.

On S we define an equivalence relation $\equiv_A$ such that for $s_1, s_2 \in S$:

$$s_1 \equiv_A s_2 \Leftrightarrow ar(s_1) = ar(s_2)$$

We now extend the notion of a software development method to a non-deterministic one. A software development method is then a function $m : R \to \mathcal{P}(P)$. An architecture development method m_A is an equivalence class

on the set of software development methods based on $\equiv_A$ on S:

$$m_1 \equiv_A m_2 \Leftrightarrow$$
$$(\forall\, r \in \operatorname{dom} m_1 \cap \operatorname{dom} m_2 : \{[o(p)]_A \mid p \in m_1(r)\} = \{[o(p)] \mid p \in m_2(r)\})$$

The volatility of an architecture development method is measured through the number of architecture descriptions that may be produced by it:

$$vol(R, m_A) := |ar(m_A(R))|$$

Analogously, given an architecture description, the volatility of a software development method regarding the implementation is measured through the number of implementations it may produce that adhere to that architectural description:

$$vol(R, m, A) := |im(m(R))|$$

Simplifications One could decompose the software development process into its task instances, and then consider the modifying effect of each task instance on the architecture description and the implementation. Based on this, the overall software development process could be partitioned into an architecture development process and an implementation process if one required that every single task instance modifies only either of the architecture description and the implementation. This would also allow a refined definition of distinct architecture development methods and implementation methods.

While we introduced non-well-defined software development methods, we neglected that still different resulting processes may have different probabilities of occurrence.

5.7 Architectural Styles and the ISO 42010 Reference Model

In this chapter, we extend the reference model for architectural descriptions as defined in ISO Standard 42010 to incorporate the notion of architectural styles.[5] The motivation is to provide a more rigorous characterisation of architectural styles within an accepted conceptual framework for architectural descriptions (cf. Section 3.4). The reference model from the standard provides the most elaborate, widely applicable reference model for our purpose. The reference model is very abstract and does not prescribe any concrete metamodel for constructing software architecture description artefacts. The metamodel for describing architectural styles and style-based software architecture within the MidArch approach is presented in the next chapter (Chapter 7), which applies or refines the abstract reference model from this chapter.

[5] This chapter is based to a large extent on a published paper [Giesecke et al., 2006].

This chapter will first discuss some characteristics of the reference model as given in the standard (Section 5.7.1). Then, we present the extension of the standard in Section 5.7.2.

5.7.1 Characteristics of the Reference Model

The standard provides a well-founded conceptual reference model for software architecture as well as a definition of software architecture. However, the standard itself notes that this model is not authoritative with respect to the cardinalities of the relationships of the represented concepts, but should merely serve as an illustration of the standard's intentions for a single architectural description.

Generic elements An exception to this general rule is the Library Viewpoint concept, since it explicitly refers to entities that are not confined to a single architectural description, but are reused across projects.

Description elements The standard does not explicitly make assumptions on the form in which the architectural description is represented, other than that the primary information is contained in the models (rather than the views, e.g., which are only conceptual entities that organise information contained in the models). It remains unspecified which description elements other than the models form part of an architectural description. In other words, it is unclear how to map the parts of a concrete architectural description to the conceptual elements provided by the reference model. While this is possible for the single architectural models, it is impossible for description elements that are not constrained to a single model: for example, the metainformation on which viewpoints have been chosen or which models are present are not part of any model, but have to be found elsewhere. It is not the intention of a standard to provide some logical or physical structure for an architecture description, but it should still be possible to use the vocabulary defined by the standard to refer to the elements of such structures defined by third parties.

In the original standard, architectural rationale is defined to comprise "the rationale for the architectural concepts selected" and "evidence of the consideration of alternative architectural concepts and the rationale for the choices made".

Conclusion With these restrictions in mind, we still consider the standard's reference model useful to provide a basis for presenting our extensions. It is important to note that the purpose of the reference model is not to provide a meta-model for instantiating architectural descriptions, neither manually nor tool-supported, but to provide the vocabulary necessary for communicating

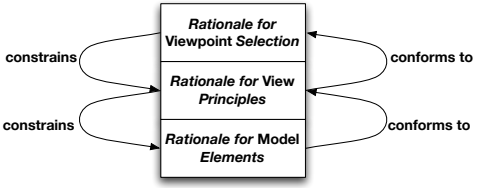

Figure 5.7: Layers of Architectural Description Rationale

about architectural description concepts and for defining specific architectural description methods.

5.7.2 Extension of the Standard for Architectural Style

In accordance with the rationale underlying the standard itself, we propose a lightweight extension of the standard, which should be general enough to be applicable to any architectural description practise that can be mapped onto the original standard. We introduce rationale documentation for each of the central concepts: viewpoints, views and models. While that view is not explicitly held in the standard, we assume these three concepts can be regarded as three layers of architectural description in the stated order.

In an analogous hierarchy, architectural rationale can be structured along the same layers, which yields the layers shown in Figure 5.7:

Rationale for Viewpoint Selection As the standard already notes, part of the architectural rationale is the documentation of the reasons for choosing the set of viewpoints used.

The standard already introduces the notion of a library viewpoint, which is a viewpoint that is typically used in documenting software architectures in a particular application domain, for example. However, even more useful are collections of viewpoints that often occur together and provide a meaningful coverage of typically relevant architectural concerns. One example for such a set of viewpoints are those defined in the Reference Model for Open Distributed Processing (RM-ODP) (ISO/IEC Standard 10746-1).

Rationale for View Principles Next, on the level of views, the principles for organising the contents of the view must be determined. Important means, especially for structurally oriented views, are *architectural styles*. Analogously to the distinction of ArchitectureandArchitecturalDescription made by the standard, we distinguish a Style (as a conceptual entity) from a StyleDescription (a representation of the former conceptual entity). The question whether multiple styles can be combined or if they may overlap

within a model remains unresolved [Clements et al., 2002], so we do not explicitly introduce a restriction in this direction into our model. When applying the reference model, it must be decided whether such a restriction should be made.

Styles may be taken from a library associated with the viewpoint, i.e. a viewpoint *proposes* certain styles, and thus be intended to be reused across different projects. Alternatively, it may be specifically modelled for the project at hand.

A style establishes principles for modelling within the view, but does not yet instantiate any concrete elements, e.g., the choice for a pipes-and-filters style does not instantiate any concrete pipe or filter, which is a separate design activity. This is different than for design patterns: E.g., when choosing the Singleton pattern [Gamma et al., 1995], the choice of the Singleton pattern and the identification of the class which should be made a singleton cannot be separated but form a single atomic design decision. However, for a style, patterns may be defined that can be instantiated once the style has been selected.

Rationale for Model Elements Concrete elements representing system entities are represented in the models of an architectural description. The documentation of reasons for choosing certain elements over others are thus documented on this rationale level. This documentation cannot always be associated with single elements of a model, sometimes multiple elements must be seen in context, for example when instantiating patterns associated with a style [Garlan, 1995].

Each of the architectural rationale parts is thought to be a description element on its own. They thus accompany the primary description elements (the models). We explicitly introduce the elements of the architectural rationale as description elements that carry information on their own.

Justification It might seem natural to model ReferenceArchitecture as a specialisation of architecture. However, we decided against this for two reasons: First, specialisation relationships are not considered at all in the original standard, and so the introduction for this class would impair the coherence of the reference model. Second, an architecture is associated with a single system, which is obviously not the case for a reference architecture, which make only sense if it is applied to multiple systems. Adapting the standard to restrain from this problem would require several further changes to the standard, which we chose not to make.

As indicated above, we associate a StyleDescription with a View. This is a deliberate design decision, and its alternatives were to associate a style with either a Model or with the ArchitecturalDescription as a whole. We decided not to associate a style with a model, because a

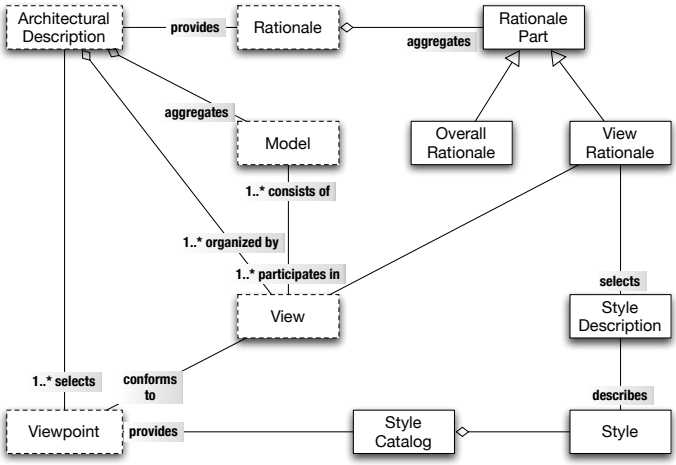

Figure 5.8: Extension of the ISO 42010 Reference Model

style governs a decomposition of the system which may be referenced in multiple models. Furthermore, we decided against associating a style with the architectural description as a whole, since different decompositions of a system (e.g. its run-time decomposition into process components vs. its design-time decomposition into module components) may be governed by different styles with no immediate relationship.

6 Taxonomy of Architectural Style Usage

In this section, we present a taxonomy of usages of architectural styles, i.e. modes of using architectural styles within formal and informal software processes (see Section 5.6.1). We initially studied the literature for these usages to gain a better understanding on the current state of research and practise in using architectural styles, and then categorised our own method within the taxonomy.

First, we discuss different types of codified architectural design knowledge, which was used as a basis for creating the taxonomy (Section 6.1). The details of the approach taken are presented in Section 6.2. Section 6.3 then elaborates the taxonomy resulting from conducting this procedure.

6.1 Types of Codified Architectural Design Knowledge

Reuse has long been an important aspect of software engineering, and it comes in many forms, e.g. code or design reuse [Krueger, 1992]. With the rise of software architecture as a discipline [Shaw and Garlan, 1996], reuse has also been elevated to an architectural level: Component-based software engineering approaches specifically target the reuse of executable units, i.e. components. However, interoperability problems may hinder the composition of components into an architecture [Compare et al., 1999; Davis et al., 2002]. Therefore, other approaches to reuse at the architectural level are still required to enable component reuse.

A component is highly tangible in a technical sense, since it is assumed to come in a form that is almost complete described in formal languages. Part of this description is the description of the component interface. Such descriptions are the object of research on elaborate component interface definition languages. Industrially relevant examples are OMG IDL [ISO/IEC, 1999] and the WSDL [W3C, 2007]. Other kinds of architectural knowledge, such as patterns, reference architecture or guidelines, are represented in forms that are usually more human-oriented at least in some part (documentation), while other parts (models, specifications, example implementations, ...) may be written in a formal language as well. Architectural knowledge may be available only implicitly, or explicitly in either free natural language or in some codified, i.e. standardised explicit, form. Examples of codification templates are the various pattern template (Coplien form [Coplien, 1996b], Gang of Four form [Gamma et al., 1995], ...), as well as architectural style specifications in ADLs such as ACME [Garlan et al., 1997].

We regard the notion of *architectural styles* as one of the most interesting, but perhaps also most often misused or misinterpreted concepts in the area of codified architectural design knowledge.

The idea of architectural styles more or less developed within the ADL community, which follows a more formally-oriented perspective on software architecture. In parallel, the idea of design patterns (in the following, we will use the term "pattern" in brief) evolved and a pattern community grew. This community follows a more pragmatic perspective on software architecture and therefore consists of software development practitioners to a larger degree. The ADL community, on the other hand, is dominated by academic and industrial software engineering researchers.

Perhaps due to their pragmatic perspective, some proponents of the pattern community claim that styles may be subsumed by the idea of patterns [Buschmann et al., 1996]. This is true to some degree, as already noted by Monroe et al. [1997], but captures only a specific usage and requires a very broad interpretation of the pattern concept. Intriguingly, on the one hand, the pattern community seems to have very specific ideas what patterns are and what role they play in software development, but on the other hand they are unable to give a rigorous definition of a pattern, which would enable deciding if a given artifact is a pattern or not [van Emde Boas, 1997]. The discussions reduce to the question of syntactic representation and relationships of patterns. These paths do not really help in deciding which artifacts are patterns, and which are not: If the descriptions contain natural language, which is an essential part of the pattern descriptions in many cases, and merely a coarse-grained structure for the text is prescribed, arbitrary concepts may be put into the form of a pattern. Thus, the form cannot suffice for deciding whether something is a pattern or not. There has been some work on the formalisation of pattern descriptions [Raje and Chinnasamy, 2001], but this is somewhat decoupled from the mainstream pattern community.

Since architectural styles and patterns are not usually distinguished consistently (cf. Section 3.4.3.1), we considered both as nearly equivalent during creating the taxonomy. In this chapter, we will use the terms "design patterns" for lower-level artifacts, and "architectural styles" for higher-level artifacts, except when explicitly discussing differences between both concepts.

6.2 Approach

We develop a taxonomy of architectural styles based on the *usages*, i.e. the manners of their use, within software engineering. This taxonomy is more or less orthogonal to the *representation* of architectural styles (e.g., informally, using an ADL or as graph grammars). However, the specific form of representation may make the style more or less apt to a specific mode of

use. We will ignore this issue for the moment and will return to it after the discussion of the modes of use in Section 6.3.7.

We analysed the literature on architectural styles and patterns and identified several clusters of modes of use. We will refer to these clusters simply as *usage* in the following for better readability. We first describe and compare these modes of use in a bottom-up manner in Section 6.3. As part of this, we list the sources that refer to each of the modes of use. Based on interesting characteristics identified within this phase, we provide a top-down categorisation in Section 6.3.7.

6.3 Usage-based Taxonomy of Architectural Styles

Based on the study of the available literature, we identified the following modes of use of architectural styles:

1. Ad-hoc
2. Platform-oriented
3. Customised
4. Pre-modelling
5. Post-documentation and analysis

Additionally, we discuss the relationship of these uses to the use of generic architectures. These are units of architectural knowledge which are not exactly architectural styles in the sense of our working definition (see Section 3.4.2), but borderline cases might fall into the definition. We elaborate on these modes of use in the following subsections.

6.3.1 Ad-hoc Use of Styles

As already mentioned above, sometimes the style concept is subsumed under the pattern concept (see, e.g., Buschmann et al., 1996; Kirchner and Jain, 2004; Schmidt et al., 2000). In this view, every style may be expressed as a design pattern, but not necessarily vice versa. This view may also be referred to as the *styles-as-patterns view*. These authors conceive to an essentially ad-hoc usage of styles: The consultation of a pattern collection and the selection of a pattern to apply remains at the discretion of the architects on the basis of their experience. Collections of patterns specify different kinds of relationships between multiple patterns [see, e.g., Zimmer, 1995], but the relationships that refer to their usage cannot be made explicit in general. Thus, using patterns/styles in this way is difficult, among other things, to teach or convey.

In principle, everything that has been said about using patterns also applies to styles in this view [Riehle and Züllighoven, 1996; Schmidt et al., 1996]. However, since not every design pattern is viewed as a style even by the proponents of this view, some additional remarks regarding the use of "typical" styles-as-patterns are necessary. The relatively simplistic general-purpose styles originally described by Shaw and Garlan [1996], for example, are criticised as being impossible to apply to a system as a whole [Shaw, 1995b]. Indeed, this is a reflection and manifestation of the ad-hoc approach to using styles-as-patterns: styles are not assumed to guide the following architectural development, but they are reduced to communication styles between two or more system elements. For any two elements that should communicate with each other, a different communication style might be chosen. This contradicts the original view of a style as embodying a *decomposition principle*: the idea was that different styles led to different system decompositions [Garlan and Shaw, 1993]. On the contrary, the styles-as-patterns view tends to view the decomposition of a software system to be relatively independent from the styles employed and the style just influences the interaction between pre-established, i.e. established prior to the selection of the style, elements. While this view masks several aspects of architectural styles deemed relevant by other authors, such as style-specific analyses, the scenario is of major practical relevance when re-engineering an existing system or building a new system of existing (COTS or not) components. The current state of the architecture may be very heterogeneous, which is the ultimate reason for re-engineering it, but if may be infeasible as well to provide a coherent target architecture which follows a single architectural style, e.g. because some components are considered intangible because they embody knowledge that cannot be safely restored. It is in principle always possible to define an overlay architecture that is coherent, but this may not always be worth the effort.

Relationship to other modes of use This usage should not be confused with the usage that regards a style as defining a pattern *language* [cf. Monroe et al., 1997], which again puts styles at a level above patterns. This view is discussed in Section 6.3.3.

Some patterns are actually complete, though very abstract, generic architectures (e.g., the Model-View-Controller architectural pattern) rather than patterns. They are discussed in Section 6.3.6.

When re-engineering a software system, the first step may be the re-documentation of the current architecture (see Post-documentation and analysis, Section 6.3.5). For the reasons discussed above, often the only way to continue using styles is then an ad-hoc usage.

6.3.2 Use as Platform-induced Styles

At first sight, this usage appears to be tightly bound to the genesis of the architectural styles, namely their derivation from the platform that shall be used. The term "platform" is used in a very generic sense here. The discussion of what an (abstract) platform is in the sense of the MDA is related [cf., e.g., Almeida et al., 2004]. For our purposes, target or *implementation* platforms must be distinguished from *modelling* platforms. A modelling platform is a modelling language or notation, for example an ADL. There are ADLs that are specifically designed to support a specific architectural style, e.g. C2SADL [Medvidovic et al., 1999] for the C2 style [Taylor et al., 1996], but even ADLs that claim to be style-independent are usually biased towards some architectural styles [Di Nitto and Rosenblum, 1999]. Implementation platforms can be distinguished into several types again:

System Software The system software (operating system) and the services it offers to applications may impose an architectural style. However, in particular for distributed applications the actual operating system is today often hidden behind an additional software layer referred to as middleware.

Middleware Middleware-induced architectural style have received some attention from the research community [Baresi et al., 2004; Medvidovic, 2002; Medvidovic et al., 2003; Di Nitto and Rosenblum, 1999]. They are also in the focus of further work forming part of this thesis (see Chapter 7).

Programming Paradigm The *language constructs* (classes, objects, exception, event mechanisms, ...) a programming language provides and its implied *execution model* (for example the object execution model of languages such as Java and C++) advocate some programming paradigm. The paradigm of the language(s) considered for implementing a system may influence the structure of the resulting system at an architectural level [Mehta et al., 2000a]. For relatively low-level languages, such as C, only very few constraints are implied, but for very high-level languages, a very specific architectural style may result, in particular if the language effectively inhibits access to lower-level constructs.

Hardware Besides the software artifacts mentioned before, the hardware also imposes some style on the software system, if only indirectly through the software layers in between. For distributed software systems, however, the physical topology and the properties of physical connections inevitably influence quality properties of the running system, and must thus be considered in system design.

Organisational Structure The famous Conway's law already postulated the dependence of system structure on the structure of the designing or developing organisation: "[...] organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations" [Conway, 1968]. While the inevitability of this constraint may be questioned, it is clear that a tendency towards it prevails and it may adversely affect the structure of the system from a technical point of view without compensating measures. Similarly, the organisational structure of the target organisation may influence the architecture.

What becomes clear from this discussion is that a software system makes use of many platforms that may suggest different architectural styles. As explained in Section 3.1.1.2, the architecture may be specified independently from the platforms it is implemented on. However, a trade-off must be made between the platform-independence and the complexity of the mapping to the chosen platform on the hand, and between the platform-independence and the risk that the mapping will lead to a system with undesired quality characteristics due to some incongruities between the structure implied by the architecture and the structures that are beneficial to the platform.

It is possible to establish a taxonomy of platforms based on the architectural styles they enable [cf. Di Nitto and Rosenblum, 1999]. In creating the taxonomy, abstract platforms may need to be introduced in addition to physically existing platforms, which reliefs the tight binding of this usage to the genesis of the style. Based on a taxonomy, the style may be incrementally refined from an abstract towards a concrete style in a process based on this usage, yielding a refinement hierarchy of architectures as well. Unfortunately, no rich taxonomies of platform-induced styles are available yet, besides the basic ideas contained in Medvidovic [2002]; Medvidovic et al. [2003]; Di Nitto and Rosenblum [1999].

6.3.3 Use as Customised Styles

This is the primary usage intended by the community which focuses on ADLs that support the definition of architectural styles and conforming architecture, e.g. Rapide [Rapide Design Team, 1994], ACME [Garlan et al., 1997] resp. Armani [Monroe, 2000b], Wright [Allen and Garlan, 1997], xADL [Dashofy et al., 2005], ArchWare [Balasubramaniam et al., 2004; Oquendo et al., 2004] or Alfa [Mehta and Medvidovic, 2002, 2003].

Garlan [1995] discusses several different approaches to the definition and use of architectural styles, but assumes several common properties of any view of "architectural style":

a. The provision of a "vocabulary of design elements", which are "component and connector types".

b. The definition of a "set of configuration rules".

c. The definition of a "semantic interpretation", which gives some well-defined meaning to all configurations of design elements that satisfy the configuration rules.

d. The definition of "analyses" for configurations of that style. Examples include schedulability analysis, deadlock analysis, code generation, and conformance checking.

Styles used in this way may be seen as a concept complementary to patterns according to Monroe et al. [1997], who point out that "for a given style there may exist a set of idiomatic uses. These idioms act as microarchitectures, or architectural design patterns, designed to work within a specific architectural style." The architectural style provides general guidance for the architectural development process, while architectural design patterns solve specific problems within one or multiple styles [Monroe et al., 1997].

Monroe et al. also point out that an architectural style can be regarded not merely as a form of pattern, but as defining a whole pattern language. However, the term "pattern language" is controversial in itself. It is misleading as it be thought to may appeal to the common notion of a formal language, at least for computer scientists. It was used by Alexander et al. [1977] as a vague metaphor to natural languages, but the same concept could better be referred to as a pattern vocabulary [cf. Schmidt et al., 1996].

Relevant literature Keshav and Gamble [1998] propose a taxonomy of architectural integration strategies, among which are both patterns and styles. While the basic elements they identify—Translators, Controllers, Extenders, and combinations thereof—are more or less patterns or individual components, they additionally identify "loosely defined integration strategies", which do not fit into their main taxonomy, which may be regarded as architectural styles.

6.3.4 Style-based Pre-modelling

This usage is tightly bound to a specific kind of architectural styles, which is briefly introduced in the following. Klein and Kazman [1999]; Klein et al. [2000] provide an evolution of the original architectural style concept which focuses more explicitly on the quality characteristics of the resulting architectures. These are referred to as "Attribute-based Architectural Styles" (ABAS). One ABAS is assumed to *address* one quality attribute[1]. A system is modelled according to several ABASs, addressing the most critical attributes, yielding several architectural models. Since each model strictly conforms to

[1] In the ISO 9126 terminology, it is probably more appropriate to speak of a quality (sub-)characteristic.

a style, style-based analyses (cf. Section 6.3.3) may be performed on them. When each model is satisfactory with regard to its assigned quality attribute, an overall system model is synthesised from the individual models.

By this approach, the original idea of different styles leading to different system decompositions is combined with (some limited form of) heterogeneity of architectural styles within a system. However, the way the final system is derived from the multiple decompositions is not specified (which is probably not possible in the generality that the approach targets). Since it is known that many quality attributes require trade-offs, the properties of the resulting architecture may not conform to the properties of the individual style-based models. Furthermore, it is unclear where later changes to the architecture should be performed.

This usage of styles may be referred to as "style and forget", since after the style-based modelling has been done initially the benefits of architectural styles are abandoned again. The concept of ABASs, however, seems promising and might be used in the opposite way, which is described in Section 6.3.5.

6.3.5 Style-based Post-documentation and Analysis

This usage may be regarded the reverse of the previously described usage. Styles are not regarded in the primary development of the architecture, but only applied to intermediate or final results of defining the architecture. They may serve to document the architecture by interpreting it in terms of a specific architectural style, to ease understanding of the overall system. Additionally, style-specific analyses may be applied to the style-based views. An automatic mapping of a generic architecture to a style will not be possible, but the mapping may be defined once and needs only be adapted on changes. Perhaps even certain dependent refactorings for architectural refactorings on the primary system architecture might be defined.

Essentially, any approach to software architecture that defines multiple views on the architecture may be interpreted as this usage. Similarly to generic ADLs, architectural views impose some style upon the modelled entity, but usually this style is conceptually not very deep.

Relevant literature Clements et al. [2002] propose a style-based documentation of software architectures.

6.3.6 Use as Generic Architectures

This usage is not strictly a usage of architectural styles, since the artifacts that are employed in this usage do not fit the definition of architectural styles, but represent a different type of architectural constraint, namely generic architectures. A generic architecture is expressed within the same (or similar) language or notation as a (product) software architecture, but may

employ a different interpretation. A generic architecture may be regarded as a template that is enriched or refined into a product software architecture. In some cases, a generic architecture can also be interpreted immediately as the architecture of a product system.

Examples of architectural styles that are essentially generic architectures are the Model-View-Controller architecture and N-tier architectures (for a given N). These generic architectures can themselves be explicitly based on an architectural style in another usage. N-tier architectures, for example, can obviously be regarded as following the layered style.

Related Concepts Reference architectures are usually a special case of generic architectures [see e.g. Grosskurth and Godfrey, 2005]. Some approaches to product-line architectures view a product-line architecture as a special kind of reference architecture that is specific to the product line. Such product-line architectures may also be regarded as generic architectures in our sense. Product-line architectures which make use of more elaborate means to automatically derive individual product architectures or implementations are beyond the scope of our considerations.

Relationship to other modes of use This usage is sometimes regarded a special case of the ad-hoc use, when only one pattern is used and this pattern has system-wide scope.

6.3.7 Characteristics of the Modes of Use

After the individual modes of use have been identified and described, we now devise a set of characteristics and classify each of the modes of use with respect to these characteristics. The characteristics describe several requirements on the form of representation of the regarded architectural styles. We thereby establish the link to the typical level of discussion of architectural styles.

We consider the following characteristics:

Compositionality The characteristic "Compositionality" describes whether the composition of individual styles into a new style is possible, either manually or computationally.

Specialisation The characteristic "Specialisation" describes whether specialisation relationships between styles are relevant within the respective usage.

Explication and Rigour The characteristic "Explication and Rigour" describes whether the architectural styles considered in the respective usage need to be explicitly described in what detail and with what degree of formal rigour.

	6.3.1	6.3.2	6.3.3	6.3.4	6.3.5	6.3.6
Compositionality	Manual	No	Manual ?	No	n.a.	Manual
Specialisation	Yes	Yes	Yes	Yes	Yes	Yes
Explication & Rigour	Informal	No	Formal	Formal	Informal	Informal
Conceptual Level	(Abstract)/ Technical	Technical	Abstract/ (Technical)	Abstract	Both	Abstract
Relationship to System Quality Attributes	No	Analyses/ Empirical	Analyses	(Analyses)	Analyses	TODO
Suitability for Design-Space Exploration	Limited	Yes	Partially	Yes	No	No

Table 6.1: Characteristics of the Modes of Use of Architectural Styles

Conceptual Level The characteristic "Conceptual level" describes whether the architectural styles considered in this mode of use are (typically) bound to technical concepts (of an existing or planned technical platform) or if they are more abstract in nature.

Relationship to System Quality Characteristics This characteristic describes which kind of relationship the styles exhibit to the quality characteristics of the system. Of course, an architectural style always has some implicit relationship to the system quality, but we consider only explicated relationships here. System quality comprises external and internal implementation quality as well as architectural quality here (cf. Section 2.3). Essentially two types of relationships can be considered here: An explicit reference to some system quality characteristic that a style addresses, and the reference to analysis techniques that are enabled by the style (cf. Section 3.4.2).

Suitability for Design-Space Exploration This characteristic describes which role the respective usage of styles can play in the design-space exploration process, i.e. how either different styles or different design based on a style can be evaluated in that usage. Since the modes of use do not exclude each other, this only refers to the contribution of the considered usage.

An overview of the classification is given in Table 6.1, the details are explained for each characteristic in the following subsections.

6.3.7.1 Compositionality

Let S be the set of architectural styles and A the set of architectures. Style composition may be performed either at the type or instance level.

Style composition at the *instance level* means the following: Let there be two styles $x, y \in S$. If an architecture $a \in A$ conforms to a style x, it is modified in a way that it conforms to style x and conforms to style y as well.

Then x and y are composable for a. Thus, there is a (partial) composition operator $\overset{\circ}{\S}_1 : (A \times S) \times S \to A$ such that:

$$(\overset{\circ}{\S}_1(a, x, y) = b \Rightarrow b \text{ conforms to } y) \tag{6.1}$$

$$\wedge\ (\overset{\circ}{\S}_1(a, x, y) \text{ defined} \Rightarrow a \text{ conforms to } x) \tag{6.2}$$

Composition at the *type level* means that a (partial) composition operator $\overset{\circ}{\S}_2 : S \times S \to S$ is defined on the set of architectural styles. Composability at the type level does not imply that two architectures $a, b \in A$ in two different styles $x, y \in S$ can be meaningfully combined into a new architecture conforming to $a \overset{\circ}{\S}_2 b$.

Compositionality can either be impossible resp. *undefined*, *manual*, *semi-automatic* or *automatic*. It is automatic if the composition operator is computable. If the composition operator is only partially computable, or if the determination of the composition is partially supported by an program, compositionality is considered semi-automatic. Of course, automatic compositionality is only conceivable for styles which are formally specified.

Ad-hoc Use of Styles The composition of styles used in an ad-hoc manner is expressly *manual* at the *type level*, and *semi-automatic* on the *instance level*. On the type level, new styles may be designed by combining the ideas underlying existing styles, which is a creative process for the most part. On the instance level, composition is necessary in most architecture, as no single style is supposed to be apt to support the development of a whole architecture. Tools may support the instantiation of new styles into an existing architectural description, but conflicts may occur, which cannot be resolved automatically.

Use as platform-induced styles The composition of arbitrary platform-induced styles is considered to be essentially *undefined*, but when specialisation makes use of multiple inheritance, similar benefits may be achieved.

Style-based post-documentation and analysis Compositionality at the instance level is not applicable, since the derived views are not meant to be tangible.

6.3.7.2 Specialisation

Specialisation relationships between different styles are conceivable for all of the modes of use, but play a different role for each of them. Specialisation relationships can be distinguished into single and multiple inheritance. Whether subtyping concepts from type theory can be applied depends on the degree of formality of style specification.

Ad-hoc Use of Styles Specialisation is one of many relationships that are defined between some patterns. It might be exploited in choosing a pattern fitting a problem at hand in a stepwise process, but no work on such a process has been published. Since specialisation relationships are typically specified informally, multiple inheritance is possible without introducing additional problems.

Use as platform-induced styles For platform-induced styles, it is possible to exploit specialisation relationships in the development process. The style may be incrementally refined from an abstract style towards a concrete style, yielding a parallel hierarchy of middleware platforms. Style taxonomies may also employ multiple inheritance. We will discuss such taxonomies in Section 7 and a development method that exploits such taxonomies for platform selection in Section 8.

Use as customised styles When defining a custom style, it is efficient to reuse existing style definitions by defining the new style as a specialisation, which also allow the reuse of analysis and design tools existing for that style.

The same applies for style-based pre-modelling and post-documentation.

Use as generic architectures Specialisation of generic architectures is possible, but this is exploited only seldom explicitly.

6.3.7.3 Explication and Rigour

While the level of explication and rigour is independent from the mode of use in principle, certain minimal levels that the style specification must fulfil can be determined on the one hand, and typical levels that can be found can be identified as well.

Ad-hoc Use of Styles In an ad-hoc use of styles, rigour is typically informal. Architectural patterns are only described very vaguely by just giving examples of their instances that do not claim generality at all.

Use as platform-induced styles Platform-induced styles are seldom described explicitly at all yet, with the exception documented by Di Nitto and Rosenblum [1999]. However, the great variety of existing middleware platforms has not yet been specified formally.

In fact, access to the style description might not be necessary at all after a taxonomy of platforms has been derived based on formal style modelling. Exploitation of additional style features still requires the explicit use of formal style specifications.

Use as customised styles In this usage, formal specification of styles is expected in the context of ADL-based architecture specification.

Style-based pre-modelling An important aspect in style-based pre-modelling are formal analyses, thus a formal definition of the style is required.

Style-based post-documentation and analysis The required level of rigour depends on the focus in this usage. If the focus is on documentation, informal and vague specification of the style may suffice, but when formal analyses should be performed, formal style specification is necessary as well.

Use as generic architectures Often, generic architectures are merely provided as a basis for stakeholder communication about a well-known structure of the system to be built. In this case, an informal specification of the generic architecture is sufficient. In other cases, a binding of elements of the generic architecture and the concrete system's architecture may be required, in which case the generic architecture must be specified in a formalism that is compatible to the formalism used to specify the system's architecture.

6.3.7.4 Conceptual Level

The characteristic "Conceptual level" describes whether the architectural styles considered in this usage are (typically) bound to technical concepts (of an existing or planned technical platform) or if they are more abstract in nature. This distinction corresponds to different conceptual levels at which software architecture can be specified in general (cf. Section 3.3.4).

Ad-hoc Use of Styles Here, both types are possible. While the old general-purpose architectural styles are quite generic and abstract, current publications on architectural patterns are focused towards specific platforms, and are either specifically tailored towards one platform or provide examples for multiple platforms and are thus more technically oriented.

Use as platform-induced styles Platform-induced styles are naturally conceptually close to the platform they intend to support and are thus technical in nature. However, generalisations made in a taxonomy of platform styles may introduce concepts that have no direct counterparts in any existing platform.

Use as customised styles Customised styles can be both abstract and technical in nature. Due to the fact that the ADLs used for specification are conceptually quite detached from typical implementation techniques, abstract styles are more prevalent.

Style-based pre-modelling Since the styles are used in an early stage of architectural development only, before the final architecture of the system to be built is established, and the link of the pre-models and the final architecture is quite loose, the styles in this usage are very abstract.

Style-based post-documentation and analysis In this usage, both types of styles are conceivable.

Use as generic architectures Generic architectures are typically much more abstract than the final products' architectures, however, in principle, generic architectures could be at the same level as well.

6.3.7.5 Relationship to System Quality Attributes

Architectural styles on the only hand are intended to improve the process quality of software development by acting as an intellectual tool to the system designers. Additionally, they are intended to improve the product quality of the product that is the outcome of the development process. The different modes of use support the product quality improvement in differing ways.

Ad-hoc Use of Styles Due to the vague nature of the styles used in this way, a relationship to system quality attributes is difficult to establish. The architectural pattern literature lists experience-based hints on when to use which patterns, but the general applicability of these rules is questionable.

Use as platform-induced styles Through defining styles for multiple platforms in a commensurable way, styles and systems based on these styles can be analytically and empirically evaluated to produce a guideline for choosing a suitable style and platform for a given scenario.

Use as customised styles Customised styles often allow style-specific analyses.

Style-based pre-modelling Similar to customised styles, style-specific analysis may be performed, however these only establish properties of the pre-models, and the link to the actual system's quality is unknown.

Style-based post-documentation and analysis Similar to customised styles, style-specific analysis may be performed.

6.3.7.6 Suitability for Design-Space Exploration

Design-space exploration seeks to support making trade-offs between system quality attributes in the development process. This characteristic serves as a summary judgement and combines the previous characteristic with the possibility to exploit the relationship to the quality attributes in the architectural development process.

Ad-hoc Use of Styles Due to the ad-hoc nature of using styles in this usage, the suitability for systematic design-space exploration is very limited.

Use as platform-induced styles The method described in Section 8 is targeted at supporting the production of a guideline to choose a style.

Use as customised styles Using styles as customised styles is only partially suited for design-space exploration. Many choices must be made in customising the style, i.e. before the actual modelling is done. A method for design-space exploration on this level is conceivable, but has not yet been proposed and evaluated in detail.

Style-based pre-modelling This usage is in fact the most direct correspondence of the idea of design-space exploration. The main issue with this usage is the linking of the design-space exploration with the actual modelling.

Style-based post-documentation and analysis Due to its post-mortem nature with respect to modelling, the suitability for design-space exploration is very limited. Perhaps well-integrated tool support that enables a frequent automatic derivation of style-based views would allow for enabling design-space exploration.

Use as generic architectures In fact, a generic reference architecture typically represents the outcome of design-space exploration for a particular domain. However, in few cases multiple generic architectures for a domain exist that differ in making different trade-offs, so the designer could choose from a set of generic architectures that makes a trade-off that fits the product's needs. Rather, certain trade-offs have implicitly been made when designing the generic architecture, and others must be made on the basis of the generic architecture.

7 Modelling Middleware-oriented Architectural Styles

The goal of this chapter is to introduce the tools required for modelling middleware-oriented architectural styles. The idea of middleware-oriented architectural styles is to capture the architectural constraints that are imposed upon a software architecture by the middleware technologies it uses explicitly in a dedicated design artifact. We claim that an architectural style (as defined in Section 3.4.2) is an adequate model of these constraints. We adhere to the idea of Baresi et al. [2003]; Di Nitto and Rosenblum [1999]; Sullivan et al. [1997b], that a middleware technology defines one or more architectural styles (see Section 7.1 for an in-depth discussion of this relationship). We name the specific type of architectural styles we consider in our work as Middleware-oriented Architectural Styles, briefly MidArch Styles.

To procedurally ensure the adherence of an architecture to a MidArch Style, both the style and the architecture must be explicitly described in a formal language. Formalising the architectural style improves understanding and communication about the global concepts underlying the architecture. Adherence to a coherent architectural style improves understandability and maintainability of the system and its architecture. In the context of the MidArch Design Method described in Chapter 8, results of architecture evaluations can be related to the style used. A formal style description is also a prerequisite for automatically checking the conformance of an architectural description with the style (architecture-to-style conformance check). Checking conformance of an implementation with its architectural description is then another issue (implementation-to-architecture conformance check), which is discussed in Section 11.2. The approach described there exploits the fact that the architecture is style-based to enforce the style's constraints on the implementation.

In this chapter, the focus is put on the MidArch Style Modelling Approach that is used to create descriptions of MidArch Styles. Based on this approach, the MidArch Design Method for middleware selection based on architecture-level design exploration that exploits MidArch Styles is presented in Chapter 8.

Overview In Figure 7.1, the main elements of our approach to modelling MidArch Styles are shown. The structure of this chapter is oriented towards the different elements, which is described in the following. First, the

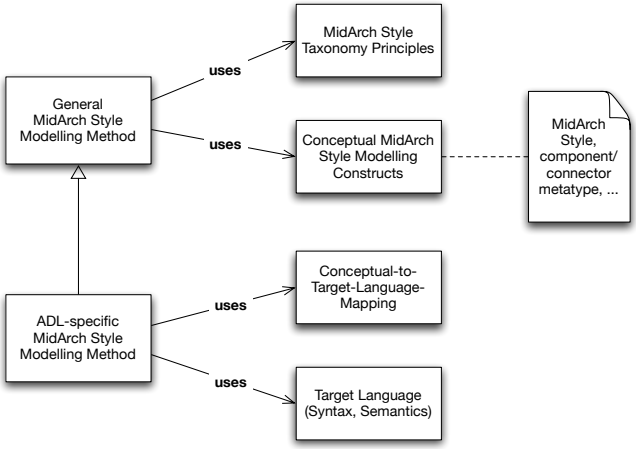

Figure 7.1: Overview of MidArch Style Modelling Approach

relationship of middleware products, middleware platforms and middleware-oriented architectural styles is discussed in Section 7.1. Then we discuss the central assumptions underlying our approach (Section 7.2) and we identify requirements and corresponding design decisions for the modelling approach (Section 7.3). We distinguish the modelling *concepts* of MidArch Styles, which are represented by the conceptual MidArch Style Modelling Constructs, from the *notation* (target language) used for expressing MidArch Styles, which we refer to as a style-enabled ADL (Section 7.4). Multiple MidArch Styles are related to each other in MidArch Style Taxonomies (see Section 7.5). The general method, i.e. that which is independent from the chosen style-enabled ADL, for deriving a MidArch Style from a middleware product is described in Section 7.6. The specific mappings to the modelling constructs of style-enabled ADLs (Acme and UML) are then discussed in Section 7.7.

7.1 Relationship of Middleware, MidArch Styles and Architectural Descriptions

The relationship of middleware and MidArch Styles has both a static and a dynamic aspect. The static aspect refers to the relationship of given middleware products, middleware platforms, MidArch Styles, and architectural descriptions (Section 7.1.1). On the other hand, the dynamic aspect refers to the relationships of the processes that create and modify these artefacts (Section 7.1.2).

7.1.1 Static Aspect

By definition, there is a direct correspondence of middleware platforms and concrete MidArch Styles. We characterise the more complex relationship between middleware products and middleware platforms, and between MidArch Styles and architectural descriptions.

Hard vs. Soft Constraints Architectural styles constrain the design space (Section 3.4). Since Middleware-oriented Architectural Styles are related to middleware products, it is suggestive that the style's constraints reflect constraints imposed by the middleware product. The nature of the middleware products' constraints is not as strict as that of the style. To characterise this property more precisely, we distinguish interpretations of constraints as *hard* and *soft* constraints. A constraint in general is a proposition made within one element A about another element B.

In the case of a hard constraint, the proposition takes the form of a logical formula, which can be evaluated in the context of B. B either conforms to the constraint or violates it. A violation of one constraint implies the *invalidation* of B with respect to A. Obviously, hard constraints only make sense if at least the constraints of A and the portions of B that are necessary to evaluate the constraint are expressed formally.

A soft constraint, on the other hand, represents a suggestion rather than a strict necessity. It may be described by an advise within an element A to be followed by an element B, which relies on A. A violation of the constraint by B is a *misalignment* of B with respect to A. While individual violations may be tolerable, excessive violations of one or multiple soft constraints may make B unusable in practise.

Many constraints imposed by middleware are not hard constraints, because software can be implemented so flexibly that most restrictions can be masked in principle at a higher level. However, this usually bears some impairment of performance and maintainability. Software systems using a middleware product are endorsed to adhere to a corresponding architectural style in order to benefit most from the technology. Violations of the middleware product's constraints may make the implementation more complicated (referring either to the development process, the resulting product, or both), and practically infeasible.

This is complemented by a view from the opposite direction, which gives priority to the style:

> "A style is often supported by infrastructure that eases the implementation of components and provides support for their interaction via the style's connectors." [Allen, 1996]

Middleware-oriented architectural styles are also said to be *induced* by the middleware product [cf. Di Nitto and Rosenblum, 1999], which has

the connotation of a tighter relationship. It may also raise the expectation that the style description may be algorithmically derived from the product, which is however unrealistic. We thus do not promote the use of the term[1]. Sullivan et al. [1997b] show that there may be subtle incompatibilities between an architectural style and a middleware product using the example of the Component Object Model (COM).

As a convention, we attribute middleware platforms to possess hard constraints as well, to emphasise the close link between a middleware platform and its description through a MidArch Style.

Degree of Conformance In practise, legacy systems do not strictly conform to any middleware-oriented architectural style [Jazayeri et al., 2000, p. 3]. One reason is that the induced styles are only implicitly described in the middleware documentation, so it is difficult to follow one of them systematically. The investigation of further reasons for this is not the subject of this thesis.

7.1.2 Dynamic Aspect

The dynamic aspect of the relationship between middleware and MidArch Styles can be split into the influence that choosing some middleware platform has on the software architecture modelling process. In an even more general sense, there is an impact of using MidArch Styles on the evolution of middleware products themselves.

Influence of Middleware on Modelling As noted by Zhu et al. [2005], to know early during system design which middleware to use impacts the structure and the behaviour of the resulting system design compared to a typical ad-hoc solution. An empirical evaluation of this hypothesis can also be imagined.

Whether the impact is positive, is another question. On the one hand, it will be easier to map the architecture to the implementation platform, and design knowledge from the design of the middleware platform is implicitly reused. On the other hand, it is not necessarily the case that the implicit design decisions in the middleware are best suited to fulfil the given requirements. They may also lead to solutions that have suboptimal internal or external quality.

Evolution of Middleware and Styles MidArch Styles can be conceived to be used as a vehicle for the evolution of middleware products. Consider the following scenario: A software architecture is modelled based on some

[1] In earlier publications by the thesis' author, the term has been used nonetheless [Giesecke, 2006a; Giesecke et al., 2007a].

MidArch Style A, which is induced by an existing middleware product p_A. During modelling, some problems arise, which are caused by the constraints of style A. The style is modified into a new style A' to eliminate the problem. However, now there is no middleware product available that implements A', i.e. the style A' is abstract. The problem can be solved by wrapping (adaptor or wrapper pattern) the middleware product p_A such that the architecture can be mapped towards the wrapped p_A, or p_A may be modified such that it also supports A'.

A related research topic, which is not addressed within this thesis, is the modelling of style refactoring operations, and their propagation to dependent software architectures (see Chapter 14).

7.2 Assumptions

In this section, we summarise the relevant assumptions that underlie the MidArch Style Modelling Approach we propose. These assumptions have been discussed in part in the previous sections, which are referenced where relevant. We later get back to these assumptions and discuss whether they might be relieved and which implications this imposes for the approach. However, we do not empirically show the validity or viability of these assumptions and definitions, as this is beyond the scope of this thesis.

These assumptions are:

Assumption 7.1. We assume that an architectural description consists of models which describe the system's architecture from different viewpoints.

Assumption 7.2. We assume that an architectural style is only meaningful for a certain viewpoint, so the description of one system may follow different styles from each viewpoint.

As a consequence of this assumption, we focus on architectural descriptions from the logical component structure viewpoint.

Assumption 7.3. We assume that an architectural description declares the conformance with an explicitly described architectural style.

Assumption 7.4. We assume that there is no universally applicable architectural level, but the level of abstraction of an architectural description is the result of a deliberate design decision within a specific software project.

That is, the level of abstraction is not prescribed for all software projects due to some general principle that is part of the modelling approach. Still, it may make sense to make design decisions that apply organisation-wide.

Assumption 7.5. We assume that the applicability of architectural styles may differ depending on the application and technology domains of the subject system.

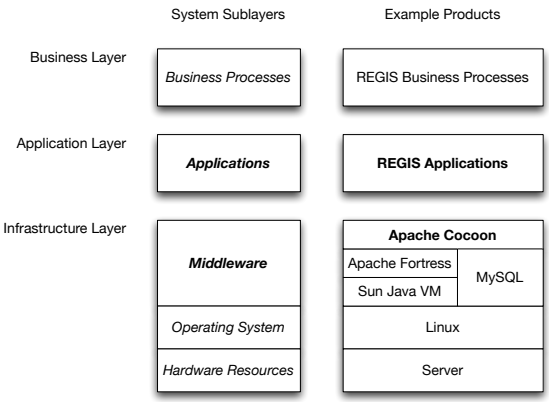

Figure 7.2: Architectural Layers and Example Technologies

As a consequence of this assumption, we focus on web-based and distributed middleware-intensive business information systems in the thesis.

Assumption 7.6. We assume that the architecture of a business information system consists of three coarse-grained layers: business layer, application layer, and infrastructure layer [cf. Hasselbring, 2000]. We additionally split up the infrastructure layer into at least three sublayers: middleware, operating system and hardware resources (cf. Figure 7.2). We regard architectures at the application layer (application architectures), which make use of primitives provided by the middleware layer (cf. Section 5.1.2).

Assumption 7.7. We assume the underlying middleware platform is chosen at the architectural level, i.e. it is not just an implementation-level decision. This does not necessarily mean that the concrete middleware product is already chosen, if there are several equivalent or conceptually similar alternative middleware products that implement the same middleware platform.

7.3 Design Decisions in Middleware-oriented Architectural Style Modelling

In this section, we present requirements that arise in devising an architectural style modelling approach and the corresponding design decisions taken in the MidArch Style Modelling Approach. The central design decision of language independence is discussed in Section 7.4.

Basically, our view of MidArch Styles is based on the ideas by Abowd et al. [1995]; Garlan [1995]. A central concept required for the definition are *configuration rules*:

Definition 7.1 (configuration rule)**.** A configuration rule is a formula in first-order (predicate) logic over the elements of configurations and their properties.

An architectural style is considered to consist of the following aspects:

1. A vocabulary for design elements, i.e. component and connector metatypes.
2. A set of configuration rules.
3. A semantic interpretation, which gives some well-defined meaning to all configurations of design elements that satisfy the configuration rules of the architectural style.
4. The definition of analyses for configurations of that architectural style, e.g. schedulability and deadlock analysis.

The first two aspects can be codified formally in an ADL, the third is necessarily at least in part informal. At the moment, we do not focus on the fourth aspect, analyses. We use the term "metatype" instead of "type" at the style level, to allow the sole use of "type" for types defined at the instance level (see also Section 7.6.1).

This style concept can be applied to any structural (or decomposition-oriented) viewpoint. The viewpoint we use is the *logical component structure viewpoint* on the application layer.

The elements of a MidArch Style are defined by the MidArch Metamodel, which is presented in Section 7.6.1.

7.3.1 Identification of Relevant Aspects

MidArch Styles are used to model conforming software architectures in the logical component structure view. The architecturally relevant aspects of middleware platforms must be identified, i.e. all aspects that are visible at the architectural level from the logical component structure viewpoint. A style-enabled ADL must also allow to describe component-and-connector configurations that declare an architectural style. It should be possible to check conformity of the configuration to its declared architectural style.

Component-and-connector views use the modelling constructs of configurations, components, connectors, ports and roles. The modelling constructs that can be used on the application layer and their relationships are provided by the adjacent underlying middleware layer. The impact of the middleware platform on application-level software architectures must thus be captured using terms that are based on the provided modelling constructs. This does not mean that the internal structure of the middleware product is modelled within the style, but only those aspects that are exposed to the application-level architect. The style expresses this impact through *metatypes* for the modelling constructs used in component-and-connector views (components, connectors, ports, roles) and through *constraints* for configurations of instances of these metatypes.

7.3.2 Implicitly Layered Architectures

Services that are offered by the middleware should also be reflected in the style description, while retaining the fact that they reside on a different architectural layer.

A middleware product typically also provides services via components to applications. In our view, these form also part of the MidArch Style. They are therefore distinguished from the application layer. Architectural descriptions based on a MidArch Style are thus implicitly non-strictly layered[2]. Non-strict layering means that references are allowed not only to the adjacent lower-level layer, but also to other lower layers [cf. Szyperski, 2002, ch. 9.1.7].

7.3.3 Relationship to General-purpose Architectural Styles

To improve the understandability of styles induced by middleware technologies, relationships of these styles to what is traditionally regarded as architectural styles, i.e. general-purpose architectural styles (cf. Section 3.4), must be identified. General-purpose styles are well-known, and knowledge on the relationship of middleware products to general-purpose styles makes their understanding easier and more reliable. However, general-purpose styles represent quite vague structuring principles. Thus, it is difficult to formalise them such that any concrete MidArch Style that seizes the idea underlying a general-purpose style can be expressed as a strict specialisation of that style.

Relationships between both general-purpose and middleware-oriented architectural styles need to be modelled, which is made possible through a MidArch Style Taxonomy (see Section 7.5).

It may be impossible or awkward to define a specialised style as a formal specialisation of its superstyle. For example, when a specialised style requires the relaxation of a constraint defined by the superstyle, this cannot be expressed in most ADLs. This may in particular be the case for styles that are variants of another style of the same middleware product. In these cases, the styles can be marked as being related in a (conceptual) MidArch Style Taxonomy (see Section 7.5.5).

7.3.4 Style Combinations

The question of what combining architectural styles means is more prominent for MidArch Styles compared to general-purpose architectural styles: The implementations of middleware technologies may be layered upon each other (e.g., Cocoon is based on Fortress). Should one style be modelled as a specialisation of the other or should they be specified as independent styles?

[2] Layering is also an architectural style, but the type of layering considered here is orthogonal to the logical component structure viewpoint.

One possible way to deal with style combinations in general is to introduce a style calculus which allows to computationally derive combined styles. While this may in theory be possible, we doubt that such an approach would be feasible. The problem is that two styles may be combined in many ways. A style calculus would need to make an implicit decision for one specific combination. Suppose there is a middleware-oriented architectural style that defines three component metatypes. When defining the combination with a client/server database access architectural style, it is unclear which component type may act as a client or a server in the combined style. While in this case a simple parameter might be introduced, this approach would not be applicable in general, since the styles to be combined may be arbitrarily complex. Simply merging the elements and constraints of both styles would also result in an architectural style that is valid, but its meaning is unclear in general. When devising the style calculus, not all styles it may be applied to are known. Thus, it is likely that the style combinations that result from applying the calculus are not meaningful at all. For these reasons, style combinations in general must be manually defined in the MidArch Approach.

An alternate approach would be to allow a configuration to declare multiple architectural styles. However, such an approach would result in the same problems as discussed above.

A special case is the situation, when a new refinement of an existing architectural style is to be made rather than a combination of two previously architectural styles. This can be implemented as a specialisation of the existing architectural style. If one middleware platform is layered upon another middleware platform, the upper layer architectural style should be modelled as a specialisation of the lower layer architectural style unless the lower middleware layer is hidden by the upper middleware layer.

For specialisation relationships referring to style combinations, the same remark applies as for specialisation relationships to general-purpose architectural styles.

7.4 Language Specificity

Architectural styles in general are described using a formal architectural description language (ADL). For our middleware-oriented architectural style modelling approach, either a specific ADL can be designed, or an existing ADL can be adapted. More generally, the style modelling approach can be defined primarily on either the conceptual level, or on the logical level.

We consider it essential to use a formal language which allows expressing both style definitions and conforming architecture descriptions.

We chose not to develop an entirely new ADL. In the past, many ADLs have been developed that tried to address a specific problem in software architecture. For example, Acme has been developed as an intermediate

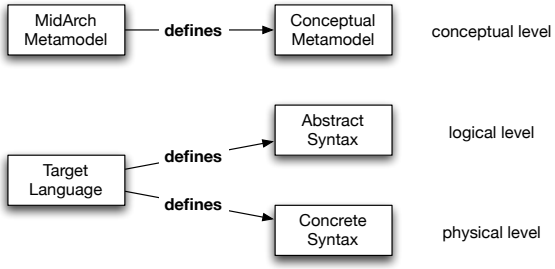

Figure 7.3: Language Definition Levels in the MidArch Approach

language which fosters the combination of specific ADLs; xArch is an ADL which is modular and can be easily extended for specific purposes (see Section 3.5.2). It might appear natural to follow the same way, however we chose not to do so in order to allow us to focus on more general aspects of architectural styles.

In Figure 7.3, the language definition levels that result from this approach are shown. These are analogous to the definition levels in data modelling as defined by Atzeni et al. [1999] (listed on the right). The general MidArch Style Modelling Method defines the conceptual metamodel on the conceptual level. The logical and physical level are only defined by ADL-specific MidArch Style Modelling Methods, since both the abstract and concrete syntax depend on the specific ADL. In addition, they define a mapping from the conceptual metamodel to the language constructs (abstract syntax) of the ADL whose complexity may vary. It depends on the alignment of the conceptual metamodel and the ADL. Multiple concrete syntaxes may exist for one abstract syntax. This is the case for Acme, for example, which has both a "plain" and a XML rendering (xAcme).

This design decision implies that compromises must be made in mapping the features we deem conceptually relevant to the concrete language elements of existing ADLs, since we subject us to the sets of language design decisions that have been made by their designers. Thus, for each target ADL, a mapping of the MidArch Style concepts to the ADL's concepts must be defined.

Unfortunately, ADL notations and their underlying concepts are not well-known by practitioners, and even in the overall software engineering research community. Moreover, it is difficult to combine them with software modelling notations that are typically used to model software, most importantly the Unified Modelling Language (UML). Therefore, it would be beneficial to use the UML for modelling component-and-connector views, which has the additional benefit of easy integration with models for other architectural

	new language	existing language	
		Acme	UML
language definition effort	++	–	+
modelling effort	–	o	+
understandability	++	+	o
prominence	– –	o	++
integration with other viewpoints	–	o	++
tool support	– –	+	++

Table 7.1: Consequences of Choosing a New or Existing Language for Modelling Styles and Style-based Architectures

views, such as behavioural aspects, within a single notation. However, the UML does not natively provide a modelling construct for architectural styles.

It could be argued that it were possible to achieve the same benefits with less effort by directly designing a UML-based modelling method, but we consider it conceptually cleaner to independently elaborate a conceptual metamodel first. The intricacies of the UML metamodel can then be dealt with when defining the mapping.

7.4.1 Rationale

In making the decision not to devise a new language for modelling styles and style-based architecture for MidArch, consequences in several dimensions have been evaluated. A summary of the evaluation results is given in Table 7.1. Without considering specific languages, only a potential range of evaluation results could have been given, so we give the evaluation results for the two languages we ultimately defined mappings for, i.e. Acme and UML. Since different mappings to these languages are conceivable, the evaluation might still vary, especially in the case of the UML. The results given are valid in a strict sense not for the target language, but for our specific mapping only.

A five-point scale (– –, –, o, +, ++) was used to evaluate each dimension. For the first two dimensions, a lower value is better, while for the others a higher value is better. We describe the dimensions as well as the evaluation results for each dimension in the following.

Language Definition Effort The language definition effort characterises the effort necessary to define the syntax and semantics of the language, and to define a mapping from the conceptual MidArch Metamodel to the language.

For an existing language, this reduces to the effort necessary for the mapping. For a new language, the definition of the mapping will be trivial, as the language will match the metamodel closely. However, the effort

for defining syntax and semantics consistently will be significant and will outweigh the former advantage. When comparing Acme and UML, the mapping of the MidArch Metamodel to Acme is straightforward, while the mapping of the MidArch Metamodel to the UML proved to be complex with many design decisions to be decided upon.

Ideally, an ADL supports modelling both styles and concrete architectures as distinct first-class syntactical entities. For example, Acme [Garlan et al., 2000] supports modelling a style as a Family and a concrete architecture as a System.

Alternative notations include other ADLs which explicitly support styles as a modelling construct, e.g. ArchWare [Balasubramaniam et al., 2004] and Alfa [Mehta and Medvidovic, 2003]. In addition, ADLs which do not provide a first-class concept for representing styles may also be considered a style-enabled ADL. For such languages, a greater effort must be taken to define the mapping of MidArch Style concepts. In our work, we did not define style-enabled ADLs for these languages (an overview of ADLs has been provided in Section 3.5). An exception is the UML: While it does not provide a first-class concept for modelling styles itself, it can be richly extended and refined by defining *UML Profiles*. We present such an approach for modelling MidArch Styles in Section 7.7.2.

Modelling Effort The modelling effort characterises the effort for modelling concrete architectural styles and style-based architectures in the language.

We assume that this effort would also be lowest in an own language, since this matches the concepts of style modelling best. Acme comes in second, since it natively supports modelling styles and style-based architectures. The UML mapping requires some trade-offs in order not to break its consistency, which results in a higher modelling effort.

Understandability Understandability of styles and style-based architectures modelled in the language is in some sense reciprocal to the modelling effort, but it refers to the result of modelling rather than to the modelling process itself.

Still, the same arguments that were mentioned for modelling effort apply here as well.

Prominence Prominence of a language is important for a similar reason as understandability. While understandability factors out the learning curve, interest in models in a language that might be understandable easily after learning the language will be low.

The prominence is worst for a newly defined language. Acme is quite prominent among the research ADLs, but nonetheless much less known than the UML.

Integration with other Viewpoints Integration with viewpoints other than the logical component structure viewpoint addressed by our work is important to allow a complete consistent architectural description beyond the scope of the MidArch approach. Every language could be extended in principle to support other viewpoints by extending its grammar or metamodel, but this requires a significant effort. We only evaluated possibilities to model other viewpoints that already exist on the language definition level.

A new language designed for MidArch would initially only support the logical component structure viewpoint, so there are no opportunities for integration. Acme was originally designed specifically to act as an interchange or integration language between multiple languages that describe different viewpoints, but this feature is not usable very well. UML is quite comprehensive by offering many diagram types, which partly lack the degree of formality (e.g., well-defined operational semantics) demanded by some researchers, but covers many viewpoints nonetheless.

Tool Support Tool support is required for the practical usability of a language. The most important features are modelling support (for both styles and style-based architectures), or at least syntax checking tools. Desirable are further general as well as style-specific analysis tools.

For a newly designed language, tools would need to be created from scratch. If a metamodelling infrastructure such as MOF were used, the effort could be reduced by using model-driven techniques, for example through the Eclipse EMF project. Acme provides tools for visual modelling (AcmeStudio) as well as a library (AcmeLib) for implementing further tools. The stability of AcmeStudio has proved to be suboptimal [Gottschalk, 2007]. For UML, a variety of general- and special-purpose modelling tools is available. Some of these tools are also extensible by third party software. Through the XMI format an exchange of models between tools is (theoretically) possible. UML tools do not natively provide style modelling features. To support the MidArch approach, they must at least support the definition and application of custom profiles, which is only offered by few UML tools. A detailed evaluation of tools in this respect has not been done yet.

It must be noted that genuine visual modelling is not possible with Acme because the graphical notation shows only a subset of the information contained in the textual representation. This is also true for the UML, where some ambiguities are introduced through the lack of a formal foundation for the graphical representation of the models, but to a much lesser degree.

Conclusion This evaluation led us to the decision not to design a new language, since the greater language definition effort, non-reusability of tools and lack of possibilities to integrate with other viewpoints outweighed the benefits of easier modelling. In addition, this enabled us to lay the focus of our

research on the conceptual level rather than having to deal with intricacies of language design. The two languages chosen as target languages, Acme and UML, were selected for different reasons: Acme was close to the conceptual metamodel and thus provided an easy starting point for experimenting with modelling. The UML was chosen in a further step primarily due to better tool support and integrability with other viewpoints, and because it is much better known outside the ADL community.

7.5 MidArch Style Taxonomies

A MidArch Style Taxonomy relates multiple architectural styles to each other, and thus amends the information contained in the isolated style descriptions. More specifically, it shows relationships between a set of general-purpose architectural styles, as well abstract and concrete middleware-oriented architectural styles. It can be created either on a conceptual level or on a logical level. We define common properties of both types of taxonomies first. The specialised types of taxonomies are discussed in Section 7.5.5.

7.5.1 Elements of MidArch Style Taxonomies

Essentially, there are three types of entities contained in the taxonomy. These represent *general-purpose*, *abstract* and *concrete* MidArch Styles, respectively:

Concrete MidArch Styles These styles are associated with a concrete middleware product, while general-purpose and abstract MidArch Styles are not.

Abstract MidArch Styles These styles are based on some middleware specification document (e.g., the Java Enterprise Edition specification), but not a concrete middleware product.

General-purpose MidArch Styles They represent more fundamental structuring principles such as those contained in well-known collections of architectural styles and patterns (e.g., the pipes-and-filters architectural style).

However, the distinction is not always clear, but it is the object of a design decision in the design of an individual MidArch Style Taxonomy whether some architectural style is a general-purpose or an abstract architectural style.

7.5.2 Relationships in MidArch Style Taxonomies

We distinguish four types of style relationships:

inherits from Strict inheritance is used to express specialisation between a general-purpose and a concrete architectural style or between concrete architectural styles of different middleware technologies which build upon each other (e.g. Cocoon builds upon Fortress).

variant of Variant inheritance is a specialisation of strict inheritance and is used to express the relationship between a specialised variant and a more basic architectural style of a single middleware product. By merging all classes that are connected by variant relationships, a taxonomy of middleware products can be obtained from a MidArch Style Taxonomy.

related to Styles that are related and are specialisations in an informal sense, but not formally, are connected by a "related to" relationship.
As noted in the list of challenges, general-purpose architectural styles represent rather vague structuring principles [Perry and Wolf, 1992], which may inhibit modelling the relationship to them as strict specialisations. In some cases, it might be possible to resolve "related to" relationships into strict inheritance relationships by introducing more intermediary general-purpose architectural styles. A trade-off must then be made between the effort necessary for modelling all styles and the rigour of the relationships.

builds upon If one middleware platform is implemented using another middleware platform, the style of the higher-level product may be based on that of the lower-level product. For example, the Cocoon middleware is based on the Avalon component platform; any application using Cocoon must make use of that component platform, too. However, this is not necessarily the case, since the upper layer platform may entirely hide the lower layer platform. In this case, the relationship is irrelevant to the application using the upper layer platform, and should not be reflected in a MidArch Style Taxonomy.
If the lower level style remains visible to the application developer, however, a *builds upon* relationship should indicate this fact. When modelling the style in a style-enabled ADL, this may be mapped to a specialisation relationship.

The inclusion of further types of relationships is left to future work.

7.5.3 Well-formedness of MidArch Style Taxonomies

The relationships defined in the previous section may not be used in an arbitrary way. We define the following constraints for a MidArch Style Taxonomy:

1. General-purpose architectural styles cannot be specialisations of non-general-purpose architectural styles.

2. For any general-purpose architectural style a subtyping path must exist to a non-general-purpose architectural style.

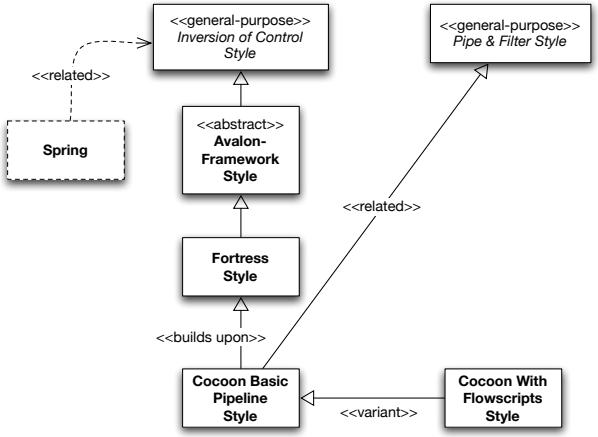

Figure 7.4: MidArch Style Taxonomy of Cocoon-related Styles

Type	Stereotype
General-purpose	«general-purpose»
Abstract	«abstract»
Concrete	*no stereotype*

Table 7.2: Stereotypes for Styles in UML MidArch Style Taxonomy Diagrams

3. Variant subtypes may only exist between concrete architectural styles.

4. A variant may only be variant subtyped.

A MidArch Style Taxonomy is *well-formed* if it adheres to all of these constraints.

7.5.4 UML Representation of a MidArch Style Taxonomy

We represent a MidArch Style Taxonomy as a UML Class diagram with specific stereotypes. MidArch Styles are modelled using Classes, which are annotated by different Stereotypes according to the types defined in Section 7.5.1 (see Table 7.2). Style relationships are modelled using Generalizations. Again, the types of relationships defined in Section 7.5.2 are distinguished using different Stereotypes (see Table 7.3).

Example An example taxonomy taken from one of our case studies can be seen in Figure 7.4, which is described in detail in Section 10.1.

Type	Stereotype
inherits from	*no stereotype*
variant of	«variant»
related to	«related»
builds upon	«builds-upon»

Table 7.3: Stereotypes for Style Relationships in UML MidArch Style Taxonomy Diagrams

7.5.5 Conceptual and Logical MidArch Style Taxonomies

As noted in the introduction of Section 7.5, we distinguish conceptual and logical MidArch Style Taxonomies. Conceptual MidArch Style Taxonomies are not bound to concrete formal style descriptions, while logical ones are specific to a style-enabled ADL. For a given set of MidArch Styles, there is only one conceptual MidArch Style Taxonomy, but different logical MidArch Style Taxonomies may exist. When moving from the conceptual to the logical level, the relationships between the styles must be mapped to representations of these relationships in the style descriptions. A logical MidArch Style Taxonomy should reflect these representations, while retaining information on relationships from the conceptual MidArch Style Taxonomy that cannot be represented in the style-enabled ADL.

In the following, first an overall method for defining conceptual and logical MidArch Style Taxonomies, which is interwoven with the derivation of individual styles and style descriptions (see Sections 7.6 and 7.7), is described (Section 7.5.5.1). The method and the underlying concepts are discussed in Section 7.5.5.2.

7.5.5.1 Defining Conceptual and Logical MidArch Style Taxonomies

To provide an overview of this discussion, the dependencies of the artefacts involved in the creation of conceptual and logical MidArch Style Taxonomies are shown in Figure 7.5. From intangible knowledge of General-purpose Architectural Style Concepts on the one hand and concrete Middleware Platforms on the other hand, both informal style descriptions and a preliminary conceptual MidArch Style Taxonomy are created. Then, the informal Style Descriptions are formalised into descriptions in a style-enabled ADL. From the set of Formal Style Descriptions, a logical MidArch Style Taxonomy can be automatically derived (see Section 7.7.1.2). A derived conceptual MidArch Style Taxonomy may be extracted from this logical MidArch Style Taxonomy. The derived and preliminary conceptual MidArch Style Taxonomies may then be reconciled into a Reconciled conceptual MidArch Style Taxonomy.

In Figure 7.6, a situation with multiple alternative formalisations of the same MidArch Styles is shown. To illustrate this, we assume that the

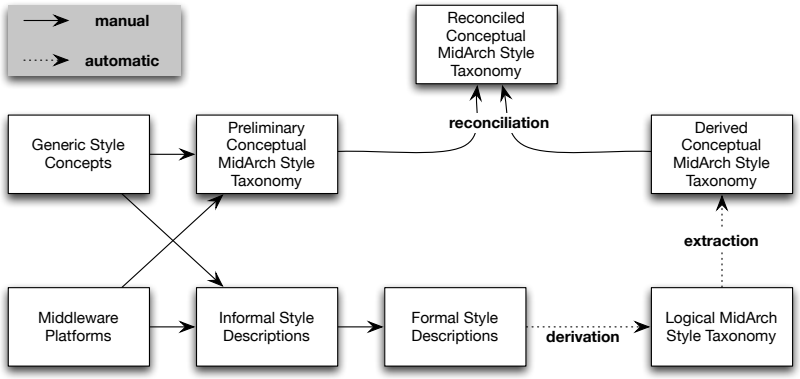

Figure 7.5: Process-oriented Illustration of the Relationship of Conceptual and Logical MidArch Style Taxonomies

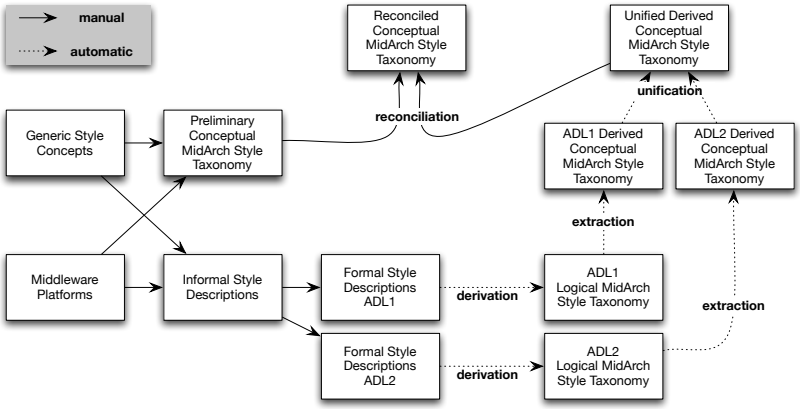

Figure 7.6: Method including alternative MidArch Style Descriptions

alternative formalisations make use of different style-enabled ADLs[3]. In this case, each set of Formal MidArch Style Descriptions yields its own logical and derived conceptual MidArch Style Taxonomy. A unification algorithm may be used to generate a unified derived conceptual MidArch Style Taxonomy before the reconciliation step, if the alternative derived conceptual MidArch Style Taxonomies deviate from each other.

7.5.5.2 Discussion

A MidArch Style Taxonomy may either be defined on the conceptual level or on the logical level. The conceptual level is independent from the style descriptions and thus from the chosen style-enabled ADL.

While a logical, i.e. description-level, MidArch Style Taxonomy does not exhibit the internal details of the formal model of each style, it exposes the formal inheritance relationships between the style descriptions. It is thus dependent both (a) on the chosen style-enabled ADL as well as (b) on further design decisions taken in mapping the informal style description to that style-enabled ADL.

Desirable Characteristics of Conceptual MidArch Style Taxonomies and of Formal MidArch Styles As is the case with design models in general, it is not always desirable to map logical artefacts one-to-one to elements of the conceptual design. Implementation decisions are not subjected to the same requirements as design decisions are. Thus, the design model should not provide simply a reduced visualisation of the implementation artefacts. While the procedure as shown in Figure 7.5 is feasible, it is not always desired. The reconciliation of the derived conceptual MidArch Style Taxonomy with the preliminary MidArch Style Taxonomy cannot be automated in the general case, but it is subject to taxonomic design decisions that depend on the individual taxonomy.

In general, on the conceptual level the concepts and their relationships should be reflected as clearly as possible. On the logical level, clarity may be traded for simplicity of the manifestation (implementation) of the concepts.

In the case of MidArch Style Taxonomies, one example has already been given above: Mapping conceptual specialisation relationships to logical specialisation relationships may result in complicated style descriptions (or might be impossible due to restrictions in the expressiveness of the style-enabled ADL). Another reason to deviate from the conceptual model may be that, if an intermediate style description is only used once, it may not be worth the effort to model it explicitly as a separate style description.

[3] Different taxonomies may also result from different design decisions in mapping a conceptual-level style to a style description in the same style-enabled ADL.

7.6 General MidArch Style Modelling Method

In this section, we describe the general method for modelling concrete MidArch Styles. The basis for the method is the MidArch Metamodel (Section 7.6.1), which defines the style and architecture modelling constructs of the MidArch approach on a conceptual level.

The method is not defined in all the details that were required to make its application repeatable and its result deterministic. Such an endeavour were not possible at all, since the derivation of a style description is a design activity which requires the designer to make design decisions. Since the result of the derivation is more abstract than designing an individual architecture, it relies on the expertise of the designer to an even greater degree. However, by the description in Section 7.6.2 a framework for the process is established.

7.6.1 MidArch Metamodel

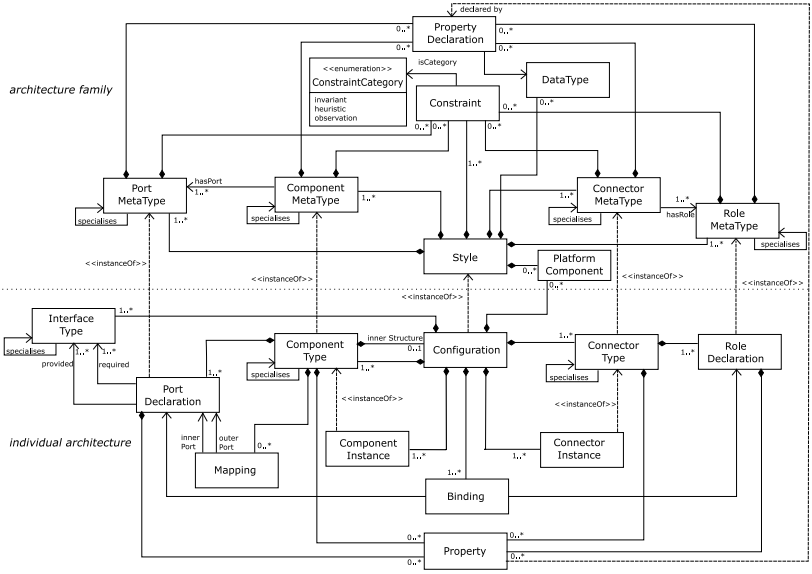

Figure 7.7: The MidArch Metamodel

The MidArch Metamodel defines the MidArch Style Modelling Constructs (cf. Figure 7.1). An overview of the MidArch Metamodel is given through the UML Class Diagram in Figure 7.7. In the text, we use the convention of

using a sans-serif font for the concepts in the metamodel, e.g., Configuration. It is partitioned by a dashed horizontal line, which separates the concepts concerning the family of architectures modelled by a Style (above the line) from the concepts concerning an individual architecture represented by a Configuration (below the line). Style and Configuration are thus the basic concepts of each level. We briefly discuss both partitions in Sections 7.6.1.1 and 7.6.1.2.

Metatypes vs. types Several concepts in the metamodel are typed, i.e. they are classified within a specialisation hierarchy. The MidArch Metamodel makes a distinction between metatypes and (regular) types: Architectural styles specify component and connector metatypes. Metatypes are simply unary relations on the sets of components and connectors [cf. LE MÉTAYER, 1996], as opposed to component and connector types, which specify complete interfaces or protocols. However, constraints on the interactions of components may be imposed by the metatype.

7.6.1.1 Architecture Family Level

An architectural style, or in our terms a MidArch Style is at the core of the style modelling level. It has a unique name and contains the following elements:

- ComponentMetaTypes, PortMetaTypes, ConnectorMetaTypes and RoleMetaTypes, which define the style vocabulary. PortMetaTypes are associated with ComponentMetaTypes, while RoleMetaTypes are associated with ConnectorMetaTypes. These only serve as a coarse classification of elements, and do not specify detailed interfaces, which are assumed to be application-specific.
- Constraints restrict the valid individual architecture level compositions (i.e., Configurations) of the elements based on that vocabulary.
- DataTypes are used in the definition of architectural properties. Any metatype may declare some property via a PropertyDeclaration. These can be used to express quality characteristics, for example [cf. FROLUND and KOISTEN, 1998]. For example, the Filter ComponentMetaType in the pipe-and-filter style may declare a "Throughput" property, whose value is then specified by the instances of that metatype.
- PlatformComponents are components that are supplied by the platform whose services can be used by the application-level components of individual architectures.

7.6.1.2 Individual Architecture Level

An individual architecture in the logical component structure view is represented by a Configuration, which is declared to conform to exactly one Style.

Concept	Metatype	Type	Instance
Component	ComponentMetaType	ComponentType	ComponentInstance
Connector	ConnectorMetaType	ConnectorType	ConnectorInstance
Port	PortMetaType	PortDeclaration	none
Role	RoleMetaType	RoleDeclaration	none

Table 7.4: Metatypes and their instances

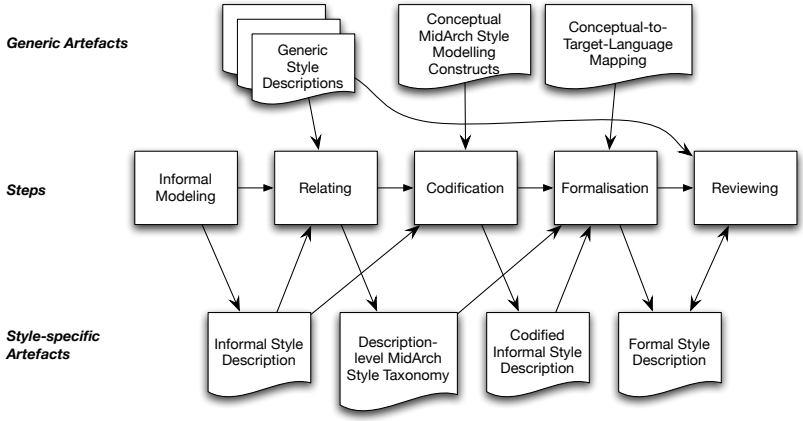

Figure 7.8: General MidArch Style Modelling Method and Artefacts Involved

Application-specific types of components and connectors for an individual architecture are specified and instantiated on the system level. Compositions of components, connectors, and their subordinate ports and roles can only be specified on the system level. The hierarchy of metatypes, types and instances is shown in Table 7.4. Interfaces, i.e., lists of operation signatures, may be specified for ports, and component ports are bound to connector roles.

7.6.2 Step-wise Procedure

As illustrated in Figure 7.8, a MidArch Style can be modelled in five steps:

Informal Modelling Informally describe all style-relevant aspects based on the available documentation of the middleware product and existing expert knowledge. This yields a list of requirements for the style specification in natural language.

Relating Identify possible specialisation relationships to general-purpose styles.

Codification Map the informal requirements to the MidArch Style Modelling Constructs defined in the MidArch Metamodel. The result of this intermediary step, if explicitly written down, can be written down as a list of construct instance names for each modelling construct.

Formalisation Formulate a style from these elements in the chosen ADL as an extension to the general-purpose styles identified in the "relating" step.

Reviewing Check for the consistency of the resulting style, most importantly whether it is instantiable. If necessary, relieve from strict logical specialisation relationships.

7.7 Style-enabled ADLs

A style-enabled ADL consists of (a) a formal language (target language) that allows the specification of MidArch Styles, comprising both syntax and semantics, and (b) a well-defined mapping of the MidArch Modelling Constructs to language elements of the target language. As discussed in Section 7.4, we use existing ADLs as target languages in our research. However, for some candidate target languages (which is the case for the UML, for example) the semantics of a target language are not sufficiently well-defined. In this case we provide more refined semantics to the degree necessary. The mapping of the MidArch Metamodel involves design decisions with several alternatives for each target language, so we may consider the same target language as different style-enabled ADLs differing in the specifics of the mapping.

7.7.1 Acme/xAcme Mapping

While the MidArch Metamodel was based on the Acme grammar, it embodies a more specific architecture modelling approach than Acme prescribes in general.

Depending on the source, the language constructs used for the definition of invariants and heuristics are sometimes considered part of Acme themselves or of an extension called Armani [Monroe, 2000a]. For the sake of simplicity, we use the former notion. Internally, we use xAcme because it may be easier processed in tools as it is XML-based, but for human readability, we consider the traditional Acme notation more convenient.

Most of the conceptual modelling elements may be mapped to Acme in a straightforward way. An architectural style is modelled as a family. Metatypes and types are both mapped to (component, connector, role

AND PORT) TYPE declarations: all types declared within a family are considered metatypes, those declared in system declarations are considered types. This solution is not perfect, since it does not allow to declare simple types (non-metatypes) for middleware layer components. Further architectural rules are modelled as INVARIANTS.

Currently, one restriction is that we cannot model "related to" relationships in Acme. This could be remedied by annotating the sub-style accordingly, and defining a property of the style that declares the extended style(s) and a specification of the details of the relationship.

The mapping of the MidArch Metamodel to Acme (respectively, the definition of the MidArch Metamodel in the style of Acme's implicit metamodel) was applied in BORNHOLD [2006].

7.7.1.1 Deviations from Acme

The MidArch Metamodel deviates from the Acme language in some ways. These deviations have been introduced for two reasons: first, to better suit the context of the MidArch project, and second, to ease the mapping to the UML. The deviations are:

- Acme does not distinguish metatypes and types.
- PlatformComponents are not contained in Acme, since they are specific to the middleware-oriented architectural styles defined in the MidArch project.
- We assume that connectors are provided atomically by the platform and encapsulate application-independent functionality. MidArch does therefore not allow to specify an internal structure of a connector.
- Interfaces do not exist as separate entities in Acme[4]. In Acme, the provided interface is immediately part of the port type definition. Interfaces are not specified at the style level, as we assume they contain application-specific information.
- Acme distinguishes only invariant and heuristic constraints.
- MidArch distinguishes between property *declarations* in styles and *instantiated* property values in configurations.

7.7.1.2 Derivation of a Logical MidArch Style Taxonomy from Acme Style Descriptions

Similarly to reverse engineering UML Class Diagrams from object-oriented code [KOLLMAN et al., 2002], a logical MidArch Style Taxonomy may be

[4] However, they are essential constituents of UML PORT definitions.

algorithmically derived from a set of style-enabled ADL style descriptions. If the style-enabled ADL supports specialisation relationships between style descriptions, style descriptions resp. specialisation relationships can be directly mapped to classes resp. specialisation relationships in the logical MidArch Style Taxonomy.

This is the case for Acme. Style declarations have the following syntax:

```
family STYLE [ extends BASICSTYLE { , BASICSTYLE } with ] {
    ...
}
```

By parsing a set of style descriptions, the style relationships can be identified.

However, only strict, variant and builds-upon relationships are represented this way in Acme, and they cannot be distinguished syntactically. Thus, a derived logical MidArch Style Taxonomy contains only one type of style relationships and entirely lacks "related to" relationships.

7.7.2 UML Mapping

The second language we mapped the MidArch Metamodel to is the UML. As discussed in Section 7.4, after the straightforward mapping to Acme, it seemed necessary to define a mapping to another target language: On the one hand, to demonstrate the language independence of the metamodel, at least two target languages must be used. In addition, for practical reasons, a better opportunity to leverage existing tools and modelling expertise can be expected with using the UML. The UML is a very complex language defining several diagram types that can be used for modelling software architectures. In order to systematically define a mapping, some preliminary design decisions needed to be made, these were: (1) to use UML Component diagrams as a means to model logical component structure views and (2) to model each architectural style through a profile that restricts the valid UML Component diagrams[5].

In version 2.0 of the UML, the OMG introduced new diagram types and a refined metamodel aimed at modelling software architectures, in particular the component-and-connector architectural view. This step has been made as part of a general step towards broader applicability of the UML and as a response to criticism that the component modelling capabilities of the UML 1.x were not suitable for modelling software architectures [see, e.g., Kandé and Strohmeier, 2000; Medvidovic et al., 2002]. However, certain architecture modelling constructs such as typed connectors and architectural

[5] Strictly speaking, a UML diagram is only a secondary artefact to a UML model. The model elements are not clearly divided by the diagram types they may occur in. To be exact, one needed to speak of a "profile that restricts the valid UML models which contain elements occurring in UML Component diagrams".

styles are still not part of the UML 2.0 [Abi-Antoun and Medvidovic, 1999; Garlan et al., 2002; Goulao and e Abreu, 2003].

Over the 1990s, ADLs have been proposed for modelling software architectures from various viewpoints. The architecture modelling capabilities of the UML have been influenced by this work. The ADL Acme [Garlan et al., 2000] was used as the basis for the inclusion of architecture modelling capabilities into the UML [Selic, 2005, p. 203].

We present a precise mapping of the MidArch architecture modelling approach to the UML. MidArch was chosen because it is used in the application context of the MidArch Design Method (Chapter 8) and because of the similarity to ADLs such as Acme. When defining the mapping, we encountered several syntactic and semantic problems in incorporating ADL-oriented modelling constructs and the modelling constructs of the UML.

The UML provides a precise syntax for architectural concepts, but the semantics and pragmatics of their use exhibit many variation points. In order to define a specific effective UML-based modelling method, these variation points must be specified. The Object Constraint Language (OCL), which is used by the UML, plays an important role in this specification, as it is used to define precise conditions and restrictions for modelling software architectures consistently. The MidArch-to-UML/OCL mapping results in the definition of such a UML-based modelling approach, which can be used to model directly in the UML: the UML/MidArch Modelling Method.

An architectural style is a model of the structural aspects of a family of related software architectures. The UML does not natively provide a modelling construct comparable to architectural styles. Thus, UML/MidArch enables style-based architecture modelling with the UML in the first place. Style-based architecture modelling has the following benefits:

- It enables style conformance checks.
- It increases the conceptual integrity of architectural descriptions.
- The modelling of system architectures can be guided by the style, which results in greater modelling efficiency.

Compared with ADLs that provide a style modelling construct, UML/MidArch has the benefit that the underlying notation is more widely known, commercial tool support is available, and the integration of the component-and-connector view with other views in a single notation is possible.

Overview First, an overview of the mapping is presented in Section 7.7.2.1. More details on the modelling approach and its design and rationale can be found in Marwede [2007]. It is followed by an examples illustrating the approach (Section 7.7.2.2). The experiences with the modelling approach, drawing from the case studies, are discussed in Section 7.7.2.3.

7.7.2.1 Mapping Definition

We use the convention of using small capitals for the metaclasses of the UML Superstructure [OMG, 2005c, 2007b][6], e.g., COMPONENT.

As noted above, one important goal of the mapping is to enable style conformance checks. Another goal is to allow the reuse of UML tools, which restricts our modelling freedom to employing the lightweight extension mechanisms of the UML, i.e., specifying PROFILES containing STEREOTYPES and CONSTRAINTS.

Our approach builds upon the fundamental design decision to map each Style onto a separate UML PROFILE, and each Configuration conforming to a Style onto a UML model that primarily uses the metaclasses for the Component Diagram which applies the corresponding PROFILE (see Figure 7.9). Thus, a Style is modelled essentially on the MOF M2 level, while Configurations are on the MOF M1 level. Due to the MOF metamodelling principles, they cannot be modelled entirely on the M2 level.

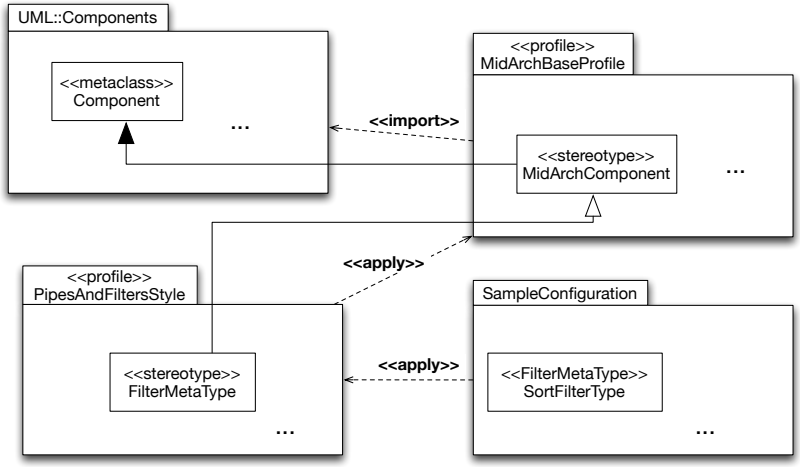

Figure 7.9: Relationships of the resulting UML/MidArch Models

Design Principles For each of the constructs of the MidArch Metamodel, we investigated several mapping alternatives. These alternatives were gathered

[6] We essentially based our work on the UML 2.0 Specification, but consulted the newer UML 2.1.1 Specification in later phases of our work in case of ambiguities in the UML 2.0 Specification.

from the literature, or they were found through our own analysis of the UML Superstructure. The decision on the adopted mapping was subjected to several design principles to ensure the quality of the mapping:

- The mapping should retain the syntactic and semantic properties of each MidArch modelling construct.
- The mapping should retain the consistency between the MidArch modelling constructs. For example, ComponentMetaType must be mapped such that an instance of the UML target entity corresponds with ComponentType.
- The mapping should preserve the conceptual integrity of the UML.
- The model complexity of the target constructs should be as low as possible among the alternatives that conform with the other design principles [cf. Garlan et al., 2002, p. 32], i.e., ideally, each MidArch modelling construct should be mapped to a single UML metaclass.

Mapping of the Modelling Constructs Table 7.5 gives an overview of the mapping of the MidArch modelling constructs discussed in Section 7.6.1 to entities of the UML. It encompasses both the style and system levels.

For components and connectors, we start defining the mapping on the middle (i.e., type) level and then derive appropriate mappings for the metatype and instance levels. All metatype concepts are mapped to Stereotypes for the corresponding type respectively declaration construct.

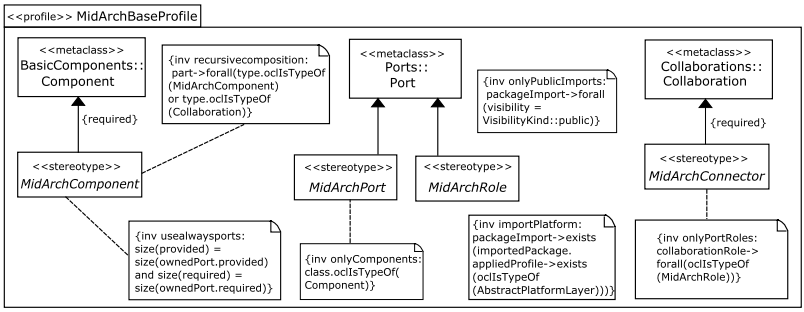

Figure 7.10: The MidArch Base Profile

MidArch Base Profile As an auxiliary means we defined a MidArch Base Profile, which is extended by each style profile. It serves to reduce the effort for modelling Styles and to harmonise the resulting Profiles. The Base Profile is shown in Figure 7.10. In addition to the Stereotypes for each of the modelling constructs the following OCL constraints are defined:

MidArch modelling construct	UML entity
Style	Profile
ComponentMetaType	Stereotype for Components
ComponentType	Component
ComponentInstance	InstanceSpecification
Mapping	Connector (delegation)
PortMetaType	Stereotype for Ports
PortDeclaration	Port within a Component
InterfaceType	Interface
ConnectorMetaType	Stereotype for Connectors
ConnectorType	Collaboration
ConnectorInstance	CollaborationUse
RoleMetaType	Stereotype for Ports
RoleDeclaration	Port within a Collaboration
Binding	Dependency
Constraint	Constraint
PropertyDeclaration	*tag definition*
Property	*tagged value*
PlatformComponent	Component
Configuration	Package
Instantiation relationship of Configurations and Styles	ProfileApplication

Table 7.5: Mapping of MidArch modelling constructs to UML entities

recursiveComposition The parts of a MidArchComponent always are MidArchComponents again or MidArchConnectors.

useAlwaysPorts A MidArchComponent always interact through its Ports.

onlyComponents A MidArchPort is always typed by a Component.

onlyPortRoles The collaboration roles of a MidArchConnector are required to be MidArchRoles.

onlyPublicImports Only public imports are allowed.

importPlatform A Package which applies a MidArch Style Profile has to import a Package which apply a Profile called AbstractPlatformLayer which represents the provided services by the underlying platform.

Exemplarily, we discuss the rationale of the chosen mapping of ConnectorTypes and RoleTypes in detail. More information on the the mappings of these and the other concepts can be found in Marwede [2007].

Mapping the Connector and Role Metaclasses While ComponentTypes can be mapped straightforwardly to Components, more problems arise for ConnectorTypes. We investigated several options, before making the decision

to map ConnectorTypes to COLLABORATIONS:

While the UML offers a CONNECTOR metaclass, it is not a first-class entity, as a connector is always owned by a CLASSIFIER (which is a superclass of COMPONENT). Moreover, connectors cannot be typed in the UML.

ASSOCIATION and INTERFACE are not suitable because they lack a means to specify roles.

CLASS and COMPONENT are more appropriate. In combination with them, INTERFACES can be used to describe RoleTypes. However, this mapping strategy still bears some problems: It conflicts with the idea to use provided and required INTERFACES for application-specific interfaces of ports and it is not possible to define two identical roles for the same ConnectorType. Furthermore, the use of COMPONENT for ConnectorTypes conflicts with the semantics of the UML.

Mapping ConnectorTypes to ASSOCIATIONCLASSES has the disadvantage of a high model complexity of the target structure, because additional ASSOCIATIONS are required to connect the roles of the connector (e.g., defined as PORTS) to the ports of components.

The metaclass COLLABORATION is best suited as the target entity because roles can be defined through CONNECTABLEELEMENTS. Moreover, COLLABORATIONS can be typed because the metaclass inherits from CLASSIFIER.

Mapping Role Bindings As mentioned above, the mapping of ConnectorTypes to COLLABORATIONS implies the mapping of RoleDeclarations to CONNECTABLEELEMENTS. Since a role specifies the required set of features of a participating port and the metaclass PORT inherits from CONNECTABLEELEMENT, it is appropriate to use the metaclass PORT for describing roles.

Furthermore, the metaclass COLLABORATIONUSE serves as ConnectorInstance and provides a means to define role bindings with a set of DEPENDENCIES.

UML/MidArch Modelling Method for Styles Now we are able to describe the UML/MidArch Modelling Method, which results from the mapping described above. The following steps must be taken in order to specify a style profile for a Style. These steps are applicable regardless of whether an existing Acme specification is translated to the UML, or the formalisation of the style is performed in the UML in the first place.

1. Create and denominate an empty PROFILE.

2. Add a PROFILEAPPLICATION to the MidArch Base Profile.

3. Import the abstract base stereotypes from the MidArch Base Profile.

4. Define STEREOTYPES for ComponentMetaTypes, PortMetaTypes, ConnectorMetaTypes and RoleMetaTypes which inherit from the respective abstract base stereotypes.

5. Define PropertyDeclarations as *tag definitions* of the defined STEREOTYPES.

6. Transcribe the style's Constraints into OCL and manifest as CONSTRAINTS.

7. Define additional OCL CONSTRAINTS to specify the associations of ComponentMetaTypes and their PortMetaTypes as well as ComponentMetaTypes and their RoleMetaTypes. This is necessary because it is not allowed to define associations between stereotypes when creating PROFILES.

7.7.2.2 Example

In this section, we describe an example of applying the UML/MidArch Modelling Method (Section 7.7.2.1) to the pipes-and-filters style. First we present the style itself, and then an example configuration that conforms to the style.

```
1  Family PipesAndFiltersFam = {
2      Port Type inputT = {}
3      Port Type outputT = {}

5      Role Type sourceT = {}
6      Role Type sinkT = {}

8      Component Type Filter = {
9          Port input : inputT =  new inputT extended with {};
10         Port output : outputT =  new outputT extended with {};
11         invariant Forall p : port in self.Ports |
12             satisfiesType(p, inputT) OR satisfiesType(p,
                   outputT)
13       <<label : string = "Only␣ports␣of␣type␣inputT␣or␣outputT␣
             are␣allowed";>>;
14     }

16     Connector Type Pipe = {
17         Role source : sourceT =  new sourceT extended with {};
18         Role sink : sinkT =  new sinkT extended with {};
19     }
20 }
```

Listing 1: A simplified Pipes-and-Filters Style in Acme

Listing 1 shows a simplified version of the pipes-and-filters style description that comes with AcmeStudio [SCHMERL and GARLAN, 2004], which leaves out the initial and final `DataSource` and `DataSink` component metatypes

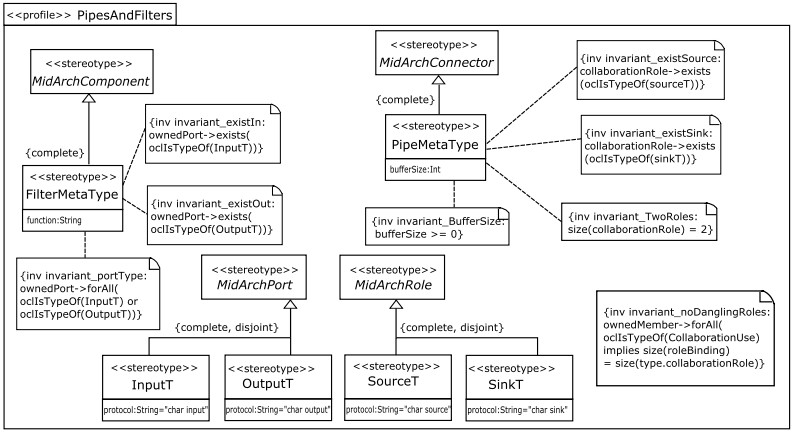

Figure 7.11: A Profile for the Pipes-and-Filters Style

as well as most constraints. The latter are only shown in the profile that resulted from the mapping shown in Figure 7.11: It defines a connector metatype PipeMetaType with roles SourceT and SinkT, as well as a component metatype FilterMetaType with ports InputT and OutputT. The following OCL constraints are defined:

portType FilterMetaType components may only have InputT or OutputT ports.

existIn/existOut FilterMetaType components have at least one InputT resp. OutputT port.

pufferSize The buffer size of a PipeMetaType connector must be at least 0.

twoRoles A PipeMetaType connector has exactly two roles.

existSource/existSink A PipeMetaType connector has at least one SourceT resp. SinkT role.

noDanglingRoles A PipeMetaType connector has no unconnected roles.

Example Configuration in the Pipes-and-Filters Style Figure 7.12 shows a UML package containing an example configuration that is member of the pipes-and-filters style family. It declares two filter types (upper left part) and one pipe type (upper right part) including their ports and roles and default values of their architectural properties. Finally these types are used in a small configuration (lower part) that demonstrates how to bind ports

and roles within our approach.

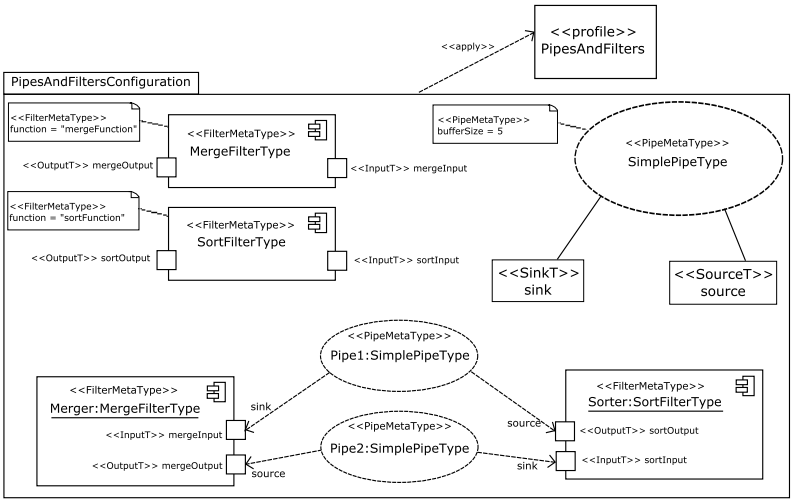

Figure 7.12: An example configuration in the Pipes-and-Filters Style

7.7.2.3 Experiences

Experiences with the MidArch/UML modelling approach can be divided into those that resulted from defining the mapping in the beginning and those that resulted from applying and using the approach.

Definition of the Mapping Two issues are at the core of the experiences in the definition of the mapping:

Basic Approach The chosen approach to map Styles to PROFILES is the only lightweight possibility to achieve the requirement of enabling style conformance checks with UML/OCL tools such as those required for MDA approaches. There have been proposals for modelling design patterns as M1-level UML models [FRANCE et al., 2004; OMG, 2004a], which cannot be used to model styles, since their instantiation cannot be specified within the UML. If one chose some metaclass X to represent styles, configurations would need to be represented as INSTANCESPECIFICATIONS referring to instances of X. The semantics of these INSTANCESPECIFICATIONS cannot be specified in the UML

specifically for some referenced metaclass. While the UML would allow an extension with user-defined semantics of InstanceSpecifications, standard UML/OCL tools would not be able to interpret these semantics and perform the style conformance check.

Implicit representation of metatype relationships Since the UML does not allow the specification of Associations between Stereotypes, any relationships between metatypes, which are explicit in the metamodel, can only be represented indirectly through Constraints that are associated with the respective Stereotypes. These Constraints are specified on the M2 level, and require the existence of certain Associations on the M1 level.

Therefore, a UML/OCL mapping of a Style contains more Constraints than the original style Constraints. These could be distinguished by applying some naming convention.

Application of the Mapping In the application of the UML/MidArch approach, we noted two issues:

Complexity For abstract architectural styles, such as the pipes-and-filters style from the example, which define only a small vocabulary and few constraints, the approach is applicable well. It produces profiles whose size is manageable. We also modelled more complex middleware-oriented architectural styles (e.g., for the Avalon Framework [Giesecke et al., 2007a]). For these, the profiles are more complex, and difficult to arrange within a single diagram. However, this is a general challenge not only of the UML but of any graphical modelling notation.

Representation of Allowed Port/Role Binding When Acme family specifications make explicit statements on the allowed binding of ports and roles on the metatype level, these can only be translated indirectly to UML/MidArch, as already noted in Section 7.7.2.3. In part, this is due to a variation point in Acme. The MidArch Metamodel assumes that RoleMetaTypes declaratively specify conditions concerning which PortMetaTypes may connect to them. In addition, it assumes implicitly that a Port's required Interfaces must be provided by the Port at the other end of the attached Connector, which is not required in Acme. Since this relationship is only specified at the system level, it can be explicitly expressed in the M1 UML model.

7.8 Summary

In this chapter, we discussed the relationship of middleware platforms, middleware products, and architectural styles (Section 7.1), which has both a static and a dynamic aspect.

Based on several assumptions on architecture descriptions and architectural styles (Section 7.2), we proposed requirements (Section 7.3) for architectural style modelling, which concern the following aspects: Identification of architectural concerns that are relevant for style modelling, the ability to model implicitly layered software architectures, the ability to model relationships of middleware-oriented architectural styles to general-purpose architectural styles, the need to consider combinations of multiple architectural styles and the language independence of the approach. For each of these requirements, we discussed design alternatives and made design decisions. Section 7.4 discusses the design decision that the modelling approach is defined independently from a concrete notation and then mapped to existing notations.

Based on these design decisions, the General MidArch Modelling Approach was defined in Sections 7.5 and 7.6. It includes an approach to model taxonomies of architectural style (MidArch Style Taxonomies) and the MidArch Metamodel, which defines the modelling constructs for middleware-oriented architectural styles on a conceptual level. This conceptual metamodel needs to be mapped to concrete ADLs to be applied.

In Section 7.7, we described the mappings of the MidArch Metamodel to the ADL Acme and to the UML 2. Since the MidArch Metamodel and the Acme grammar are closely related, the former mapping is straightforward. We defined a method for modelling architectural styles and style-based architectures on the basis of the UML 2. This indicates that the UML 2 in combination with OCL is suited for modelling the component-and-connector view of software architectures. The advancement of the Component metaclass and the introduction of the new Port metaclass have significantly improved UML's capabilities to model component-and-connector architectural views compared to the UML 1.x.

The mapping could be adapted to the original Acme language with only slight modifications, so that an automatic translation from Acme to the UML-based modelling approach and vice versa should be implementable, since the deviations to MidArch are only minor. This could be done either through XSL transformations between xAcme and XMI, or through model-to-model transformations between a MOF-based representation of Acme and the UML.

As a result of the mapping we defined the UML/MidArch Modelling Method for modelling styles and style-based component-and-connector configurations, which is of practical relevance on its own. It forms the basis of style evaluations in the context of the MidArch Design Method (see Chapter 8).

A design method *exploits* MidArch Styles if the design method involves some activity that has MidArch Styles artefacts as an input or output artefact (*MidArch Style-oriented Design Activity*). A MidArch Style-oriented Design Activity is a software process activity that has MidArch Styles artefacts as an

input or output artefact. More specifically, it uses a MidArch Style Taxonomy for selecting a MidArch Styles, uses a Formal MidArch Styles Description for modelling an architecture, or produces respectively refines Formal MidArch Styles Descriptions, a MidArch Style Taxonomy or information enriching these artefacts.

In Chapter 8, we propose the MidArch Design Method that involves several MidArch Style-oriented Design Activities.

A tool which provides access to a MidArch Style Taxonomy and the evaluation results of former MidArch instances can support the selection of a MidArch Style in the context of the MidArch Design Method. We refer to such a tool as a *MidArch Repository Tool* (Section 11.1).

8 The MidArch Design Method for Early Middleware Selection based on MidArch Styles

This chapter describes the exploitation of MidArch Style modelling in an overall design method, the *MidArch Design Method*. The core idea of the MidArch Design Method is to use MidArch Styles as a vehicle for selecting a suitable middleware platform (respectively, a suitable MidArch Style) early within a software project. The early selection is achieved by making the selection decision on the architectural level. In the long term, the quality of the selection is improved by enabling design knowledge to be transferred among instances of the MidArch Design Method. The relevant design knowledge refers to the influence of middleware platforms on the architecture-level system quality.

Figure 8.1 shows the elements of the MidArch Approach. The knowledge gained in evaluating architectures conforming to some style can be reused and exploited in further instances of the MidArch Design Method (MidArch instances), i.e. conducting a MidArch instance produces design knowledge both for the current project and for future projects. A tool, the MidArch Repository, supports creating the MidArch Style Taxonomy, amending it with evaluation results, and guiding developers in selecting suitable candidate MidArch Styles using the Taxonomy.

Overview The chapter is structured as follows: First, assumptions underlying the design of the MidArch Design Method are introduced in Section 8.1. An overview of the activities of the MidArch Design Method is then given in Section 8.2, the details are found in Appendix A. Then, the specification of the MidArch Repository is given in Section 8.3. Examples of MidArch instances are discussed in Section 8.4. The summary in Section 8.5 concludes the chapter.

8.1 Assumptions

First, we make two assumptions that are typically made in software architecture research [e.g., Reussner et al., 2003]:

- quality requirements can be stated on the architectural level

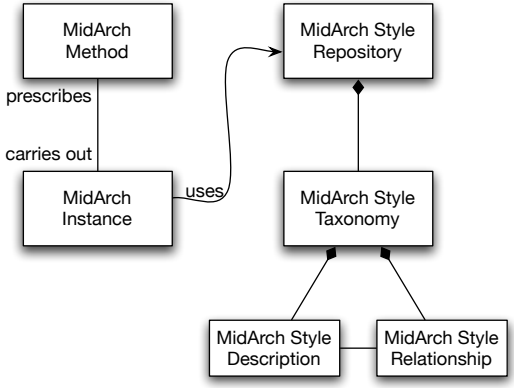

Figure 8.1: Elements of the MidArch Approach

- quality characteristics of a software system can be evaluated on the architectural level

The first assumption means that the quality requirements can be expressed by referring to concepts used for architectural description only. The second assumption extends the first assumption and is much more restrictive: it additionally requires that given an architectural description and a formulation of quality requirements on the architectural level, a proposition may be made on whether the described architecture satisfies the requirements. Such a proposition cannot be made with absolute confidence, since many implementations with varying quality properties conform to the same architecture. This assumption implies that the differences between implementations conforming to different architectures outweigh the differences among the implementations conforming to the same architecture. The soundness of the second assumption is particularly difficult to validate, since it does not merely require a formal proof on the relationship of implementations and architectural descriptions (which would probably fail anyhow), but empirical observations on the influence of architectural descriptions on the development process.

Additionally, we make two assumptions that are more specific to the MidArch Design Method:

- the choice of middleware platform significantly influences topological constraints

- the topological constraints of a MidArch Style significantly influence quality characteristics

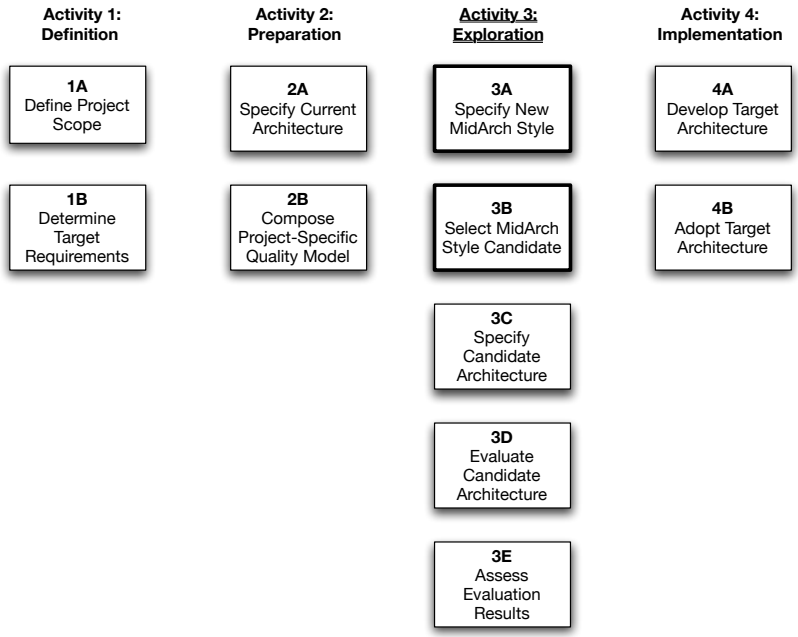

Figure 8.2: Activities and Tasks of the MidArch Design Method

As discussed in Section 3.1.1.2, middleware-intensive software systems are defined by the property that their logical component structure is significantly influenced by the choice of middleware platform. To be more specific, this significant influence means that given a configuration c_A conforming to a MidArch Style A, choosing a different middleware platform (which is the same as choosing a different MidArch Style B) will yield a configuration c_B that is different from c_A because the topological constraints defined by B differ from those defined by A. The difference must be visible in the metrics of the quality model that is used for architectural evaluations (see Section A.3.2.2).

8.2 Activities of the MidArch Design Method

In Figure 8.2, the four activities of the MidArch Design Method and their tasks are shown. In this overview, no dependencies between the tasks are displayed. The tasks should not be thought of as being related in a waterfall-

like process, but in an iterative way. The activities are described briefly in the following. The tasks associated with each activity are defined in Section A.3 in the appendix.

Definition The scope of the project, i.e. the systems involved, and the goals to be achieved are defined.

Preparation A project-specific quality model (cf. Section 2.4) is developed, and the current architecture is modelled, if an architecture description does not yet exist in a suitable form.

Exploration This activity is central to the proposed method and is assumed to consume the majority of the effort required for an instance of the method. It involves the preselection of MidArch Styles, modelling of candidate target architectures that conform to the selected styles, and the evaluation of the resulting architectures. For evaluation, existing architecture evaluation methods are used. Finally, the evaluation results are assessed to decide whether a candidate target architecture (or a set of such) that allows to achieve the stated goals has already been found.

Implementation Based on the previous activity, the architecture to be implemented is defined, which might involve adapting the chosen target architectures for practical reasons, and the implementation is performed.

The core element of the method, and its most distinctive feature, is the style modelling approach which is part of the exploration activity.

The definition, preparation and implementation activities are structured straightforwardly, since each of their tasks occurs exactly once (or at most once) within a MidArch instance. The exploration activity is more complex, because of its iterative and exploratory character. The generic workflow through its tasks is shown in Figure 8.3. Concrete examples are provided in Section 8.4.

8.3 MidArch Style Taxonomy and Repository Specification

In this section, the Object-Z specification given in Section 5.4, where the basic entities have been defined that are used in the following definitions, is extended towards the specification of MidArch Taxonomies and a MidArch Repository.

However, only the abstract specification of the MidArch Repository is given. Work towards a concrete CASE tool supporting the MidArch Design Method (MidArch Repository Tool) is described in Section 11.1.

In Section 8.3.1, we give the specification of a MidArch Style Taxonomy. Then, in Section 8.3.2 the specification of style evaluation results is defined.

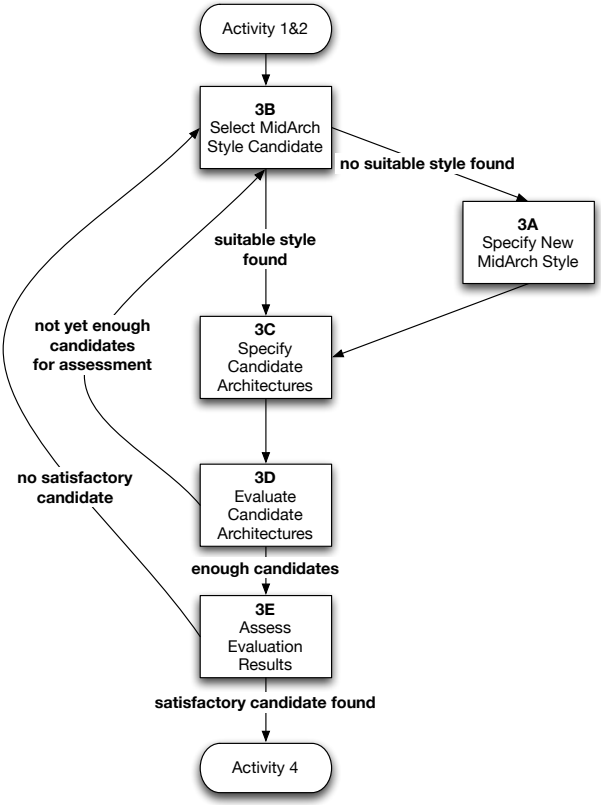

Figure 8.3: Workflow through the Exploration Activity

Section 8.3.3 presents the specification of the MidArch Repository, which stores both a MidArch Style Taxonomy and associated evaluation results.

Section 8.3.4 then describes the algorithm used to select a style from the repository by matching requirements against the stored evaluation results.

8.3.1 MidArch Style Taxonomies

The *Taxonomy* class is used to model a MidArch Style Taxonomy. It contains a set of styles and their relationships as discussed in Section 7.5. We omit the builds-upon relationship here for simplicity. It can be mapped to formal specialisation relationships for the purposes of the Z specification. Besides the direct relationships between styles in the taxonomy, indirect formal and informal (related-to) relationships (cf. Section 7.5.2) are derived (*isAnyFormalSuperstyleOf*, *isAnyInformalSuperstyleOf*). These relations are transitive, but not reflexive.

The relation *isCommonSuperstyleOf* determines whether two styles possess a common informal superstyle. For any two styles, the closest common superstyle is determined *closestCommonSuperstyle*). The mapping is defined by the last two formulae in the specification. The first formula states that the closest common superstyle of two styles must be their common superstyle in the first place. The second formula states that any common superstyle must be either a substyle of the closest common superstyle or it must be the closest common superstyle itself.

Taxonomy

$$
\begin{array}{l}
containedStyles : \mathbb{F}\ Style \\
isDirectInformalSubstyleOf : Style \leftrightarrow Style \\
isVariantSubstyleOf : Style \leftrightarrow Style \\
isDirectFormalSubstyleOf : Style \leftrightarrow Style \\
\Delta \\
isAnyInformalSubstyleOf : Style \leftrightarrow Style \\
isAnyFormalSubstyleOf : Style \leftrightarrow Style \\
isCommonSuperstyleOf : Style \leftrightarrow Style \times Style \\
closestCommonSuperstyle : Style \times Style \nrightarrow Style \\
\hline
\mathrm{dom}\ isDirectInformalSubstyleOf \subset containedStyles \\
\mathrm{ran}\ isDirectInformalSubstyleOf \subset containedStyles \\
isDirectFormalSubstyleOf \subset isDirectInformalSubstyleOf \\
isVariantSubstyleOf \subset isDirectFormalSubstyleOf \\
isAnyInformalSubstyleOf = isDirectInformalSubstyleOf^{+} \\
isAnyFormalSubstyleOf = isDirectFormalSubstyleOf^{+} \\
\forall\ a, b, c : Style \bullet c\ \underline{isCommonSuperstyleOf}\ (a, b) \Leftrightarrow \\
\quad a\ \underline{isAnyInformalSubstyleOf}\ c \land b\ \underline{isAnyInformalSubstyleOf}\ c \\
\forall\ a, b : Style \bullet closestCommonSuperstyle(a, b)\ \underline{isCommonSuperstyleOf}\ (a, b) \\
\forall\ a, b, c : Style \mid c\ \underline{isCommonSuperstyleOf}\ (a, b) \bullet \\
\quad closestCommonSuperstyle(a, b)\ \underline{isAnyInformalSubstyleOf}\ c \\
\qquad \lor\ c = closestCommonSuperstyle(a, b)
\end{array}
$$

Init

$$
\begin{array}{l}
containedStyles = \varnothing \\
isDirectInformalSubstyleOf = \varnothing \\
isVariantSubstyleOf = \varnothing \\
isDirectFormalSubstyleOf = \varnothing
\end{array}
$$

The operation *AddStyle* is used to add a new style to a MidArch Style Taxonomy and to update the relationship information.

AddStyle

$$\Delta(containedStyles, isDirectInformalSubstyleOf, isDirectFormalSubstyleOf, isVariantSubstyleOf)$$
$$style? : Style$$
$$formalSuperstyles?, informalSuperstyles?, variantSuperstyles? : \mathbb{F}\ Style$$

$$style? \notin containedStyles$$
$$\#variantSuperstyles? \leq 1$$
$$formalSuperstyles? \cap informalSuperstyles? = \varnothing$$
$$formalSuperstyles? \cap variantSuperstyles? = \varnothing$$
$$informalSuperstyles? \cap variantSuperstyles? = \varnothing$$
$$containedStyles' = containedStyles \cup \{style?\}$$
$$isDirectInformalSubstyleOf' = isDirectInformalSubstyleOf \cup \{super : Style \mid super \in informalSuperstyles? \cup formalSuperstyles? \cup variantSuperstyles? \bullet (style?, super)\}$$
$$isDirectFormalSubstyleOf' = isDirectFormalSubstyleOf \cup \{super : Style \mid super \in formalSuperstyles? \cup variantSuperstyles? \bullet (style?, super)\}$$
$$isVariantSubstyleOf' = isVariantSubstyleOf \cup \{super : Style \mid super \in variantSuperstyles? \bullet (style?, super)\}$$

8.3.2 Style Evaluations

In this specification package, we provide definitions of comparative measurements that represent differences between two evaluations on the one hand, and commonalities of two evaluations on the other hand.

For the specification, we assume that the same quality model is used for all architectural evaluations that are compared. This is a simplification to improve the understandability of the specification. Actually, the intersection of the sets of metrics used in the participating architectural evaluations must be used, which can yield different sets of metrics for each comparative measurement.

ComparativeMeasurement

$$referencedStyles : \mathbb{F}\ Style$$

Difference

ComparativeMeasurement

$$metricMeasurement : \mathbb{F}\ MetricMeasurement_2$$

$$\#referencedStyles = 2$$

Commonality

ComparativeMeasurement

$$metricMeasurement : \mathbb{F}\ MetricMeasurement_1$$

8.3.3 Repository

A MidArch Repository wraps the ability to determine and store evaluations around a MidArch Style Taxonomy. The operation *addEvaluation* is used to add evaluations to the repository once the differences have been determined using a *ComparativeEvaluator* (see below).

Repository

$$evaluator : ComparativeEvaluator$$
$$t : Taxonomy$$
$$differencesStore : \mathbb{F}(Difference)$$
$$commonalitiesStore : \mathbb{F}(Commonality)$$

addEvaluation

$$\Delta(differencesStore, commonalitiesStore)$$
$$first?, second? : MeasurableConfiguration$$
$$differences? : \mathbb{F}\ MetricMeasurement_2$$
$$commonalities? : \mathbb{F}\ MetricMeasurement_1$$

$$\begin{lambda}
\textbf{let}\ firstStyle == first?.declaresStyle;\\
\quad secondStyle == second?.declaresStyle \bullet\\
\quad (\exists\, d : Difference \mid d.referencedStyles = \{firstStyle, secondStyle\} \land\\
\qquad d.metricMeasurement = differences?\\
\qquad\quad \bullet\ differencesStore' = differencesStore \cup \{d\}) \land\\
\quad (\exists\, c : Commonality \mid c.referencedStyles = \{firstStyle, secondStyle\} \land\\
\qquad c.metricMeasurement = commonalities?\\
\qquad\quad \bullet\ commonalitiesStore' = commonalitiesStore \cup \{c\})
\end{lambda}$$

$$performAndAddEvaluation \mathrel{\widehat=} evaluator.evaluate \parallel addEvaluation$$

differencesStore and *commonalitiesStore* contain only direct measurements that correspond to styles for which candidate architectures have been modelled. When defined this way, it cannot be identified which differences and commonalities stem from the same evaluation. Also, for any differences and commonalities, the reference to the original configurations that have been evaluated is abstracted from. The MidArch Repository Tool must additionally store these references.

The *ComparativeEvaluator* specifies the properties of the differences and commonalities of evaluation results. It is implemented by the MidArch Repository Tool (Section 11.1).

ComparativeEvaluator

$$isDifferentFrom : Measurement \leftrightarrow Measurement$$
$$isEquivalentTo : Measurement \leftrightarrow Measurement$$

$$\forall a, b : Measurement \bullet$$
$$\neg (a \ \underline{isDifferentFrom} \ b \land a \ \underline{isEquivalentTo} \ b)$$
$$\forall a, b : Measurement \mid a \ \underline{isDifferentFrom} \ b \lor a \ \underline{isEquivalentTo} \ b \bullet$$
$$\exists m : Metric \bullet \{a, b\} \subset scale(m)$$

evaluate

$$qm? : QualityModel$$
$$first?, second? : \downarrow MeasurableEntity$$
$$differences! : \mathbb{F}\ MetricMeasurement_2$$
$$commonalities! : \mathbb{F}\ MetricMeasurement_1$$

$$\forall metric : Metric;\ m_1, m_2 : Measurement \mid$$
$$(metric, m_1) \in qm?.measure(first?)$$
$$\land (metric, m_2) \in qm?.measure(second?) \bullet$$
$$(metric, m_1, m_2) \in differences! \Leftrightarrow m_1 \ \underline{isDifferentFrom} \ m_2$$
$$\land (metric, m_1) \in commonalities! \Leftrightarrow m_1 \ \underline{isEquivalentTo} \ m_2$$

Derived Evaluations The basic *Repository* must be extended in order to include derived evaluations. The derived evaluations are associated with those MidArch Styles that have not directly been used to model candidate architectures, but are positioned higher within the hierarchy of a MidArch

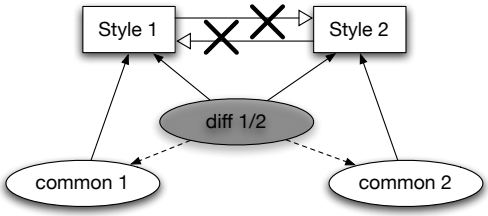

Figure 8.4: Derivation of Evaluation Differences

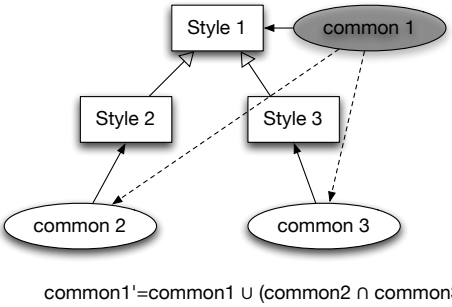

Figure 8.5: Derivation of Evaluation Commonalities

Style Taxonomy.

The evaluation results are raised to the style level by comparing the evaluation results of candidate architectures conforming to different MidArch Styles. They are separated into a part containing measurements that significantly differ between the architecture evaluations *(differences)*, and a part containing the identical or very similar measurements *(commonalities)*. These parts do not need to cover all evaluation results, i.e., metrics that are neither significantly different nor similar for all evaluated architectures may be dropped.

The MidArch Repository Tool implements the derivation process, which proceeds step by step towards the root of the MidArch Style Taxonomy. Each derivation step lifts the evaluation results one level upwards in the hierarchy. Figures 8.4 and 8.5 illustrate the two cases of the derivation process, where evaluation results are derived from evaluation results corresponding to two styles. The non-shaded ellipses refer to the evaluation results that are already present before a derivation step, while the shaded ellipses refer to

the evaluation results that are updated by the derivation step.

Figure 8.4 illustrates the derivation of differences. Common evaluations for each of two styles Style1 and Style2 are given, under the condition that Style1 and Style2 are mutually neither direct nor indirect superstyles of each other (if they are, the evaluation results are considered in different derivation steps). The differences between the evaluation results are derived by first forming pairs of evaluation fragments relating to each style, and then removing those pairs that contain equivalent results for both styles.

Figure 8.5 illustrates the derivation of commonalities. Common evaluations for each of two styles Style2 and Style3 are given, which possess a common direct or indirect superstyle Style1. The new commonalities of the superstyle are determined by forming the intersection of commonalities of each of the styles.

Limitations For simplicity, we assume here that the same quality model is used for all evaluations. This will not be the case in practise. We regard it as an implementation issue to deal with this problem by selecting a suitable intersection of quality models for each comparison. In fact, the existing implementation of the MidArch Repository Tool (Section 11.1) resolves this issue.

8.3.4 Selection of a Style from the Repository

Based on the evaluation of Task 3E (see Section A.3.3.5), the selection of a suitable style from a MidArch Style Taxonomy can be performed. These are distinguished into those obtained *directly* by architectural evaluations, and those that are derived *indirectly* from differences and commonalities of other evaluations.

The selection process then matches the requirements (depicted as rounded rectangles) stated in terms of the quality model against these evaluation results. The matching starts at the top of the MidArch Taxonomy and proceeds along the specialisations down to its leafs.

The algorithm shown in Figure 8.6 describes how the selection process works. The step "form clusters" can use different clustering algorithms, for example forming buckets of a fixed number of elements, or buckets with a fixed match value interval.

8.4 Examples

Essentially, two illustrative cases exist in applying the MidArch Design Method:

1. New MidArch Styles are specified,

```
SelectStyle(MidArchTaxonomy t, Requirements r)
        returns set of MidArchStyle
current ← {root node of t}
while ∃ s ∈ current : substyles(s in t) ≠ ∅ do
  next ← ∅
  for all s ∈ current do
    current ← substyles(s in t)
    form clusters of current by matchCommonalities(q, r, s, t)
    order current first by the cluster and second within each cluster
            using the partial order induced by
            matchDifferences(q, r, s1, s2, t)
    next ← next ∪ (first n styles from current)
  end for
  {optionally sort out styles below a certain commonality match threshold
  value}
end while
{now all styles in current are leaf nodes in the taxonomy t}
{propose current as candidates for modelling architectural candidates}
return current
```

Figure 8.6: Algorithm for Selecting a Style from a MidArch Repository

2. MidArch Styles to be used are selected from the MidArch Repository.

The first case is described in Section 8.4.1 with a focus on the style evaluation. Complete MidArch instances for this case are given through the two case studies presented in Chapter 10. Since the case studies do not cover the second case, we provide a detailed example for that in Section 8.4.2.

8.4.1 MidArch Instance Involving Style Evaluation

Figure 8.7 gives an overview of the pairwise comparative evaluation process of MidArch Styles. The figure clearly separates the entities that are on the style level (shown with a dark background) and those on the architecture level (shown with a white background).

The three elements at the top represent a fragment of a MidArch Taxonomy.

In the row below, the two candidate architectures resulting from two occurrences of Task 3C are shown, which *conform to* the respective MidArch Style. These architectures are evaluated, which yields the evaluation results in the row below the architectures. The evaluation results are still at the architecture level.

The differences and the commonalities for the evaluation results for the two candidate architectures for the current system are shown at the bottom of Figure 8.7.

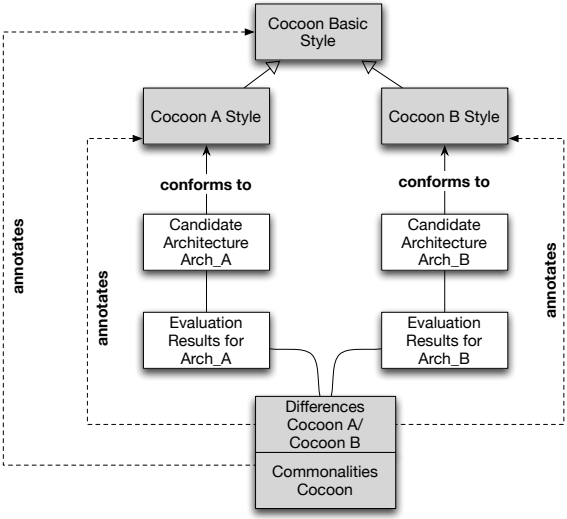

Figure 8.7: Comparative Evaluation Process of MidArch Styles

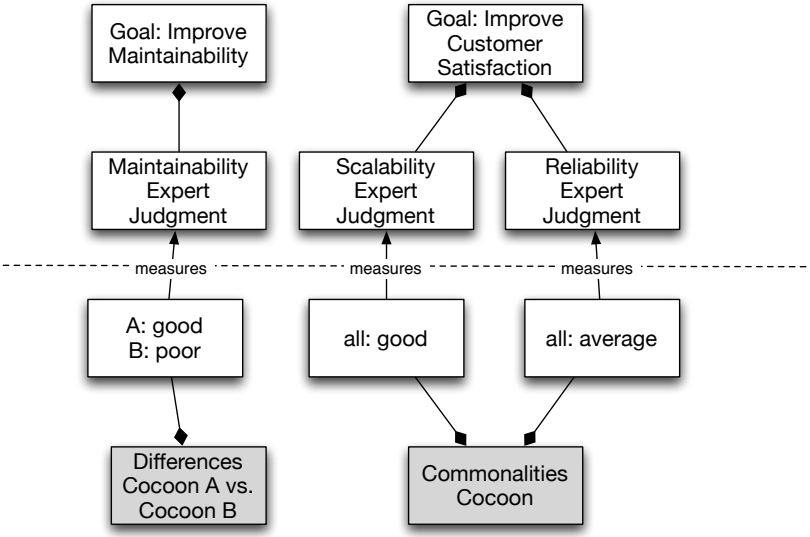

Figure 8.8: Comparative Result of the Evaluation of Two MidArch Styles

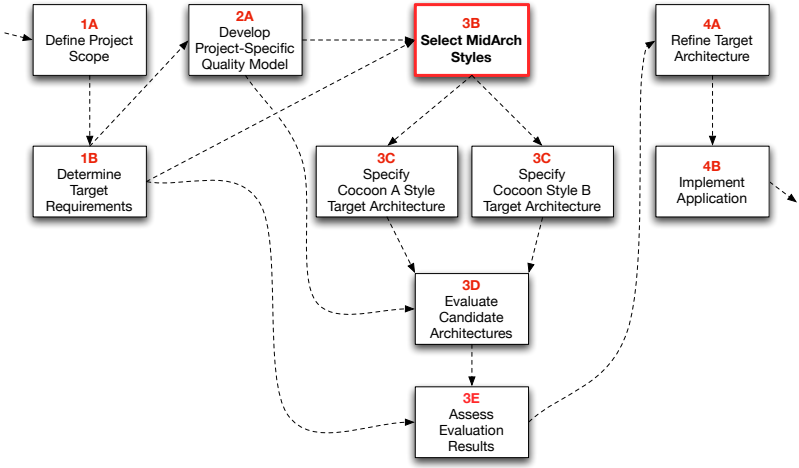

Figure 8.9: An Example Application of the MidArch Design Method

Figure 8.8 illustrates their contents. In the top of the figure, above the dashed line, the metrics level of the quality model resulting from Task 2A is shown. Below the dashed line, the measurements of the architectures for the given metrics are shown and associated to either the differences or the commonalities.

8.4.2 MidArch Instance Involving Style Selection

Figure 8.9 shows a simple example application of the MidArch Design Method, which also depicts the dependencies between the tasks. The dependency arrows have a completion (finish-to-finish) semantics, i.e., the target task cannot be completed before the source task is completed. However, for simplicity, we discuss the process as if the tasks were performed sequentially.

First, the project scope is defined in Task 1A, which is the development of a new web-based information system. In Task 1B concrete target requirements are identified. The goals covered by these target requirements are refined into a detailed quality model in Task 2A. An excerpt of the simple quality model for this example is later shown in the top part of Figure 8.8. In a real application of the MidArch Design Method, a full goal/question/metric quality model should be used.

In general, a quality model in the MidArch Design Method may exhibit a wide range of metrics, ranging from expert judgements to formal prediction techniques. The degree of detail of the architecture models significantly

impacts the accuracy of the results but also the cost of modelling and evaluation. Our research does not provide an original contribution to architecture evaluation techniques. Therefore, we use the following example in order to keep the evaluation simple: the quality characteristics maintainability, scalability and availability are each judged by an expert on a simple ordinal 5-level scale. The experts are free in their judgement, so the result is obviously subjective, while it can be made more reproducible by using some guideline for the judgement. In addition, intersubjectivity can be achieved by including multiple experts in the judgement process.

In Task 3B, the target requirements—stated in terms of the quality model—are matched against the evaluation results for the styles contained in the MidArch Repository. We assume that several styles have already been added to the MidArch Repository and evaluation results for these styles have been obtained in previous MidArch instances.

The matching procedure yields two candidate styles termed Cocoon A and Cocoon B. Both of these styles are based on the Cocoon Basic Pipeline Style, but they differ in the implementation of database accesses: Cocoon A requires all database accesses to be encapsulated within one filter of the pipeline, while Cocoon B does not. More details on this task are discussed in Section 8.4.2.1. For both of these styles, candidate architectures are modelled in the two occurrences of Task 3C.

Task 3D then applies an architecture evaluation method to both candidate architectures. The results are compared and added to the MidArch Repository. In Task 3E, an overall assessment of the evaluation results is done, which evaluates the results against the requirements and ranks the candidate architectures.

In Task 4A, the final target architecture is determined and a mapping to the implementation level artefacts is defined. Finally, the implementation is carried out in Task 4B.

8.4.2.1 Style Selection Task

Figure 8.10 shows a MidArch Style Taxonomy, which contains the Cocoon styles (depicted as rectangles) that will finally be selected, and in addition the abstract MidArch Styles Blackboard and Pipe-and-Filter. Evaluation results that are annotated to the styles are shown (depicted as ellipses).

In the example, the selection process proceeds as follows (the step numbers refer to the small ellipses in the figure):

1. The requirements are matched against the difference set of the abstract pipe-and-filter style and the abstract blackboard style. The result is that the pipe-and-filter style better achieves the requirements.
2. The only substyle of the pipe-and-filter style is the Cocoon Basic Style, which matches the requirements. Since this is already a concrete style,

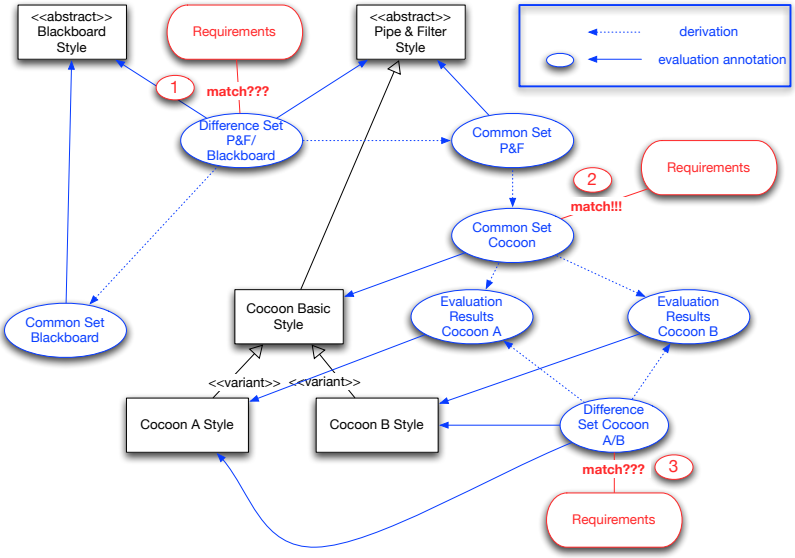

Figure 8.10: Incremental Style Selection from a MidArch Style Taxonomy

the matching could stop here. However, we assume that the variants are also considered.

3. Thus, the matching process continues with the difference set of the Cocoon A & B variants. However, the result does not clearly suggest one or the other of the variants, so both are used to construct candidate architectures for subsequent evaluations in Task 3D (see Figure 8.9).

8.5 Summary

In this chapter, we have introduced the MidArch Design Method that enables the early, architecture-level selection of middleware platforms. The basis for the selection support is a design knowledge base that is built around a taxonomy of MidArch Styles (introduced in Chapter 7).

In Section 8.1, we introduced the assumptions underlying the MidArch Design Method, which allow the expectation that the application of the MidArch Design Method is beneficial. These include the widely accepted assumptions that software quality requirements can be stated on the architectural level, and an architectural description can be evaluated with respect to the quality characteristics these requirements refer to. Furthermore, we assume that the choice of middleware platform influences topological constraints in the logical component structure architectural view, which are captured in MidArch Styles, and these variations in the constraints influence the software quality characteristics. All in all, these assumptions allow us to make the proposition that it makes sense to associate knowledge on the quality characteristics of architectures with the MidArch Styles the architectures conform to.

The activities of the MidArch Method are defined in Section 8.2. The high-level activities comprise the definition, preparation, exploration, and implementation activity. The central activity, which also forms the major share of the effort of a MidArch instance, is the exploration activity. The tasks of this activity include the specification of MidArch Styles (Task 3A), the selecting of MidArch Style candidates (Task 3B), the specification of style-based candidate architectures (Task 3C), the evaluation of individual candidate architectures (Task 3D), and the assessment of evaluation results for making a final style selection decision (Task 3E). The details of the tasks are defined in Appendix A.

Core aspects of the MidArch Repository that is supposed to store the knowledge needed to support these tasks, have been formally specified in Section 8.3.

Section 8.4 provided two examples of MidArch instances, one that centres around the specification of MidArch Styles and the acquisition of design knowledge, and one that centres around the selection of MidArch Styles and the (re)use of that design knowledge.

Part III

Evaluation

9 Overview

The MidArch Design Method, which has been proposed in Chapter 8, must be evaluated in the light of the state of scientific knowledge and industrial practise. There is an inevitable gap between the abstract, decontextualised description of the method and its concrete, recontextualised instances [Floyd, 1997], i.e. empirical evaluations referring to instances of the method have limitations in validity with respect to the method as such. Evaluating software design methods from a rationalistic point of view has also been challenged [Robinson et al., 1998].

To achieve an evaluation that is as complete as possible, both views of the method are evaluated. First, the decontextualised form is discussed in Section 9.1, which refers to the *rationale* of the method. That is, a line of reasoning towards the viability of the approach is made from a rationalistic point of view.

Then, the empirical validation of the method is discussed. Within this thesis, only a limited empirical validation has been done through two case studies. The qualitative nature of the case studies is more suited to providing insight into potentials for improving the MidArch Design Method. If this thesis were part of a comprehensive research program, it would make sense to first perform some iterations performing case studies and adaptations of the MidArch Design Method based on their results, before performing costly experiments with the MidArch Design Method. To complement these MidArch-specific research activities, general experiments could be made that are able to provide more fundamental insights into the nature of software engineering, which could contribute to a general theory of software development. However, since this thesis is an isolated effort we can only perform initial case studies. The general goal of these case studies and the limitations of this validation method are discussed in Section 9.2. Opportunities for an in-depth validation of the method, given more resources, are discussed in Section 9.3. The discussion refers to the research questions that have been posed in Chapter 4.

9.1 Rationale

A validation of the MidArch Design Method is made from a theoretical, rationalistic viewpoint. The MidArch Design Method builds upon experiences made in prior research in middleware modelling, architectural styles and architecture evaluation methods. With the style usages taxonomy in

Chapter 6, we have systematically established a framework for defining new design methods using architectural styles. In particular, in the work presented in this thesis, we have used the framework for designing a software design method combining a platform-oriented and customisation-oriented approach (cf. Sections 6.3.2 and 6.3.3). A detailed discussion of the MidArch Design Method in relation to prior work is made in Section 12.

The discussion of the rationale primarily contributes to the answer of research question Q4.1:

> "How can architecture-level software design be supported by a systematic method exploiting a MidArch Style Taxonomy for selecting an appropriate middleware platform?"

The discussion justifies the choices taken in designing the MidArch Design Method by referring to experiences from prior work and extending on the line of justification of these methods.

We can also establish some hints towards the quantitative reasoning that could in theory be used to measure the quality of the MidArch Design Method in comparison to other design methods, and argue why the measurements of the MidArch Design Method are probably better than those of the alternatives.

The basic question for evaluating this aspect is: "How good (with respect to a quality model) is the architecture produced by the design method compared to the best architecture?" However, there are at least two problems with this question. First, design processes are never deterministic (cf. Section 5.6). So, instead of the quality of a single architecture the probability distribution of the quality evaluations for all architectures that may be produced by a design method needs to be considered. Second, the best architecture is usually unknown. As discussed in Section 9.3, it is already costly to obtain any architecture for comparison.

As a consequence, we need to make some additional assumptions for arguing that the MidArch Design Method is an improvement over the state of the practise. These will be elaborated upon in the following. We assume that:

1. Any architecture design method at least implicitly chooses an architectural style.

2. Making the design decisions more objective improves the quality of the architecture.

3. Style conformance is an important quality of the architecture (and does not negatively affect other quality properties).

4. The more specific a style is, the better is the quality of the architecture.

To each of the second to fourth assumption, a corresponding metric can be defined such that an improvement of the metric can be considered to be an indicator for an improvement of the design process.

These metrics are:

1. The *number of architectural styles proposed by the design method*, independent from the judgement of the architect. This measures the degree of support the architect receives from the method. Alternatively, it measures the degree to which the subjectivity is reduced. The larger the metric is, the greater is the influence of the individual experience of the architect on the outcome of the design process.

2. The degree of conformity of the architecture to the style. To measure this in the case the style has only been implicitly chosen, an explicit style description must be used in spite of that. Then, the *number of violations of the style's constraints* of an architectural description can be counted. Obviously, the more restrictive the style is, the easier it is to violate the constraints. Thus, the specificity of the style must be considered as well.

3. The specificity of the style can be easily measured by the *number of declared constraints*. This is an indirect measure, since the number of constraints does not allow any implication regarding their restrictiveness. In theory, the number (resp. the order of magnitude) of conforming architectures were a metric for that. However, it cannot be effectively measured. A third metric were the complexity of architecture-to-implementation mappings, which draws from the fact that the more specific a style is, the closer the architectures it allows are to the implementation level.

The MidArch Design Method influences these metrics as follows:

1. Initially, when the MidArch Repository does not contain any evaluations, the MidArch Design Method does not restrict the number of styles to consider and it does not provide any guidance to the architect in selecting a MidArch Style to model and use. Thus, the first metric is the same as for any other design method. When the MidArch Design Method is repeatedly applied and the design knowledge in the MidArch Repository grows, the proposals of the MidArch Design Method get more and more refined.

2. Since the MidArch Design Method uses an explicit architectural style description as the basis for modelling the architecture, the conformance of the architecture to the style is intrinsically reached through the design process. This is not usually done in architectural development.

3. When architectural styles are considered in other architectural development methods, usually only general-purpose styles are considered. The MidArch Styles that are considered in the MidArch Method are closer to the implementation level through their middleware-oriented character.

9.2 Case Studies

The purpose of the case studies is to evaluate the applicability of the MidArch Design Method in a setting with a real-world software system. The case studies are empirical and gain primarily qualitative data. The results provide no substantial basis for generalisation, but can help in stating hypotheses claiming general applicability (see Section 9.3).

The case studies primarily contribute to answering research question Q4.2:

> "Is the proposed method applicable?"

A summary of the two case studies that have been conducted in the course of this thesis is found in Chapter 10.

9.3 Experimental Validation

The validation approaches described in Sections 9.1 and 9.2 do not suffice to answer research question Q4.3:

> "How does the proposed method compare with existing software engineering practises in terms of productivity and predictability?"

An experimental setup is required to obtain the quantitative data that can answer this research question. Two approaches can be thought of in setting up an experiment: Either the MidArch Design Method is compared to specific other design methods, or it is compared to an ad-hoc development process. Both approaches pose different major obstacles for conducting this type of research. Common to both approaches is the high cost of obtaining data that is sufficiently substantiated to allow making statistical propositions with acceptable confidence levels.

Obstacles of comparing the MidArch Design Method with specific other design methods The major problem in comparing the MidArch Design Method to specific other design methods is that no other methods that fit the coverage of the MidArch Design Methods activities are readily available. In comparing the MidArch Design Method, the influence of different properties could be investigated: The ideal comparison candidate were a method that has the same activities, but differs only in its non-usage of architectural styles. Despite its non-availability, such a method could be constructed, but

there are many design choices that could be made in the method's design. Since this artificially constructed method would not have been used outside this research project, translating the results to the real world would be very difficult.

Other aspects that could be investigated are the benefit of performing an explicit design space exploration, or the benefit of making design decisions explicit. However, these questions are not directly related to the MidArch Design Method, but are more general questions about software engineering. If validated knowledge on these issue were in fact available within a comprehensive software engineering theory, this theory could be used to design methods whose benefit could be evaluated a priori in the light of the theory. Then, only the general theory would need by validated empirically. It would not be necessary to validate these general propositions for every individual design method based on the theory.

In general, suitable domains for any design method have not been systematically determined through empirical research. In fact, even the structure of the total space of these domains is still unknown.

What could be done relatively easily is the comparison of the MidArch Design Method to other middleware selection methods. One would need to consider that other middleware selection methods do not produce a target architecture along with the selection decision, but this is an obstacle that can be dealt with. The drawback is, that the conclusions that can be drawn from such a comparison are very limited. The overall productivity gain (or loss) of using a middleware selection method could only be considered within a complete development project, which would bring the endeavour back to the situation described above.

Obstacles of comparing the MidArch Design Method with ad-hoc development Comparing the MidArch Design Method to ad-hoc development processes would be more faithful to actual development practises, since most smaller development teams do not follow strict development methods. Even if a more rigorously defined method (such as the V-Modell [KBSt, 2008]) is used, these methods do not specifically address the problem of middleware selection. Thus, middleware selection is usually done ad-hoc in practise.

In evaluating the MidArch Design Method against ad-hoc development processes, the following questions may be considered:

- How much does the MidArch Design Method increase or decrease costs compared to the ad-hoc process?
- How much does the MidArch Design Method reduce or increase the risk of making a middleware selection which needs to be revised later?
- How much does the MidArch Design Method reduce or increase the risk of designing an architecture that cannot be easily implemented on

the selected middleware?

- How does the use of the MidArch Design Method influence the time at which the middleware selection process is initiated and completed?
- Under which circumstances and preferences of the actors do the effects on cost and benefit justify the use of the MidArch Design Method?

The most interesting questions in this list refer to risks, which refer to cost and benefit measurements only indirectly, and require statistical propositions over a larger number of projects.

High cost of conducting experiments The problem of the high costs of conducting software engineering experiments is not specific to this research project. Costs are often reduced by falling back to using students as experiment participants rather than experienced practitioners, who are not reimbursed monetarily but rather rewarded with credit points. This solution is also used in other disciplines, such as psychology: Large portions of validated psychological theories base primarily on experiments with college students. This approach always poses a threat to the validity of the results for a different population. In the case of psychological experiments, usually the population that propositions claim to cover is the general population. The assumption is made that the phenomenons analysed are found within the student population with variances comparable to the general population (the proper selection of students from the student population is another issue, however more a technicality). While basic psychological properties may not depend much on personal experience, performance in complex specialised activities is very likely to do, which increases the threat to validity for experiments involving complex design methods.

Since the MidArch Design Method is not generally known (neither within the software engineering student population nor in the general population of software engineering practitioners), a learning phase for the participants must be part of the experiment, which adulterates the results, since a newly learnt method is compared with another method, which may or may not be well-known to the individuals. It would be even more costly to compare a realistic setting, i.e. equally well-known and practised methods.

Conclusion We may conclude that performing experiments to validate the MidArch Design Method suffer not only from the high costs, but also from the lack of a sufficient software engineering theory.

10 Case Studies

In this chapter, we report on two case studies that applied and partially evaluated the MidArch Design Method. Both case studies were conducted together with students in the context of student theses. Also, they were both applied to real software systems from the actual work of industrial partners. In Section 10.1 the case study conducted together with Johannes Bornhold [Bornhold, 2006] and in cooperation with regio GmbH, Oldenburg is reported upon (RegIS Online Case Study). In Section 10.2, we describe the case study conducted together with Angela Eiben [Eiben, 2008] and in cooperation with Thales Defence Deutschland GmbH, Wilhelmshaven (Thales Case Study).

The documentation of both case studies is structured in the same way: First, the *context* of the case study, i.e. the subject system and its environment are described. Then, the *setup* of the case study is described, that is the goals from both the practical (i.e. that of the developer or the customer) viewpoint as well as the scientific viewpoint. Then, the *instantiation* of the MidArch Design Method is described and the progress of its activities is documented. A *discussion* of each case study completes the description.

Conclusions covering both the results of both case studies are drawn in Section 10.3.

10.1 RegIS Online Case Study

10.1.1 Context

The subject system of this case study is called RegIS Online, which has a public web interface at `http://www.regis-online.de/`. It is developed by regio GmbH, Oldenburg, an institute associated with the University of Oldenburg, and is provided on behalf of the business development authorities of the municipalities around Oldenburg.

The trade information system is provided as a supporting tool for sustainable regional development. The general idea of this system is to make information on the economic potential of a region available to companies to increase regional business collaboration.

RegIS Online consists of three subsystems which have been developed rather independently in the past. Each of these subsystems currently provides a distinct user interface. Two of these interfaces are already web-based. There are three roles of users accessing these interfaces. The relationships

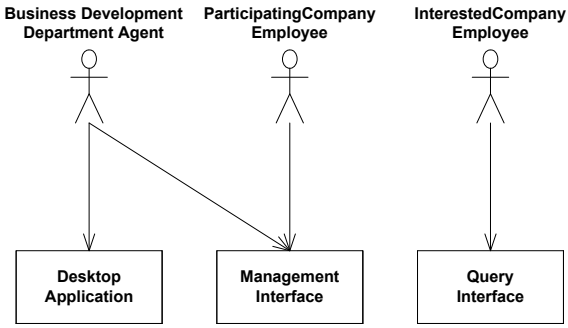

Figure 10.1: User roles and their relationships to RegIS Online interfaces

of user roles and interfaces are shown shown in Figure 10.1. The business development agencies of the counties and municipalities in the covered region collectively form the current customer, to which our cooperation partner provides the service.

First, an access-controlled web interface is used to collect and manage the data about the participating regional companies. It has two main groups of users. The first group represents the agents at the business development agencies. These users can administrate the data of their district's companies and manage the users of the second group, which represent the participating companies themselves.

Second, a web-based query interface is publicly available. It presents information about companies which are located in the covered region. The ability to query this information by different filter criteria facilitates finding potential collaboration partners among regional companies, and thereby supports building regional business networks. The data about each company consists of statistical and address data as well as information on offered services, special skills and cooperation interests. In addition, the business development agents have the ability to export their data in a spreadsheet file format.

The third user interface is provided by a desktop application which originates from a point in time before the development of the other subsystems. Only part of the functionality offered via this interface is still in use. It allows to manage private additions to the data records, which are used only internally by the business development agencies. Currently, the new management subsystem provides an export facility which allows the users of the desktop application to manually download the up to date data in the desktop application's proprietary file format and afterwards import it to update their locally stored data.

The desktop application was originally also used to manage the data,

which is now managed through the web-based user interface. Originally, the data was sent by the business development agencies to the service provider by email and the service provider manually combined the data fragments to feed the query subsystem.

10.1.2 Setup

10.1.2.1 Business Goals

From the point of view of the supplier, there are three main migration goals: First, the system shall be made ready for use by multiple customers. Second, it shall be made more evolvable. Third, the availability of the system should be improved.

The first goal must be seen in the context that this system was originally developed to be used in a single instance for a single region and therefore no effort was made to support multi-customer capabilities and customer-specific customisation needs. In the future, this system shall be offered to multiple customers (i.e., other regions). This means on the one hand that a greater effort must be put on support the adaptability to special customer needs with a manageable amount of human resources. On the other hand, special care must be taken in the product development process to either support hosting of multiple instances and a (semi)automated update-mechanism to new releases of the product, or to add multi-customer capabilities to a single instance of the system.

The second major goal is to increase the system's evolvability. The system itself and its parts evolved over time. Adoption to new requirements has become a challenging task which requires involved developers to be familiar with many parts of the current system. Evolvability is enabled at the architectural level, which must be adequately reflected in the system's implementation.

The third goal is to increase the availability of the system. Availability becomes more important and business-critical as more customers are using the system.

The second goal is related to the supplier's cost of development, while the third goal is related to the benefit the system offers to its users. The first goal combines both aspects. These two aspects form the basis for defining the detailed quality model within the case study.

10.1.2.2 Scientific Goals

The case study was the first application of the MidArch, so naturally no MidArch styles existed before, and the selection of MidArch styles was not part of the case study. In the beginning, it was planned that a prototypical implementation of one candidate architecture should be made as part of the case study, but this was dropped during the course of the case study due to

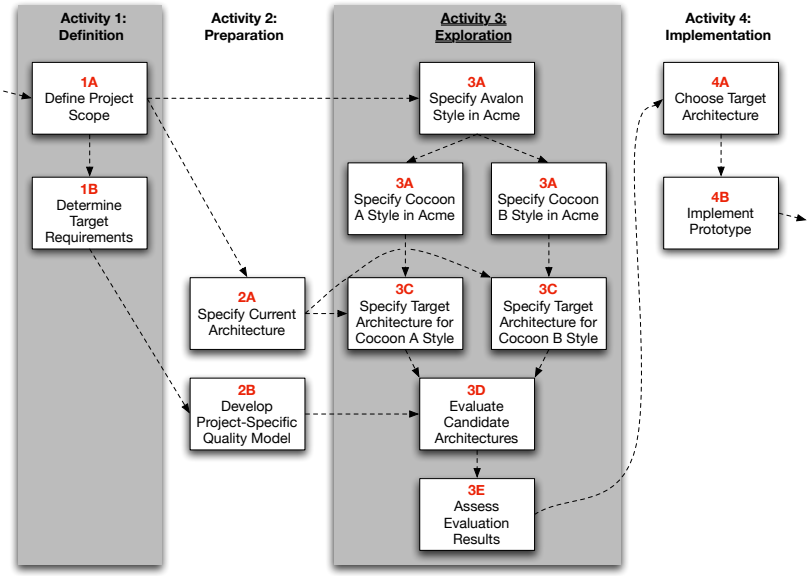

Figure 10.2: Instance of the MidArch Design Method in the RegIS Online case study

the effort necessary for the other activities. The main goal of this case study was to evaluate the feasibility of the overall method when applied to a real world project, and to gain some insights into the relative effort necessary for the tasks of the method. In addition, the suitability of Acme, xADL and ArchiMate for modelling architectural styles and style-based architectures should be evaluated, including the existing tool support.

10.1.3 Instantiation

In this section we describe the instantiation of the MidArch Design Method in the case study. Figure 10.2 shows the activities, tasks and their dependencies, which also determined their chronology. This section is structured according to the activities and tasks of the MidArch Design Method (see Section A.3).

Activity 1: Scoping and Goal Definition

Task 1A: Define Scope In the case study, we selected the three external interfaces *QueryInterface*, *ManagementInterface* and *DesktopApplication* (and the subsystems they depend on) as the subject

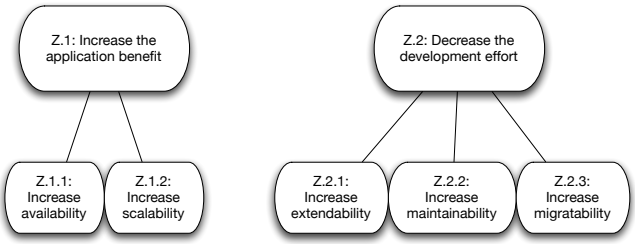

Figure 10.3: Goal level of the GQM quality model for the RegIS Online case study

system.

Task 1B: Determine Target Requirements The requirements were elaborated on the basis of different internal documents of the supplier. These documents contain information on the long-term vision for the software system, non-functional requirements and use cases.

Activity 2: Preparation

Task 2A: Model Current Architecture Originally, we planned to use ArchiMate and xADL to model both the current and the candidate target architectures in the case study. While ArchiMate was indeed used for some high-level modelling (see Figure 10.4), xADL proved not to be usable due to its lack of support for style-based modelling. Thus, Acme was used to model the current architecture as well as styles and style-based architectures, as described in Section 7.7.1). In addition, informal graphical representations were used to illustrate some aspects of the architectures.

Figure 10.4 shows an excerpt from an ArchiMate model of the current architecture, which shows the parts necessary for the registration of new users. When a new company's employee requests a login to manage the data about his company, he is in the role *Company* and uses the *RegistrationService* to request his new login. This service is implemented on the business layer by the business process *Registration* which depends on the *PostOffice-Service*. This service is implemented on the application layer by the *PostOffice* and responsible for informing the right person in the *BusinessDevelopmentDepartment* role (*BDD*) to *Check* and possibly *Accept* this request. In the bottom part, this figure shows that the *PostOffice* component needs an available *EmailService* which, in the current case, is implemented by a *MailServer* on the technology layer which is installed on some device that in not

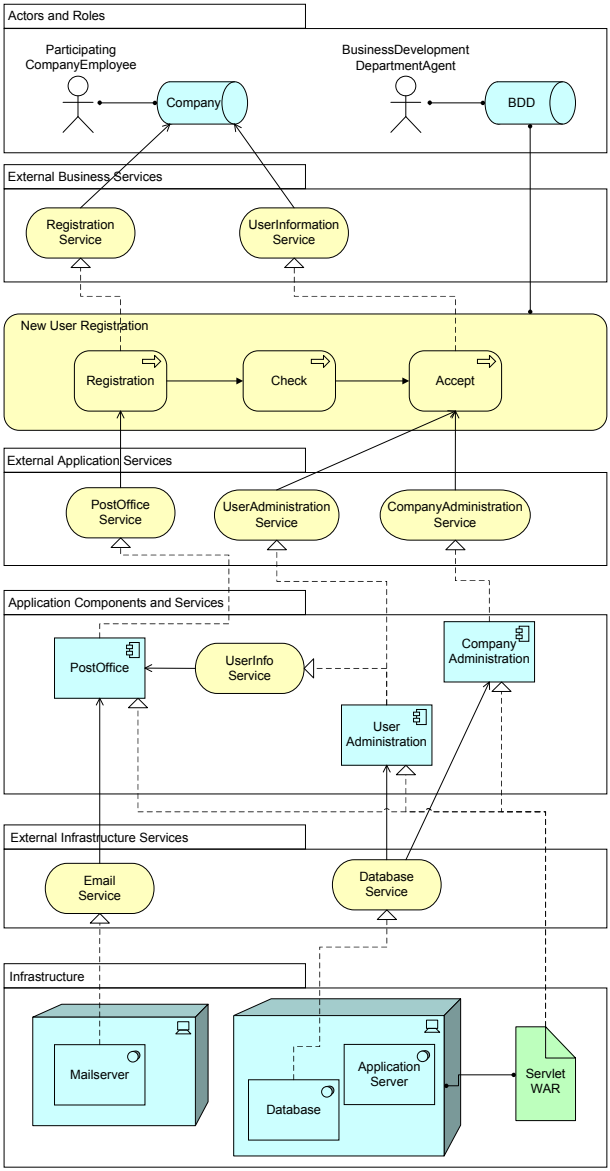

Figure 10.4: Partial ArchiMate Model of the Case Study System

further specified.

For more detailed modelling of the application-level architecture, we focused on the subsystems of the regional trade information system which already provide a web-based interface, i.e. the query and management interfaces. Both subsystems are based on the Apache Cocoon application framework but use the technology in different ways. With regard to the regional trade information system the most important Cocoon extensions are a framework for form handling and extended control flow support (CForms).

The query interface of the regional trade information system does not use special extensions of Cocoon. Most of its functionality is embedded in XSP documents (a Cocoon-specific language similar to Java Server Pages) which allow Java code to be embedded into XML documents. From these XSP documents, direct queries to the underlying database are made and the results are written into an XML representation of the query result which is then further processed by the following filters. These filters transform the query result into an appropriate HTML representation.

From a very abstract point of view, the management interface works in a similar fashion. The first filters perform some operations on the data and the following filters transform the result into a HTML representation. Differences appear with a closer look to the first part of the pipeline. The management interface makes use of Cocoon's CForms framework, which allows for better handling of form data and constraints, and of Cocoon's control flow framework, which allows to send forms with a blocking function call and to formulate control flows in an explicit, closed form. This framework is implemented through a JavaScript API which provides access to underlying Java objects. The second difference in comparison to the query interface is the way data is accessed. In the management interface, all data queries and manipulations are performed with Java objects which map to the underlying data storage (object-relational mapping using Hibernate [JBoss Labs, 2006]).

There are several problems with the current architecture with respect to the migration goals. First, especially the query interface is closely coupled to its underlying database, which is one of the reasons why it still uses a database of its own with an old schema. Because of this, it is technically hard to adopt new requirements that have an impact the database schema, and the effort is difficult to estimate.

Second, the mechanism which transforms the intermediate results to a HTML representation has been identified as another difficulty in practise. An own proprietary language has been developed for

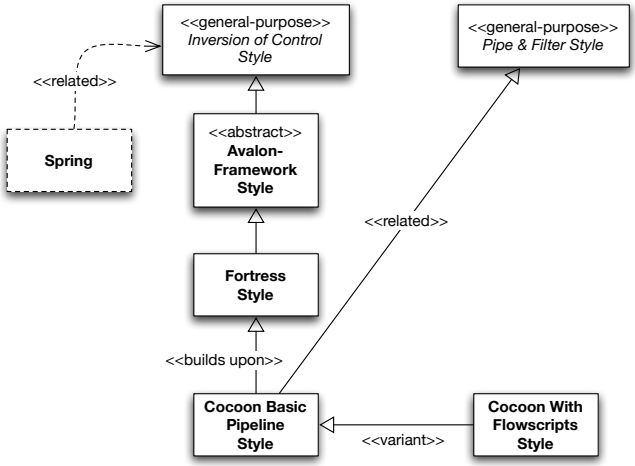

Figure 10.5: MidArch Style Taxonomy from the RegIS Online case study

the intermediate results which has grown over time and is nearly unmaintainable now.

Third, there are many tight couplings within the Java implementation of the data model, so that requirement changes often result in changes at many different places of the implementation which makes it harder to parallelise development tasks. For this reason, it is not easy to isolate the data tier from the presentation tier in the query and management subsystems.

Task 2B: Develop Project-Specific Quality Model We used an approach based on the GQM (goal/question/metric) approach [van Solingen and Berghout, 1999] to create the quality model. The goal level is displayed in Figure 10.3, where we distinguished goals and sub-goals. For more details on the quality model, see Bornhold [2006, ch. 4]. The metrics within the quality model were mainly expert evaluation metrics, where the expert in this case was the student, who already had much experience with the subject system due to his prior work. However, the quality model itself was reviewed by the other staff involved in the development of RegIS Online at regio GmbH.

Activity 3: Architecture Exploration

Task 3A: Model MidArch Styles Since the case study was the first application of the MidArch Design Method, no MidArch style

description existed at the beginning of the case study. Two concrete Cocoon MidArch Styles have been modelled within the case study: The basic Cocoon styles `Cocoon` and the Cocoon style with Flowscripts `CocoonFlow`. In addition, some auxiliary styles have been modelled, the concrete `Avalon` MidArch Style, and the abstract `ContainerIoC` MidArch Style, which represents component containers that apply the inversion of control principle for component execution. Furthermore, one Spring MidArch Style was defined in the case study. The resulting MidArch Taxonomy is shown in Figure 10.5.

Task 3C: Model Candidate Architecture Coarse architectural candidates have been modelled for both the (basic) Cocoon style and the Spring style. Due to problems with the modelling tools and time constraints, the architectures were not very detailed.

Task 3D: Evaluate Candidate Architecture The candidate architectures were evaluated according to the GQM quality model by the student.

Task 3E: Assess Evaluation Results The assessment of the evaluation results for the candidate architectures resulted in a slight advantage of the Spring-based architecture.

Activity 4: Architecture Selection and Adoption The case study did not cover the fourth activity since the actual adoption was left to the operation of Regio GmbH.

10.1.4 Discussion

We discuss the experiences gained in the case study according to the scientific goals defined in Section 10.1.2.2. First, we discuss the experiences regarding the feasibility of the MidArch Design Method and the relative effort of its tasks. Then, we discuss the experience with the specific ADLs used in the case study and the tools that are available for these languages.

10.1.4.1 Feasibility and Relative Effort

The relative effort of the task instances in the case study is illustrated in Table 10.1. The tasks of the core activity 3 as well as the high effort task 2B are discussed in more detail in the following.

Task 2B: Develop Project-specific Quality Model Task 2B took significantly more time than expected. This was due to the fact that we aimed at a highly customised quality model without referring to some predefined reference quality model. The customised quality model was elicited from various project documents and discussed and reviewed with project team

Task	Relative Effort
1A	–
1B	o
2A	o
2B	++
3A	++
3B	n.a.
3C	+
3D	o
3E	–
4A	n.a.
4B	n.a.

Table 10.1: Relative effort of task instances in the Regio Case Study

members. In a first step, a large number of goals (18 goals) was elicited. These goals were then prioritised and grouped to allow for a manageable number of goals on the one hand, and to filter out those goals that were expected to be affected by the software architecture. One example of an aspect that was of high importance to the supplier but was not expected to be affected by the component-and-connector viewpoint of software architecture was personal data protection. This step resulted in 2 top-level goals with a total of 5 sub-goals. Still, the resulting quality model contained numerous metrics (47 metrics).

For the MidArch Design Method to be applicable in practise, we hold it is necessary to reduce the elicitation effort. While we still see the necessity to customise the quality model to some extent, it is probably less costly to select appropriate parts from a given reference quality model rather than defining a project-specific quality model from scratch. This also has benefits for gathering design knowledge (cf. Section 11.1). However, an open question remains which reference quality model to select. Well-known quality models such as that of ISO 9216 appear not to be suitable since they do not focus on aspects that are affected by software architecture, but focus on properties of the final software system, and neglect genuine architectural qualities (cf. Section 2.3.2) at all.

We will get back to the problem of the effect of software architecture on the goals when discussing Task 3E.

Task 3A: Style Modelling Task 3A proved to be very costly as well, however for this task this was also expected. The high effort for style definition is acceptable, since it is not expected to be necessary in every instance of the MidArch Design Method. Moreover, a significant portion of the time can be attributed to learning and understanding the corresponding

middleware products, which would also be necessary when considering the use of the product in any other development approach. Some overhead is created through the need of formalising this knowledge in the style definition, but also the benefit of the reusable and commensurable style definition artefact is gained. In the case study, the student export reported that a better understanding of the middleware product was gained through the target-oriented analysis of the middleware products (Avalon, Cocoon, Spring) than through the typical learning through copy-and-paste from example applications, which often results in eclectic software architectures.

Task 3C: Style-based Architecture Modelling The effort of modelling the candidate architectures was also high, but this is the case for architecture modelling in general. The effort is assumed to be somewhat reduced by providing more specific modelling elements through the MidArch Styles, however this was not explicitly addressed in the case study and remains to be evaluated in future work (see Section 14). The Acme description of style includes such knowledge only implicitly, an explicit guideline for modelling architectures based on a particular style that amends the style description itself was believed to be helpful by the student.

Task 3D: Architecture Evaluation The architecture evaluation was performed through expert judgements in the case of the case study's quality model, where the student served as the expert. Not all of the metrics could be evaluated on the basis of the architectural descriptions, since the degree of detail was not high enough.

Task 3E: Evaluation Assessment Giving a recommendation based on the architectural evaluations from task 3D proved to be hard, since the results did not vary much between the architectural candidates. This was also attributed to the coarse level of detail of architecture modelling that was achieved during the case study.

Together with the experiences from Task 2B and 3D, it can be said that the design of a meaningful quality model is critical for the successful application of the MidArch Design Method. Two requirements for metrics to be used in quality model within the MidArch Design Method can be stated:

1. Each metric must be evaluable on the basis of a component-and-connector configuration only.

2. The measurements of the metric need to vary significantly across different MidArch styles resp. middleware platforms.

In this case study, no particular multi-objective evaluation or visualisation methods have been applied to the individual evaluation results.

10.1.4.2 Suitability of Modelling Languages

As noted above, xADL proved not to be usable due to its lack of support for style-based modelling. Thus, Acme was used to model the current architecture as well as styles and style-based architectures. Since our understanding of architectural styles was close to the ideas underlying the language design of Acme, no fundamental problems were encountered in this respect.

However, AcmeStudio (Version 3) proved to be quite unstable, which lead to unpredictable application crashes. In addition, it showed weird results when evaluating style constraints for architectural configurations. A fundamental problem with the Acme approach is that Acme is primarily a textual language. While AcmeStudio is able to visualise an Acme configuration, this visualisation is incomplete and editing a configuration always requires resorting to the textual representation at some point.

These aspects lead us to explore opportunities to use the UML for modelling styles and style-based architectures, since a variety of tools is available for the UML which were believed to be more stable, and since the UML is more genuinely a visual language (while some restrictions in this respect apply to the UML as well).

10.2 Thales Case Study

10.2.1 Context

The second case study was conducted in cooperation with Thales Defence Deutschland GmbH, Wilhelmshaven, who fits ships for the German and other navies. The subject system of this case study was the ASTT (Action Speed Tactical Trainer), which is a distributed simulation system used for education of navy officers. The ASTT represents a product family for which various customisations exist. Figure 10.6 exemplarily shows the architecture of the TVT (tactical procedure trainer) used by the German Navy. The system consists of both hardware and software components.

10.2.2 Setup

10.2.2.1 Business Goals

In its current implementation, the ASTT is a closed system which is not prepared to interoperate with third-party components. The German Navy, as an important customer of Thales, aims to perform advanced trainings where participants from different NATO countries take part in a shared simulation. Each participating country is assumed to use their own simulation system which form a simulation compound.

In addition, manufacturers of real components such as sensors and effectors should be able to supply simulation components for their components as well.

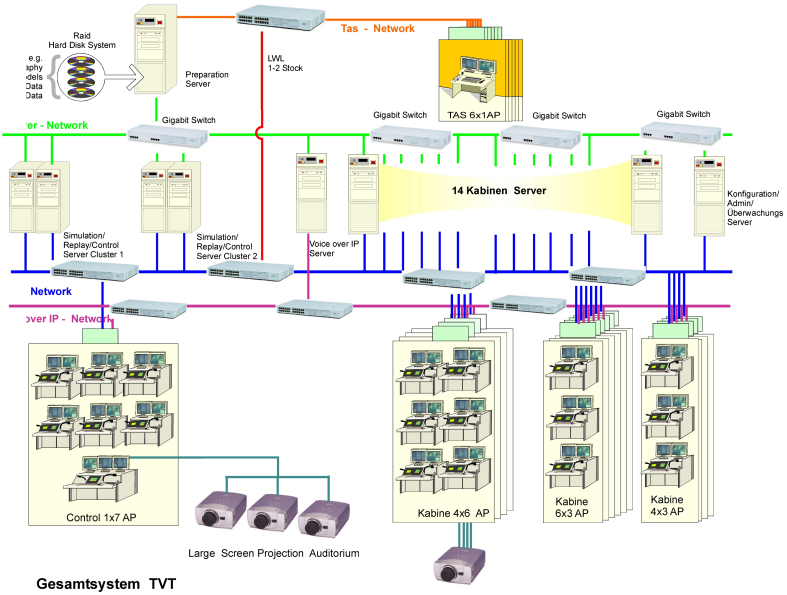

Figure 10.6: System Architecture of the TVT Implementation of the ASTT [from Eiben, 2008]

Since currently the whole simulation is performed by a monolithic simulation kernel, it is only possible to change predefined parameters of simulation components. An exchange of the whole simulation code which is necessary to simulate complex deviating behaviour is not possible.

The interoperability of the ASTT should thus be improved to facilitate such simulation compounds and manufacturer-supplied components.

10.2.2.2 Scientific Goals

The scientific goal of this second case study was to evaluate whether it confirms or rejects the results of the first case study. While the first case study was very exploratory due to the fact that the MidArch Design Method had never before been applied, the second case study could build on a somewhat firmer basis. The case study applied the UML/MidArch approach for the first time, however. Another goal was to evaluate the suitability of a commercial off-the-shelf tool (Rational Enterprise Architect) for modelling architectural styles and style-based architectures using the UML/MidArch approach was.

10.2.3 Instantiation

In this section we describe the instantiation of the MidArch Design Method in the case study. Figure 10.7 shows the activities, tasks and their dependencies, which also determined their chronology. This section is structured according to the activities and tasks of the MidArch Design Method (see Section A.3).

Activity 1: Scoping and Goal Definition

Task 1A: Define Scope The focus of the case study is on the software aspects of the system only. Since the system is very complex, only the APPLICATION_KERNEL subsystem was covered in the case study. The APPLICATION_KERNEL subsystem consists of several components within two groups:

Basic Services SYCO, COMM
Basic Applications HUMI, PACO, EASI, SCCO

The EASI (Environment and Simulation) and SCCO (Simulated Combat Control) components have been subjected to a more detailed modelling.

Task 1B: Determine Target Requirements The requirements were elaborated on the basis of different internal Thales documents. These documents contain information on the long-term vision for the software system, non-functional requirements and use cases.

Activity 2: Preparation

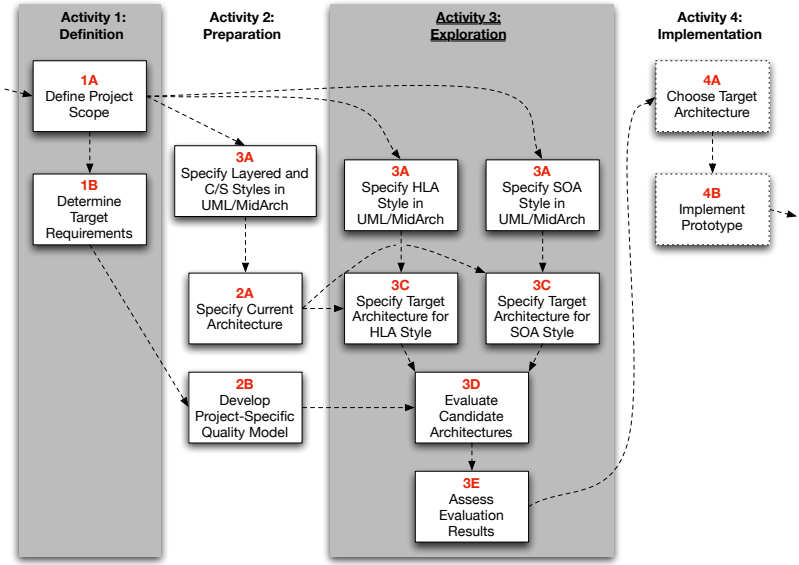

Figure 10.7: Instance of the MidArch Design Method in the Thales case study

Task 2A: Model Current Architecture Before the current architecture was modelled, two MidArch Styles were specified which were used to model the current architecture in a style-based fashion as well. The current architecture is an instance of the n-tier client-server style, which has been defined as a specialisation of the layered style.

Task 2B: Develop Project-Specific Quality Model The quality model was again developed using the GQM approach. As before, the metrics within the quality model were mainly expert evaluation metrics, where the expert in this case was the student, who already had much experience with the subject system due to his prior work.

Activity 3: Architecture Exploration

Task 3A: Model MidArch Styles MidArch Styles were modelled for the Service-oriented Architecture (SOA) as well as the High Level Architecture (HLA) Standard. The modelled styles are not related to each other, so the MidArch Taxonomy covering them is not very expressive.

Task 3C: Model Candidate Architecture Candidate architectures for three migration steps were modelled for each of the styles.

Task 3D: Evaluate Candidate Architecture The candidate architectures were evaluated according to the GQM quality model by the student.

Task 3E: Assess Evaluation Results The assessment of the evaluation results for the candidate architectures resulted in a clear advantage of the HLA-based architecture.

Activity 4: Architecture Selection and Adoption The case study did not cover the fourth activity.

10.2.4 Discussion

The case study resulted in four additional styles being modelled, most notably a SOA style and a HLA style. Modelling of architectural styles for SOA has been addressed by others [Baker and Dobson, 2005; Baresi et al., 2006; Budgen et al., 2004] but with a different focus and targeting other application domains. Comparing the resulting style descriptions remains future work. Modelling the HLA formally as an architectural style has been done by Allen [1996]. However, the effort was more focused on behavioural aspects and the result was too complex to be easily applicable in practise.

Originally, it was expected that the migration steps depended on the chosen style, but this did not turn out to be the case, so the same migration steps

were used for both styles. However, the (expected) cost of implementation the migration differs for the two styles.

The HLA style modelled in the case study is more concrete, more domain-specific and closer to the middleware layer, but also more complex than the SOA style. This corresponds reciprocally with the complexity of the architectural models themselves, where the HLA model is less complex than the SOA model. Learning the HLA style is thus more costly than learning the SOA style, but the knowledge can be reused for other HLA-based architectures.

While the HLA was evaluated clearly superior to the SOA in the case study, one aspect in favour of the SOA was not considered in the quality model: SOA technologies are much better known than the HLA and it is much easier to find SOA developers than HLA developers. Also, the market of SOA middleware implementations is much better developed than that of HLA implementations, which are primarily available as proprietary military implementations.

10.3 Conclusions

The two case studies were similar in their setup: In both case studies, a migration of an existing software system was targeted. New styles were modelled before the architecture modelling took place. The two case studies showed that the MidArch Design Method is indeed applicable to real software systems. The applicability was only shown in a lab setting, not in a real project, and nearly all roles were taken by the same person. However, the requirements underlying the lab studies were real. The case studies did not unveil contradictory results.

A reuse of design knowledge between these two MidArch instances was not possible, since the application and technical domains were too different. The implementation activity (Activity 4) as well as the style selection task (Task 3B) were not part of any of the case studies. Therefore, further case studies should be conducted to validate those aspects of the MidArch Design Method which have not yet been covered.

The second case study conducted with Thales also showed differences to the Regio case study besides the differences in the application and technical domains: It used the UML/MidArch approach rather than the Acme language for style modelling. More emphasis was laid upon modelling detailed architectural candidates involving multiple migration steps. The quality model was created in a more straightforward manner, without analysing design documents in depth and an extra review with system experts.

While assessment in the Regio case study did not result in a clear recommendation for either of the styles, the result was much clearer in the Thales case study, where the HLA-based architecture was superior to the

SOA-based architecture.

A problem that was encountered already in the Regio case study was persistent in the Thales case study, although the experiences of the Regio case were considered in conducting the second case study: The difficulty to define metrics for several quality characteristics such as performance and reliability that could be effectively applied on the architectural level by a domain expert. However, this problem is not specific to the MidArch Design Method, but is a general problem of software architecture research. Technically heavyweight methods that aim to predict system characteristics based on architectural descriptions have only a restricted applicability potential since they require very detailed architectural models. The problem becomes very prominent in the MidArch Design Method, however, since the evaluations of such metrics play an important role in an instance of the MidArch Design Method.

The two case studies also show a dilemma: On the one hand, it would be desirable to include viewpoints other than the logical component structure viewpoint into a style description, since the style modellers found that important aspects of the middleware platforms could not be reflected appropriately in that viewpoint. But, on the other hand, the style descriptions were already very complex for the logical component structure only, and including other viewpoints as well would result in unusable style descriptions. It is questionable whether the fact that advanced formal analyses on the architectural descriptions would be enabled by multi-viewpoint styles would outweigh the cost incurred by doing so.

11 Tool Support

In this chapter, we report on two tools that support the MidArch Design Method, which have been developed in two diploma theses. First, we present the central support environment MidArch Repository Tool in Section 11.1. Second, we present work on a tool for mapping MidArch architectural descriptions to the implementation level, which allows the definition of the basic mapping on the style level (Section 11.2).

11.1 Tool Support for the MidArch Design Method: MidArch Repository Tool

An initial implementation of the MidArch Repository Tool (Section 8.3) was done by Ulbts [2008] based on some preliminary work by Jung [2007].

Use Cases Figure 11.1 shows the use cases supported by the MidArch Repository Tool. The repository assumes that styles and candidate architectures have been modelled externally, and the resulting descriptions can be added to the repository. Currently, only Acme descriptions are supported, but an extension towards UML descriptions is possible with little effort. The most important task supported by the MidArch Repository Tool is querying for a suitable MidArch Style candidate, which is used in MidArch Task 3B. This task requires evaluation results from prior MidArch Instances, so the tool also supports adding architecture evaluation results, which is done in MidArch Task 3D or 3E. GQM quality models that have been externally created can be added to the repository. The repository assumes that the quality models have been created using the MidArch Support Environment [Jung, 2007].

Architecture The MidArch Repository Tool is implemented on the Eclipse Rich Client Platform. Figure 11.2 shows the plug-ins that make up the tool and their dependencies amongst each other and to external plug-ins. The tool amends the MidArch Support Environment, which is also implemented on the Eclipse Rich Client Platform. There is currently no integration between the two tools on the API level or user interface level. The integration is achieved through XML file system artefacts as shown in Figure 11.3.

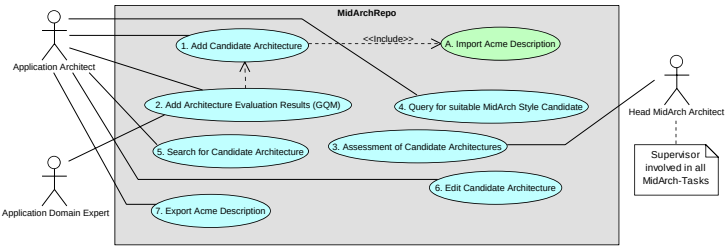

Figure 11.1: MidArch Repository Tool: Use Cases

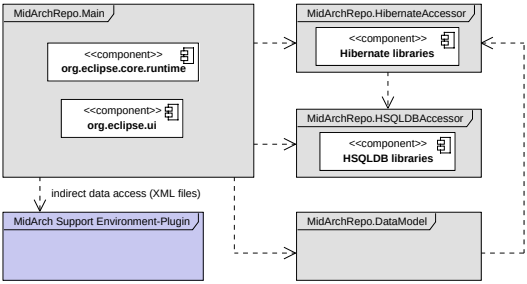

Figure 11.2: MidArch Repository Tool: Plug-in Structure

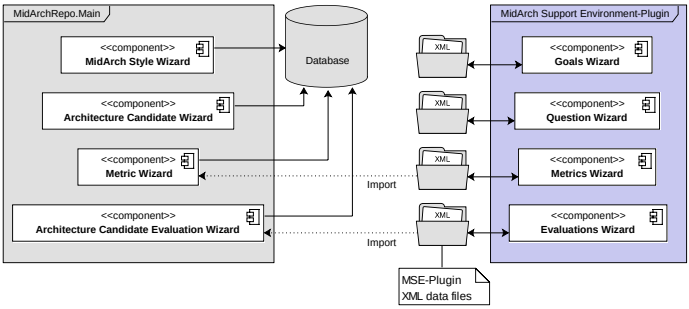

Figure 11.3: Data Exchange between the MidArch Support Environment and the MidArch Repository Tool

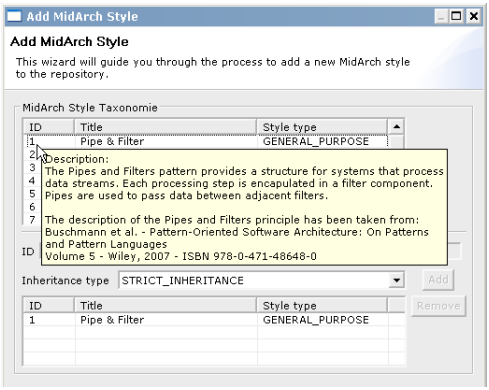

Figure 11.4: MidArch Repository Tool Screenshot: Style Import Wizard

Evaluation The MidArch Repository Tool has been evaluated by retrospectively applying it to the results from the RegIS Online case study (Section 10.1).

First, the styles that have been modelled in the case study were added to the repository. Figure 11.4 shows the wizard that allows the addition of styles to the repository. Similarly, the candidate architectures were added to the repository. Then, the quality model and the evaluation results for each architecture were added to the repository.

Figure 11.5 shows a visualisation for the assessment of architecture evaluation results. On the left, the project and architectures that should be compared are selected. Below, the metrics for which evaluation results for these architectures are available are shown, and a selection among those can be made. On the right hand, a kiviat chart for the selected architectures and metrics is shown. The kiviat chart can be exported as a graphics file. An alternative visualisation is a bar chart, as shown in Figure 11.6.

Outlook In its current version, the MidArch Repository Tool only offers some core functionality, which centres around the storage of MidArch Taxonomies. It could be extended towards a tool which supports all of the tasks of the MidArch Design Method and guides the workflow through an instance of the MidArch Design Method.

11.2 Tool Support for Implementation Mapping

To further validate the usefulness of the MidArch Modelling Approach (Chapter 7), an approach for mapping style-based architectural descriptions

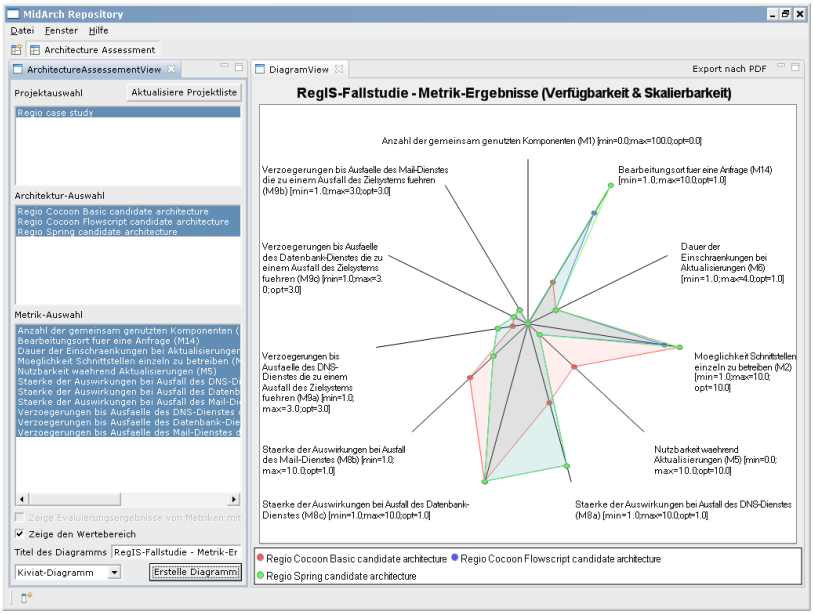

Figure 11.5: MidArch Repository Tool Screenshot: RegIS Online Case Study Kiviat Chart

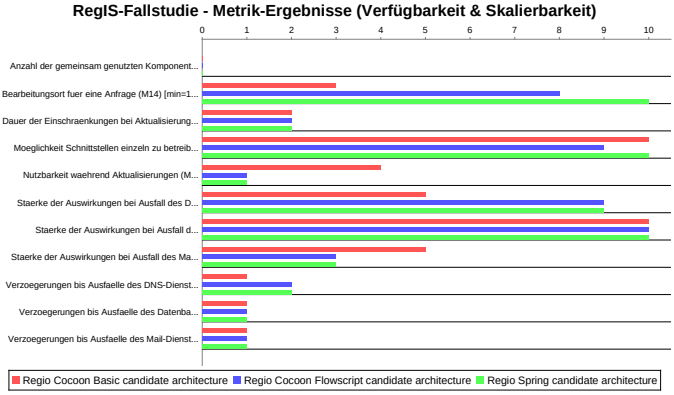

Figure 11.6: RegIS Online Case Study: Bar Chart

to implementation elements was conceived. The mapping approach could also be used with the overall MidArch Design Method (Chapter 8), however this has not yet been evaluated in detail. An Eclipse-based tool, the *ArchMapper*, supporting the mapping approach was developed as part of a student thesis [Gottschalk, 2007].

The ArchMapper tool supports two tasks:

1. generating code skeletons and configuration files on the implementation level from an architectural description based on a MidArch Style,

2. checking the conformity of the implementation with the architectural description and the MidArch Style (cf. Section 2.3.2).

In addition, the mapping approach can be used for improving traceability of implementation elements to architectural elements, but this task is not currently supported by the tool implementation specifically.

Conceptually, the approach extends existing architecture-to-implementation mapping approaches [e.g., Abi-Antoun et al., 2005; Aldrich et al., 2002; van Dijk et al., 2005; Sefika et al., 1996], which allow architectural conformance checks. However, these approaches do not consider architectural styles at all. In particular, they do not exploit the fact that an architectural description is known to be conforming with an architectural style in the definition of the mapping towards the implementation level.

In general, the relationship of the elements of an architectural description and the elements that can be identified on the implementation level can be arbitrarily complex. For example, in general the mapping of architecture-level components to implementation-level components is a m-to-n-mapping. Given an architectural description that is compliant with a middleware-oriented architectural style[1], a much simpler mapping to implementation-level elements is possible, since the types of implementation-level elements are embodied in the vocabulary of the style. Architectural conformance can also be checked more precisely by deriving style-specific conformance checking rules.

ArchMapper currently supports the binding to style and architectural descriptions in Acme, and to Java implementations. The underlying approach is not specific to any language. ArchMapper may be extended towards other ADLs and implementation languages.

The mapping specification consists of a part that is specific to the MidArch Style, which extends the Acme MidArch Style Description, and of a part that is specific to the concrete architecture, which extends the Acme architectural description. The mapping specification is done in a custom

[1] This is not already the case for a general-purpose architectural style, which also constrains the architectural design space, but does not make a specific restriction on the aspects that govern the implementation mapping.

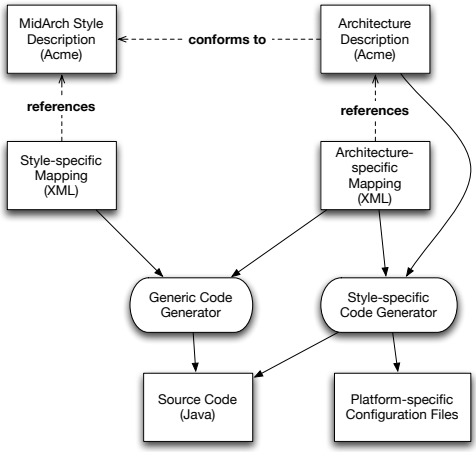

Figure 11.7: ArchMapper Code Generation Approach

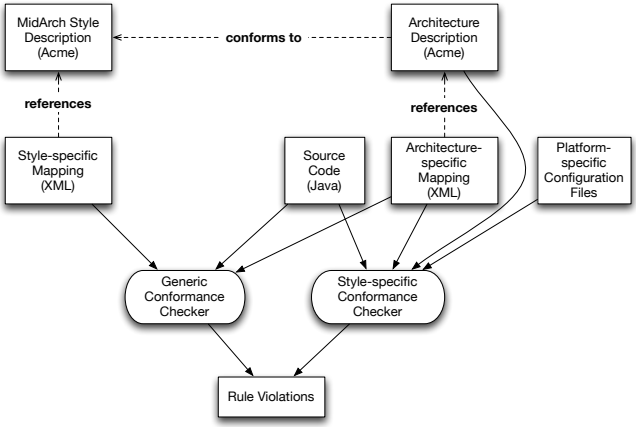

Figure 11.8: ArchMapper Conformance Checking Approach

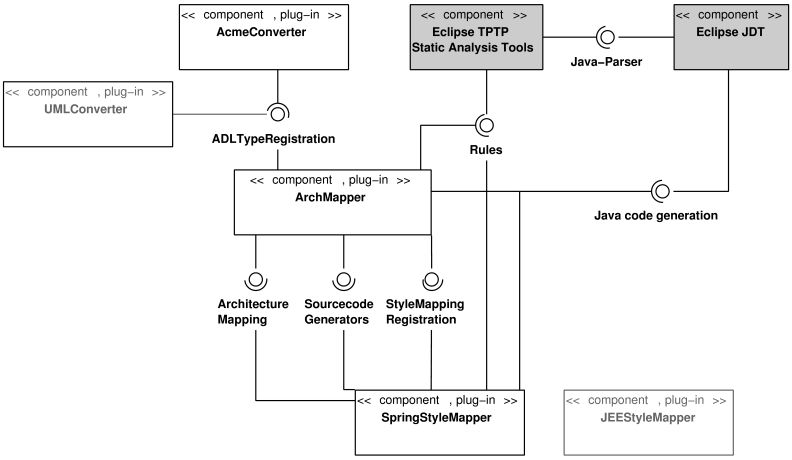

Figure 11.9: Architecture of the ArchMapper Eclipse Plug-ins

XML format. These mappings are used in both code generation (Figure 11.7) and conformance checking (Figure 11.8).

Both the code generator and the conformance checker consist of a generic component and (optionally) a style-specific extension component. The generic component processes the style-specific mapping to interpret the architecture-specific mapping, while the style-specific component processes only the architecture-specific mapping, since the style-specific knowledge is embodied in the code. The generic code generator is able to generate Java code, while a style-specific code generator may also generate other files, for example configuration files for the platform. Vice versa, the generic conformance checker is only able to check Java code, while a style-specific conformance checker might also check other files.

Architecture Figure 11.9 shows an overview of the architecture of the ArchMapper tool, which is implemented on the Eclipse platform. It extends both the Eclipse Java Development Tools (JDT) and the Static Analysis Tools from the Eclipse Test & Performance Tools Platform Project. There is a core ArchMapper component which implements the generic code generator and conformance checker as well as the user interface. It uses an internal representation of the architectural description, which is supplied by ADL-specific plug-ins. Currently, only the Acme plug-in is implemented, but UML support could be easily added as indicated in the figure. Style-specific

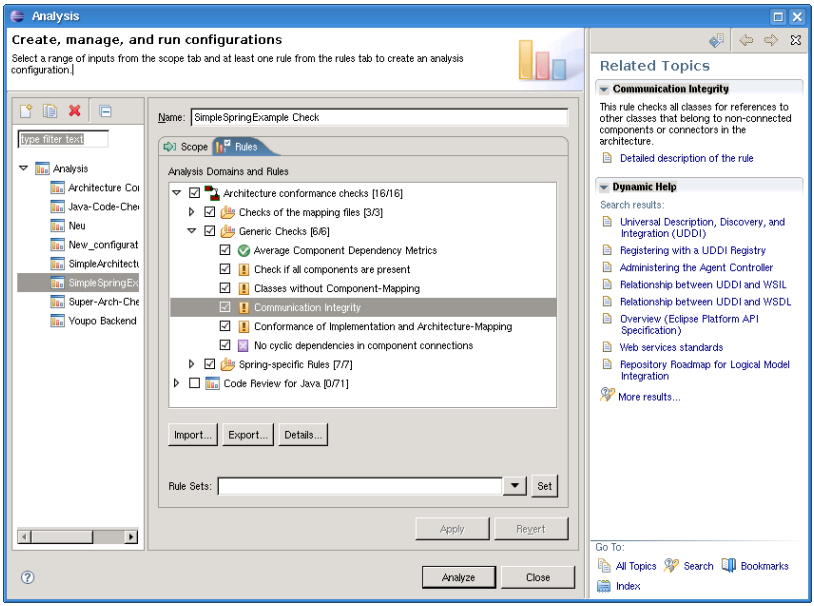

Figure 11.10: ArchMapper Screenshot: Conformance Check Selection Dialogue

code generators and conformance checkers are also added via the Eclipse plug-in infrastructure, as it is currently done with the `SpringStyleMapper` plug-in. While adding support for a new ADL is quite simple and possible without modifying the core ArchMapper plug-in, adding support for other implementation languages is much more difficult, because both the frontend and backend are tightly bound to the Eclipse JDT.

User Interface Figure 11.10 shows a part of the user interface of the conformance check functionality of ArchMapper: The dialogue, supplied by the Eclipse Static Analysis Tools, allows the selection of the checks to be executed. The list shows the generic checks that can be applied to any architecture[2], regardless of the MidArch Style it conforms to.

[2] More specifically, to any architectural description created with the MidArch Approach. All of these architectures could be described to conform to the most basic MidArch Style, which is refined by all other MidArch Styles.

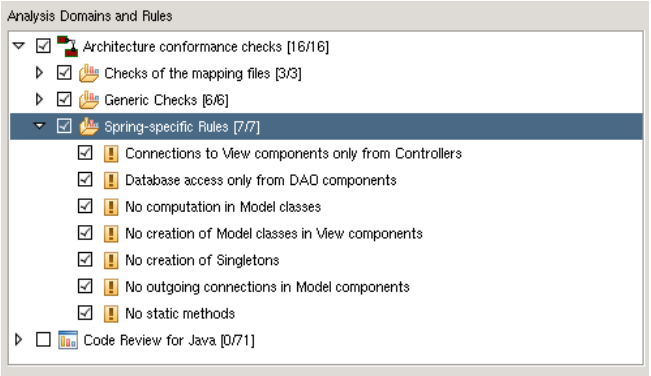

Figure 11.11: ArchMapper Screenshot: Specific Conformance Check Rules for the Spring Style

Evaluation An evaluation of the ArchMapper tool and the underlying mapping approach was made with a case study involving an application by Youpo[3]. The application uses the Apache Spring framework. As part of the case study, a Spring MidArch Style was modelled, the architecture of the existing application was modelled.

Figure 11.11 shows the conformance check rules that have been defined specifically for the Spring MidArch Style, which is a specialisation of the general-purpose Model-View-Controller style.

One of the architectural rules is the "No computation in Model classes" rule. It is an architectural rule, since it governs the global interaction structure within the application. It may be stated on the architectural rule, however it cannot be checked on the architectural level, but only using the implementation. In terms of the implementation elements, it means that classes within a Model component may only offer simple getter and setter methods. This rule has been violated 40 times in the original implementation that was checked using ArchMapper, which is shown in Figure 11.12.

Additional violations of architectural rules that have been uncovered by ArchMapper include a large number of violations of the generic architectural rule of communicational integrity, for example direct accesses from classes of the controller component to database classes. Communicational integrity means that accesses between classes of the implementation is only allowed if the components the classes are associated with are connected in the architectural description.

In addition violations of the rules "No outgoing connections in Model

[3] The software company based in Oldenburg is under new management and the former website is now defunct.

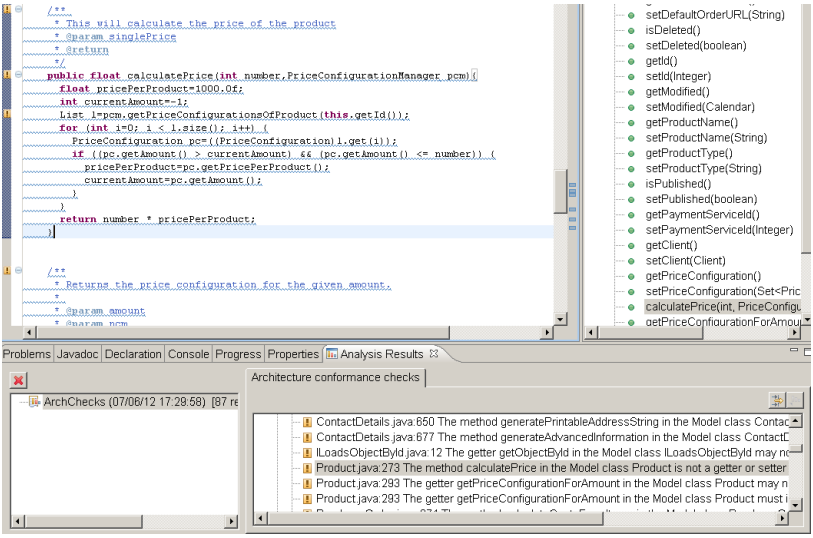

Figure 11.12: ArchMapper Screenshot: Result of a Conformance Check in the Youpo Case Study

components" and "No static methods" have been detected.

While the violations of the communicational integrity rule could have been detected with a generic architectural conformance checking tool, the other architectural rules could not have been stated easily without the unique style-based approach underlying ArchMapper, and hence the detection of their violation would not have been possible.

12 Related Work

In this section, we discuss prior or concurrent work that is related to our goals. We focus on work that is comparable to the two major contributions of this thesis: the MidArch style and Style-based architecture modelling approach introduced in Chapter 7 (see Section 12.1) and the overall MidArch Design Method introduced in Chapter 8 insofar it aims at the evaluation and selection of middleware on the architectural level (see Section 12.2).

12.1 Style and Style-based Architecture Modelling

There has been much work on modelling architectural styles in general. These approaches have been discussed as foundations for our work in Section 3.4. The different approaches use a variety of formalism for representing different aspects of architectural styles relating to different architectural views.

An overview of the limited work on the general relationship of middleware platforms and software architecture is given by Bahsoon et al. [2005]. The foundation for the investigation of platform-oriented architectural styles was laid by Di Nitto and Rosenblum [1999]. There have been approaches extending upon this work [Baresi et al., 2004, 2006; Medvidovic et al., 2003]. All of these approaches are specific to a particular ADL.

The MidArch Style Modelling Approach is unique in defining a conceptual metamodel for modelling architectural styles (with a focus on middleware-oriented architectural styles) independently from a particular notation. In some sense, it is comparable to Acme [Carnegie Mellon University, 2006], which originally aimed at serving as an interchange and integration language for various ADLs. However, it defined its own abstract and concrete syntax and ultimately failed in its goal. MidArch takes a top-down approach with respect to allowing mappings from the conceptual metamodel to the syntax of individual ADLs. Zimmermann et al. [2007] approach the problem from a different direction, they propose a classification of architectural design decisions into four levels, which clearly distinguish platform-oriented from platform-independent decisions.

Two of these mappings were defined in the course of the thesis, to Acme and the UML. The UML mapping deserves further attention, since various other attempts have been made on using the UML for modelling architectural styles. These approaches are of particular interest, since the UML is a comprehensive language which bears the potential of extending style descriptions and style-based architecture descriptions to further architectural views. Furthermore,

it is better known both within the research community and in industry than any of the ADLs. These approaches are discussed in Section 12.1.1.

12.1.1 UML-based approaches

To our knowledge, there is no prior work that proposes a platform-independent approach to model styles and style-based architectures in the UML. However, there has been work on style-based architectural description with other languages.

Design patterns, which are similar in content to architectural styles, have been described in the UML as well. The problem of mapping existing languages other than ADLs to the UML has also been treated.

UML Profile-based Architectural Modelling Approaches Some authors have proposed *generic* profiles for modelling software architectures in the UML [e.g., Hudaib and Montangero, 2002; Kandé and Strohmeier, 2000], but they do not consider style-specific profiles.

Zdun and Avgeriou [2005] provide a lightweight extension to the UML 2.0 [OMG, 2005c], i.e. a UML2 profile, to explicitly represent architectural patterns in UML Composition Structure diagrams. They specify stereotypes and OCL constraints for several UML 2 Metaclasses occurring in Composition Structure diagrams to define semantics for instances architectural primitives. They intend to overcome certain limitations of the formalisation of patterns, such as the restriction that variants of patterns can only be specified in isolation.

As opposed to the approach described by Englisch [2006], the patterns are explicitly modelled as part of the UML diagrams, not as meta-information in a separate file. To be precise, the patterns are not modelled at all, only their instances are. The modelling of the instances relies on the architectural primitives, which are themselves modelled as a UML profile with additional semantics described textually. They represent an instance of a variant of an architectural pattern by referring to architectural primitives.

Platform-specific Profiles Roh et al. [2004] pursue a similar approach as we do: They define a domain-independent base profile and domain-specific profiles that specialise the former. However, their profiles are not correct UML profiles: They define associations between Stereotypes, which is not allowed. It remains unspecified how the Collaborations they define are instantiated to CollaborationUses. A similar problem also applies to Fuentes et al. [2002].

Selonen and Xu [2003] adopt the concept of UML profiles for a mapping of a set of architectural views (e.g., structural view and behavior view) to a taxonomy of related profiles. The approach of Baresi et al. [2006] is similar to our concept concerning their design of a metamodel for architectural

description. In essence they propose to define UML profiles for platform-specific models (e.g., a SOA profile).

Zarras et al. [2001] discuss the design of a base profile for architectural description with the UML 1.3. They mention the possibility to use UML profiles for the description of architectural styles, but do not present a detailed approach.

Garlan et al. [2002] present several mapping alternatives to translate the basic modelling constructs of the component-and-connector view to UML metaclasses. Similarly to our approach, they define criteria (e.g., semantic match and visual clarity) to make a selection.

Mapping ADLs into the UML Other approaches to map ADLs to the UML exist, such as Inverardi et al. [2005]; Medvidovic et al. [2002]; Oquendo [2006], but they do not address the problem of describing styles and style-based architectures.

Heavyweight extensions to the UML In addition to approaches that use UML's lightweight profile extension mechanism, there are also approaches that use heavyweight extensions to the UML metamodel [Kacem et al., 2006; Pérez-Martínez, 2003]. The problem with these approaches is that they break UML tools (as well as the understanding of the modeling elements), and so significant benefits of using a UML-based approach are lost, while the complexity of UML is kept.

12.1.2 Non-UML-based approaches

Mehta presents an approach [Mehta and Medvidovic, 2002, 2003, 2004] to describe architectural styles using architectural primitives. Architectural primitives are elements that are used to describe architectural styles, which occur in various architectural styles and reside on an abstraction level above the usual building blocks. These architectural primitives have been used to define the ADL Alfa (see also Section 3.5.2.7). Essentially, it is another approach to describe architectural styles that is bound to a special-purpose notation. Since the ADL has not been used other than by its author, its applicability cannot be judged yet.

Schwanke [2001a] proposes two notations: the pattern-composition diagram for representing an architecture in the real-time event flow style [Schwanke, 2001b] and the patterns it uses, and the attribute/decision graph to represent the rationale of such an architecture. By rationale, Schwanke refer to dependencies of design decisions and the properties[1] of the system they aim to support. A pattern-composition diagram shows the

[1] Schwanke [2001a] refer to these by the term "attributes", which is reserved for quality attributes (cf. Section 2.3) in the terminology of this thesis. Schwanke refers to arbitrary properties of the system or its requirements.

system as a hierarchy of nested layers. The patterns used are identified as textual annotations within the label of each layer. The linking of these patterns towards implementation elements is not discussed. This link may be regarded as being implicitly determined by the conventions of the style used, but the description is not detailed enough to make a definite statement.

12.2 Architecture-Level Evaluation and Selection of Middleware

The work we consider related to the MidArch Design Method as a whole approaches the evaluation and selection of middleware at the architectural level that is driven by specified technical or business goals. In order to be not too restrictive in the consideration of other approaches, we do not require that other work makes the distinction of middleware products and platforms, and we consider a wide view of architecture. However, we require that a systematic method is described that is intended to be applied in various project contexts and to a variety of middleware products. Isolated case studies concerning the selection of middleware in a particular project, or approaches targeting the selection among a specific set of middleware products are not immediately related to our approach (such as middleware benchmarks [Liu and Gorton, 2004; Maheshwari and Pang, 2005; Ran et al., 2002]).

If we further restrict the criterion to methods that consider the details of architectural descriptions in making the selection, we exclude work that evaluates middleware on a feature-oriented basis, such as the i-Mate process [Liu and Gorton, 2003].

If we additionally require that the architectural description to be considered is not an archetypal architecture tied to the middleware, but a genuine architectural description related to the project context, we also include architecture-oriented methods such as the MEMS method [Liu et al., 2006]. However, we will have a closer look at the MEMS method below in Section 12.2.2, since it is the most closely related method known to us.

In summary, there is currently no design method known to us that combines the following features of the MidArch Design Method:

- it supports the architect in designing a software architecture,
- it supports the architect in making a decision among middleware (platform) alternatives,
- it includes both the project's requirements as well an architectural description specific to the project in making the decision.

General-purpose architecture evaluation methods could also be compared to the MidArch Design Method, however this requires a more rigorous inves-

tigation, the methods must be tailored towards the specifics of middleware-intensive application to make a well-founded comparison.

In the tasks of the MidArch Design Method, various existing techniques are used, for example for making trade-offs amongst conflicting requirements. Since no specific contribution to these techniques is made in this work, it is not considered in the examination of related work.

In the following, two approaches are discussed in more detail: the ArchPad Method (Section 12.2.1) and the MEMS Method (Section 12.2.2).

12.2.1 Zimmermann et al.: The ArchPad Method

Zimmermann et al. [2008] introduce the ArchPad Method, which combines the use of architectural patterns (i.e. architecture-level design patterns) and reusable architectural decision models (RADMs). An individual RADM captures the architectural decisions that are connected with the use of a particular technology platform, such as SOA. In some sense, a RADM is comparable to a subtree of a MidArch Style Taxonomy which contains all the variant styles that relate to a given family of middleware products. While the elements of such a subtree are architectural styles in our approach, the elements of the RADM are individual architectural decision templates. It should be possible to define a mapping between these two representations, which make different aspects of the middleware selection explicit. The strength of the RADM approach is that dependencies between design decisions are explicitly contained in the model, and the architect is guided through a sequence of decisions that need to be taken for the chosen technology. However, the approach does not support the architect in preselecting the technology. Combining all technologies into one RADM would not be feasible, since the combined RADM would be too large to be practically manageable.

12.2.2 Liu et al.: The MEMS Method

The MEMS (Method for Evaluating Middleware Architectures) is an architectural evaluation method designed for the application to the comparative evaluation of middleware technologies with respect to multiple quality goals. It is a scenario-based techniques, where scenarios are formulated as natural language in the form of a "utility tree" [Bass et al., 2003, ch. 3].

Architectural Level In Liu et al. [2006], there is a slight inconsistency regarding the architectural level. The architectural level actually regarded is exactly the same as in our approach, however sometimes the paper conveys the impression that they would be evaluating the level of middleware itself. This becomes most apparent in the discussion of software architecture evaluation methods, which are deemed to be on application level in contrast to MEMS. MEMS uses middleware concepts as a basis for modelling the architectural

MEMS Activity	Techniques
Generate Key Scenarios	ATAM Utility Tree
Define Quality Attribute Scale	Weighted Scoring Method (WSM)
Determine Architecture Alternatives	
Evaluate Quality Attributes	Inter-rater reliability analysis method
Present Evaluation Results	Radar diagram

Table 12.1: Techniques used within MEMS

alternatives, which themselves are on the application level. This does not undermine the influence of middleware on the overall system quality. In the section on related work (Section 2.1 in Liu et al. [2006]), however, it is stated that MEMS actually is a scenario-based software architecture evaluation method.

Specificity to Middleware Middleware is regarded as some distributed component environment, which is typically an off-the-shelf product developed independently from the application under consideration. This means that business goals of the middleware supplier and of the application developer do not match. The consequences of this for the design of the method remain implicit in Liu et al. [2006].

Middleware-intensive applications depend "on the mechanisms and services provided by the infrastructure". The consequence for Liu et al. is that technical documentation of the middleware must be considered in detail. We make this knowledge explicit by capturing this dependency in the formalisation as an architectural style.

In conclusion, while the method is derived from several middleware-specific assumptions, these are only implicit in the final method and it could also be applied in other contexts.

Focus The method focuses on leveraging existing techniques in an overall evaluation method. The techniques used are summarised in Table 12.1.

Matching of Activities The activities of MidArch and of MEMS match not exactly, but to a significant degree. Table 12.2 gives an overview.

The definition of the scope is implicitly assumed to have been accomplished before MEMS starts, so this task is not matched. The MidArch definition of the quality model covers most of the first three MEMS activities, which is more fine-grained in this respect. The generation of scenarios is partially done in a separate step prior to the definition of the quality model in MidArch.

The selection of architectural styles is not part of MEMS. Thus, the task of defining architectural alternatives will be much different in instances

MEMS Activity	MidArch Task
Determine Quality Attributes	2B
Generate Key Scenarios	2B, partially 1B
Define Quality Attribute Scale	2B
Determine Architecture Alternatives	3C
Prototype	optionally: 3D, 4A
Evaluate Quality Attributes	3D
Present Evaluation Results	3E

Table 12.2: MEMS vs. MidArch activities

of the two methods. Prototyping is not considered a defining property in MidArch, but may be performed as part of evaluating individual architectures depending on the metrics in the quality model, or in the selection of the architecture to adopt.

The evaluation of quality attributes is an activity that is mostly equivalent in both methods. Presentation of the results is part of task 3E in MidArch.

A MEMS instance ends with the presentation of evaluation results, the selection of the architecture to adopt (possibly employing trade-off analysis) and the adoption itself are not considered to be in the scope of the method, so the entire fourth MidArch activity is not matched.

Further Method Design Decisions Liu et al. argue that the evaluation should be driven by application-independent quality attributes rather than business goals, due to the technical nature of middleware technologies and their indifference to vertical application domains (which distinguishes MEMS from other architectural evaluation methods such as SAAM, ATAM and i-Mate [Liu and Gorton, 2003]). While the motivation is true, in our view the quality attributes that are considered must still be derived from the business goals, otherwise the evaluation will be decoupled from the actual needs (implicitly, this is also the case in Liu et al. [2006]). This is the reason for us to apply the GQM method, which starts from (business) goals and derives technical metrics from them in a systematic way.

Liu et al. state that it is "difficult to evaluate the quality of a complete middleware technology", which is the motivation for the scenario-based approach. This concurs with our view, while we did not originally focused on a scenario-based approach.

Part IV

Conclusions and Future Work

13 Conclusions

In this chapter, we summarise the results achieved in this PhD thesis. The main contribution of the PhD thesis is the MidArch Design Method. The MidArch Design Method is a new approach to the early selection of middleware within a development project, which combines the selection of a middleware platform with modelling application architectures using that platform. Within one instance of the MidArch Design Method, multiple candidate architectures using different middleware platforms are modelled. Middleware platforms are modelled using middleware-oriented architectural styles (MidArch Styles). The candidate middleware platforms are chosen in a systematic way, thereby the method represents an architecture-level design exploration approach. By isolating the reusable design knowledge within the style, the effort for modelling individual architectures is reduced and the conceptual integrity of the resulting architectural descriptions is improved as a side effect. The use of architectural styles as the means for middleware selection is a novel contribution of this thesis. The MidArch Design Method has been partially evaluated using two case studies that use real-life examples as subject systems.

In the following, we relate the results to the work packages and research questions stated in Chapter 4. The research questions are recited at the beginning of each section. Research questions Q4.3 and Q5.2 have not been addressed in this thesis, and are thus not discussed in this chapter, but will be covered in the discussion of future work (Chapter 14).

Figure 13.1 lists the work packages of the thesis. It extends Figure 4.1 by including the projects that have been carried out in student theses, which contribute to the goals of this PhD thesis. These student these have been planned as part of the PhD project, and they were supervised by the author of the PhD thesis. The contributions of these student theses are referred to in the following discussion of the answers to the research questions.

13.1 WP1: Analysing Architectural Styles and their Usage

13.1.1 Q1.1: Nature of Architectural Styles

> *What is the nature of architectural styles and how are they related to similar software architecture concepts?*

We analysed the nature of architectural styles as defined by various authors, and of related concepts such as architectural patterns and reported upon

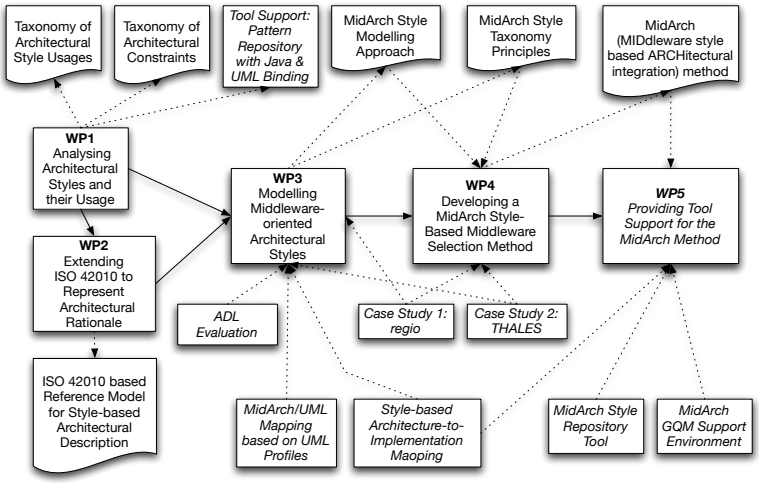

Figure 13.1: Work Packages of this PhD Project including contributing Student Theses

the results in Giesecke et al. [2007b]. A summary of this article has been presented in Section 3.4. In particular, we analysed the differences between the understanding of the pattern community and of the style community (see Section 3.4.3.1). While architectural styles and architecture-level design patterns embody the same ideas, the core differences are that:

- Design patterns are considered to intrinsically represent proved design knowledge, while architectural styles refer only to the formal representation. Evaluation of architectural styles is only done after they have been described.

- Design patterns are considered to be an archetypal description and they are instantiated locally within a design description. Multiple design patterns may be instantiated within the same design description. An architectural description (in the corresponding viewpoint) is considered to be an instance of an architectural style as a whole. The question of whether a design description conforms to a design constraint makes only sense for architectural styles.

- Within the style community, the constraints are usually considered to be described in a formal language, while this is not assumed by the majority of the pattern community. The formal nature of architectural style is a prerequisite for effectively performing conformance checks.

As basic elements of an architectural style we identified:

- A vocabulary of design elements, i.e. component and connector types.
- A set of configuration rules.
- A semantic interpretation of the design elements.
- Optionally, analyses for configurations of that style.

The results of this study have provided the initial input for the subsequent definition of the MidArch Metamodel (see Q3.1).

In his thesis, Englisch [2006] analysed and modelled types of relationships between design patterns. This work was done as part of the development of an approach to relate UML-based pattern descriptions and their uses in Java applications in a design pattern repository.

13.1.2 Q1.2: Existing Architectural Style Usages

> *Which usages of architectural styles exist and how are they related to each other? (or: What is the state of the art in using architectural styles?)*

Existing architectural style usages have been analysed on the basis of a literature review. We have categorised the usages in a taxonomy in Giesecke and Hasselbring [2006], which has been the basis for Chapter 6. We consider the resulting taxonomy to be representative of the research literature, which identifies the following usages: Ad-hoc, Platform-oriented, Customised, Pre-modelling, and Post-documentation and analysis.

13.1.3 Q1.3: New Architectural Style Usages

> *How can these usages be made more effective, e.g. by combining their features?*

The taxonomy of architectural style usages resulting from research question Q1.2 has been used for defining the details of the remainder of the research design. We chose a combination of the platform-oriented and the customised usage as the basis of the overall MidArch Design Method, which is discussed in connection with research question Q4.1. The proposed method allows the use of architectural styles for middleware selection, which has not been possible before.

13.2 WP2: Extending ISO Standard 42010 to Represent Architectural Rationale

13.2.1 Q2.1: Relationship of Architectural Styles and Architectural Rationale

How do architectural styles relate to architectural rationale?

We have postulated that an architectural style codifies a reusable set of architectural design choices. Architectural styles are traditionally considered for some variant of the component & connector viewpoint. A description of an architectural style is always bound to a specific architectural viewpoint and its use is established within one of an architectural description's architectural view. This informal notion was formalised in the answer to research question Q2.2.

13.2.2 Q2.2: Architectural Styles and ISO Standard 42010

How can architectural styles be reflected within a standardised conceptual framework for architectural description?

We have chosen the conceptual framework for architectural description that is defined by ISO Standard 42010 [ISO, 2006] as the basis for reflecting architectural styles. An extension of the reference model was proposed in Giesecke et al. [2006] and is presented in this thesis in Section 5.7. It refines the representation of architectural rationale by relating the application of architectural styles to architectural views. It formalises the relationship of architectural styles and architectural rationale as discussed for question Q2.1.

The IEEE working group for the revision of ISO Standard 42010 aims to refine the representation of architectural rationale in the standard and considers our contribution in the revision process.

13.3 WP3: Modelling Middleware-oriented Architectural Styles

13.3.1 Q3.1: Suitability of ADLs

Which ADLs are suitable for modelling MidArch Styles and architectural descriptions exploiting MidArch Styles?

The MidArch Metamodel (Section 7.6.1) and the overall MidArch Style Modelling Approach (Chapter 7) have been designed in accordance with the extended reference model (cf. Q2.2).

Hilbrands [2006] initially evaluated the suitability of ADLs for modelling MidArch Styles. Based on this work, the ADL Acme was selected as a first

target language for defining a mapping of the MidArch Metamodel. An additional mapping to the UML, which uses UML Profiles and composition structure diagrams was defined [Giesecke et al., 2007c; Marwede, 2007] (see Section 7.7.2). By defining these two language mappings of the MidArch Style Modelling Approach, we demonstrated the generality of the approach. The usability of the mappings was evaluated in the context of the two case studies. The models resulting from the Acme are simpler, since the language constructs of Acme are closer to the MidArch Metamodel. However, the UML approach is assumed to lead to more widely understood models, since the UML is widely known. In addition, the comprehensiveness of UML has the potential for integrating the style and architectural descriptions with other architectural views.

Jung [2008] demonstrated the suitability of description logics for modelling metadata of MidArch Styles by defining and evaluation an approach for using OWL-DL for this purpose.

13.3.2 Q3.2: Induction of MidArch Styles

> *Can formalised MidArch Styles be induced from actual middleware platforms?*

The applicability of the MidArch Style Modelling Approach was demonstrated in the context of the two case studies, where MidArch Styles for the Cocoon and Spring middleware products as well as the HLA (High Level Architecture) standard for simulation systems and the SOA (Service-oriented Architecture) approach were defined. Another style for the Spring platform focusing on the implementation of the Model-View-Controller paradigm was defined by Gottschalk [2007].

In addition, the applicability of the metadata modelling approach of Jung [2008] was shown by applying it to the MidArch Styles identified by Bornhold [2006] and to the architectural styles identified by Shaw and Clements [1997].

13.3.3 Q3.3: MidArch Style Taxonomies

> *Is it possible to organise MidArch Styles for a given set of middleware platforms in a taxonomy including specialisation and other relationships?*

The principles for relating MidArch Styles, i.e. for defining a MidArch Style Taxonomy, have been defined in Section 7.5. They have been successfully applied in the theses of Bornhold [2006]; Eiben [2008]; Gottschalk [2007] to model MidArch Taxonomies, which shows that it is possible to organise MidArch Styles for these middleware platforms in a taxonomy using the

MidArch Taxonomy Principles. The MidArch Taxonomy from Bornhold [2006] was used to feed the MidArch Repository developed by Ulbts [2008].

13.4 WP4: Developing a MidArch Style-Based Middleware Selection Method

13.4.1 Q4.1: Method Engineering

> *How can architecture-level software design be supported by a systematic method exploiting a MidArch Style Taxonomy for selecting an appropriate middleware platform?*

The MidArch Design Method was defined on the basis of the MidArch Style Modelling Approach. It has been presented in Chapter 8, and consists of four activities:

Definition The scope of the project and the goals to be achieved are defined.

Preparation A project-specific quality model is developed, and the current architecture is modelled, if an architecture description does not yet exist in a suitable form.

Exploration This activity involves the preselection of MidArch Styles, modelling of candidate target architectures that conform to the selected styles, and the evaluation of the resulting architectures. For evaluation, existing architecture evaluation methods are used. Finally, the evaluation results are assessed to decide whether a candidate target architecture (or a set of such) that allows to achieve the stated goals has already been found.

Implementation Based on the exploration activity, the architecture to be implemented is defined, which might involve adapting the chosen target architectures for practical reasons, and the implementation is performed.

These activities are divided into several tasks. The central and distinguishing activity is the exploration activity, which also consumes most of the time of an instance of the MidArch Design Method.

Gottschalk [2007] has developed an approach for defining mappings form the architectural level to the implementation level, which are partly based on the MidArch Style that is used for the architectural description. This mapping can be used to generate implementation skeletons and for checking the conformance of the implementation with both the rules of the MidArch Style and the architectural description.

13.4.2 Q4.2: Applicability

Is the proposed method applicable?

The applicability of the proposed MidArch Design Method has been demonstrated using two case studies, as it was planned in the initial research design. These have been conducted in student thesis in cooperation with industry. The first case study was done by Bornhold [2006] in cooperation with Regio Institut, Oldenburg, and the second case study was done by Eiben [2008] in cooperation with Thales Defence Deutschland, Wilhelmshaven. A report on the core results of the two case studies is given in Chapter 10. The case studies have shown that the MidArch Design Method is applicable. Modelling MidArch Styles and style-based architectures was demonstrated to be possible for both the Acme and the UML mapping of the modelling approach. Both case studies showed problems in selecting suitable metrics that could be efficiently evaluated on the basis of the modelled architectural configurations.

A significant remaining problem is that for industrial use in a real-world project setting the method is currently too costly. The effort for applying the method would be significantly reduced if no new MidArch Styles would need to be modelled, but a MidArch Repository filled with suitable MidArch Styles and evaluation results would already exist.

13.5 WP5: Providing Tool Support for the MidArch Design Method

13.5.1 Q5.1: Tool Support

How can the proposed method be supported by tools?

Three tools have been developed for supporting the MidArch Design Method as part of student theses. First, Gottschalk [2007] developed a tool that implements the architecture-to-implementation mapping approach discussed under research question Q4.1, which is presented in Section 11.2. Second, Jung [2007] developed a tool supporting task 2B of the MidArch Design Method, which can also be used to support the application of the GQM method in general. This tool was extended by Ulbts [2008], who implemented an initial version of the MidArch Repository tool. The tool has been presented in Section 11.1.

14 Future Work

In this chapter, opportunities for future work on the basis of the presented thesis are discussed. The presentation is organised by the work packages (see Chapter 4) the extensions are most related to.

First, we discuss ideas for extending upon the general character of architectural styles and their usages (Work Package 1, Section 14.1). Then future work regarding the ISO reference model for architectural description is presented (Work Package 2, Section 14.2). Section 14.3 discusses extensions to the MidArch Metamodel and on the method for modelling styles and style-based architectures using the MidArch Metamodel (Work Package 3). Then, extensions to the overall MidArch Design Method (Work Package 4) are discussed in Section 14.4. Tool support (Work Package 5) is also considered in that section.

14.1 WP1: Analysing Architectural Styles and their Usage

The existing work on the identification and classification of design pattern relationships could be used to refine the relationships of architectural styles.

The taxonomy of architectural style usages, which has so far been based on literature only should be verified and refined using studies in the field.

14.2 WP2: Extending ISO Standard 42010 to Represent Architectural Rationale

In this section, opportunities for future work with regard to architectural description concepts are discussed.

The reference model for architectural descriptions defined by ISO [2006] (see Section 3.2) has been initially extended in this thesis (see Section 5.7) to reflect architectural styles.

Example applications of our work by mapping architectural description methods to the proposed standard extension should be provided. In particular, we consider mapping the viewpoint template from Kandé et al. [2002], the template for documenting architectural decisions from Tyree and Akerman [2005] and the descriptions of design patterns as proposed in the UML Profile for Patterns [OMG, 2004a].

A combination of our extension and the related extension proposed by Sarkar and Thonse [2004] should also be considered.

14.3 WP3: Modelling Middleware-oriented Architectural Styles

14.3.1 MidArch Metamodel

The MidArch Style Modelling Approach has been proposed in Chapter 7. Here, we discuss extensions of the underlying MidArch Metamodel with respect to platform-supplied services, combinations of styles, and deviations in style-based architectural descriptions.

All proposed extensions of the MidArch Metamodel would need to be included in the ADL mappings. The specific problems arising from integrating such extensions in the existing Acme and UML mappings are not further discussed at this point.

Platform Services Platform services are currently modelled as platform components within the style, which makes the distinction of platform components and application components explicit. However, this results in inhomogeneous architectural descriptions, and complicates the mapping to ADLs. In the UML, no components can be defined on the M2 level, they can only be indirectly required through OCL constraints. In Acme, a default structure can be defined within the style definition, but the interpretation of this default structure is on the application level, and Acme does not distinguish between components of the default structure and other application components defined in an individual system. Furthermore, it is not currently possible to specify the access to platform components of different platform levels. Both aspects should be improved in a revision of the MidArch Metamodel.

Style Combinations In the thesis, the problem of automatically deriving style combinations was not addressed. While an automatic derivation of style combinations will not lead to meaningful results in general, in special cases a derivation may be possible. Work is required to identify these cases and devise derivation algorithms for these cases.

One specific case in this context is the horizontal heterogeneity of architectural styles. Two styles are not merged but exist in distinct sections of the architectural configuration. However, the interactions between the sections need to be specified independently from either style. The MidArch Metamodel needs to be extended to allow the specification of configuration sections, architectural styles for each configuration section, as well as some means for specifying the bridges between the sections. With the current modelling approach, this situation requires manually specifying a single combined MidArch Style that reflects the styles for each partition as well as the bridging rules. One problem arising from this approach is that it is difficult to clearly separate the partitions and their architectural styles from each other.

Another case is the vertical heterogeneity of styles, where different architectural styles are applied on different architectural layers. Suppose there is an upper layer where a client-server style is applied. The server internally is organised in a pipe-and-filter style. This case is much simpler than the horizontal combination case, since the style definition do not need to be aware of the combination. What is needed, however, is the ability to model composite components, which requires (delegation) bindings between the ports of the inner components and the ports of the outer component. We do not currently include this ability in the MidArch Metamodel.

It remains subject to further research which kinds of architectures can be covered by allowing style bridges and vertical style combinations.

Style Deviations One argument often held against the very idea of using architectural styles for modelling is that in practise architectures cannot conform strictly to one style. One aspect of the argument is that multiple styles may coexist in real-world architectures, which is addressed by style combinations. However, with style combinations the problem of deviations from either style stays. As a prerequisite to making it possible to document such deviations both the style as well as the architecture need to be explicitly expressed, which was one contribution of this thesis. It remains future work to find concepts and notations for expressing the deviations on this basis.

14.3.2 MidArch Style Modelling Method

We propose extensions and refinements of the approach that refer to the operationalisation of the approach, the adequate degree of detail and rigour to be used in modelling styles, the description of styles using a style description template, the definition of style-specific architectural refactorings and of more refined style relationships within a MidArch Style Taxonomy. The aspects so far are independent from the concrete ADL that is used. Finally, specific extensions to the UML/MidArch modelling approach are discussed, and ideas towards a specific MidArch ADL are proposed.

Tool support is discussed in the next section, as it covers the style-based architecture modelling approach as well.

Operationalisation The style modelling method should be described more operationally as a concrete design method with detailed descriptions of each step. The guidance given to the style modeller is currently limited, most significant choices are left to the discretion of the style modeller. Qualitative case studies with different middleware products should be done focusing on the question which artefacts are best suited as a basis for eliciting the aspects relevant for an architectural style description from a middleware product, and which techniques can be used to make the elicitation efficient and effective. Qualitative content analysis [Mayring, 2003] could be applied

to middleware documentation. An open problem is that aspects relevant to architectural styles could be stated only implicitly in the middleware documentation.

Degree of Detail and Rigour One result of the ReGIS Online case study (Section 10.1) was that the attempt to model all middleware aspects that could be expressed in the style leads to very complex style definitions, which are difficult to understand and use. This problem becomes even more apparent when trying to include behavioural specifications in style definitions (which is probably the reason for the scarcity of work in this direction). On the other hand, leaving out too many details from the style definition makes it impossible to distinguish architectures based on closely related architectural styles. Research is still required on an optimal trade-off between these conflicting design requirements.

Style Description Template The formal description of a MidArch Style could be embedded into a style description template, which is a structured document containing a combination of natural language text and formal description. The style description template could be based on pattern description templates (cf. Section 6.1). This presentation could improve the usability of MidArch Styles.

Style Refactoring Operations Architecture-level refactoring operations have been proposed to reorganise architectural descriptions [Wohlfarth and Riebisch, 2006]. The applicability of such refactoring operations could be improved by making them style-specific.

Refined Style Relationships Manifold further types of relationships might be considered for inclusion into a MidArch Style Taxonomy, especially such relationships that refer to classification dimensions that are only relevant to specific styles. We restricted ourselves to this basic set of relationships in our research.

The problem of relationships between styles that are informally considered to be in a specialisation relationship which cannot be expressed formally has only been addressed initially. These relationships can currently only be expressed by using the generic "related to" relationship. It is desirable to specify the details of the relationship, which deviations were introduced that inhibit the use of a formal inheritance relationship. This is particularly important for styles that participate in multiple (inheritance) relationships. Multiple inheritance is an even greater problem for architectural styles than it is in object-oriented programming, as its plain application almost necessarily leads to unusable architectural styles (cf. Section 7.3.4).

UML/MidArch Modelling Approach Specialisation relationships between styles are of great relevance to the MidArch Design Method. These have not yet been addressed in detail, but they are already defined in the MidArch metamodel and the mapping. Further refinement is necessary.

Opportunities for future work include the incorporation of modelling heterogeneous architectures that have multiple styles, and of style-specific design patterns [Monroe et al., 1997] and their instances.

While generic UML tools can be used for the approach if they support custom UML Profiles, specific tool support for using the style to guide the definition of system architectures would be beneficial.

A UML/MidArch Profile is a form of an explicit platform model [Wagelaar and Jonckers, 2005], which could be used within PIM to PSM transformations in the MDA terminology. A style-independent architecture could be transformed into a style-conforming architecture. This use should be further investigated.

Implementation mappings for the MidArch metamodel for the Spring framework are currently bound to Acme representations. The implementation mappings should be integrated with UML/MidArch as well.

Specialised Style and Architecture Modelling Language As discussed in Section 7.4, it was a fundamental decision within this thesis to focus on the conceptual basis for modelling architectural styles and style-based architectures rather than being distracted by details of designing a new architectural description language. When more experiences with the current ADL mappings have been made, and the MidArch Metamodel has matured, it may make sense to design an ADL specialised for modelling MidArch Styles then. The experiences with the current ADL mappings have shown that both the Acme and the UML mapping are far from being optimal: The Acme language is too simplistic and does not allow integration with views other than the logical component structure view. The UML mapping, on the other hand, is awkward to use since the alignment of the basic architectural concepts used in the MidArch Metamodel (and in ADLs in general) with the modelling constructs of the UML is complicated.

Building a Comprehensive MidArch Style Taxonomy Currently, only a limited number of MidArch Styles have been modelled (i.e. those developed in the case studies described in Chapter 10 and those used in the evaluation of the implementation mapping technique described in Section 11.2). These do not form a contingent MidArch Style Taxonomy, as their possible application areas are too diverse. Furthermore, evaluation results exists for at most one architectural candidate for any of these styles. All of the deficiencies should be remedied by performing more applications of the MidArch Design Method, either in laboratory or field settings in order to obtain comprehensive

MidArch Style Taxonomy and MidArch Repository.

Integration of Architecture Reconstruction Techniques The style-based architecture modelling technique currently only supports forward engineering. Reverse engineering is not explicitly considered. However, style-oriented architecture reconstruction is also an interesting topic: Given the vocabulary and constraints of a MidArch Style, which is assumed to be used in an implementation, the architecture of the implementation should be reconstructed such that the resulting architectural description conforms to the given MidArch Style. In case of deviations from the style, an explicit documentation of them should be part of the results of the technique. The applicability of existing architecture reconstruction techniques, for example cluster analysis techniques, should be evaluated. A style-oriented architecture reconstruction technique could also be used in Task 2A (Section A.3.2.1) of the MidArch Design Method.

14.3.3 MidArch Style-based Architecture Modelling Approach

The MidArch Style-based Architecture Modelling Approach is closely related to the MidArch Style Modelling Approach. In this section, extensions to this approach are proposed. These refer to style-based guidance for architecture modelling and to tool support.

Style-based Guidance for Architecture Modelling The style description itself supports the architect in modelling architectures based on a MidArch Style by providing a more specific design vocabulary and constraining the design space. This aspect could be improved by complementing the declarative style description by an explicit guide on the usage of the style for modelling conforming architectures. Such a guide could be part of a style description template (see above). One way to provide such guidance is to formulate style-specific patterns (as proposed by Monroe et al. [1997]) or transformation rules, which enabled developing an architecture by transforming it step-by-step through style-conforming intermediate partial architectural descriptions.

Tool Support Tool support for modelling MidArch Styles and style-based architectures should be provided that supports all of the proposed techniques. While there is some tool support for modelling styles in Acme (AcmeStudio), the current tool is not very robust and user-friendly. In the case of the UML, various commercial modelling tools exist. Some of these tools also support the definition and use of custom UML Profiles, which is a required feature for the use with the UML/MidArch Style Modelling Approach. However, they do not provide specific support for modelling style profiles based on the MidArch base

profile, nor for modelling architectures using such styles. While the extension of AcmeStudio does not seem to be very promising due to the inherent problems with defining a graphical representation of Acme descriptions, we consider extending a commercial UML tool worthwhile. Initial experiences have been collected with using Rational Enterprise Architect within the MidArch Design Method.

14.4 WP4: Developing a MidArch Style-Based Middleware Selection Method

In this section, extensions to the MidArch Design Method are proposed, which has been proposed in Chapter 8. These comprise the refinement of the method definition, embedding the method in standard software process models and development methods, applying real options theory to value the style selection decision supported by the MidArch Design Method, and the improved integration with architecture evaluation methods. Furthermore, a comprehensive evaluation of the MidArch Design Method is pending. Finally, the extension of the application domain of the MidArch Design Method could be considered.

Refinement of Method Definition The method definition should be refined by classifying the tasks within a general model of architectural design tasks. Several tasks are decision-oriented: Using the terminology of decision analysis [Eisenführ and Weber, 2002], tasks 3A and 3B involve the generation of alternatives, tasks 3C and 3D are concerned with determining the consequences of the adoption of each alternative, and in task 3E the architect makes the final decision on selecting a MidArch Style. The style selection decision should be modelled as a specialisation of a general architectural design decision.

Embedding in Standard Software Process Models and Development Methods The MidArch Design Method is currently modelled in isolation, i.e. without reference to standard software process models or development methods such as the V-Modell, spiral model, Rational Unified Process, or Cleanroom Software Engineering.

Application of Real Options Theory to Style Selection Empirical investigations on the applicability of the MidArch Design Method should be complemented by reasoning on the benefits of the approach in a theoretical model. Real options theory [Schwartz and Trigeorgis, 2001] has been applied to software engineering [Boehm, 2005; Geppert and Roessler, 2001; Sullivan et al., 1997a, 1999, 2001]. The underlying idea is to model the impact of design decisions on the value of the investment into the software

systems, considering that the decisions are taken under uncertainty in a risky environment. There has been some work on applying the theory to the question of architecture selection for middleware-intensive software systems [Bahsoon et al., 2005]. After modelling the style selection decision more rigorously as discussed above, applying the real options approach would enable a more thorough analysis of the consequences of such decisions with respect to the business goals of software development.

Improved Integration with Architecture Evaluation Methods The integration of architecture evaluation methods with the MidArch Design Method could be improved. Currently, it is proposed to use such techniques as part of the the metrics of the quality model used to evaluate architectures in Task 3D. However, the tasks of architectural evaluation methods overlap with those of the MidArch Design Method. Applying the full evaluation methods is a complex activity by itself. The integration of a scenario-based architecture evaluation method with the MidArch Design Method should be modelled more explicitly.

Comprehensive Evaluation of the MidArch Design Method Both case studies were not comparative in nature, i.e. they only performed and analysed the application of the MidArch Design Method, but not of an alternative software design method. This would require a significant additional effort and could not be achieved as part of this thesis. However, in Section 9.3, ideas for evaluating the MidArch Design Method comprehensively, using an experimental setup, have already been discussed.

Extended Application Domain While the technical aspect of middleware-intensity is central to the style-modelling approach underlying the MidArch Design Method, the application to systems other than data-centred (business) information systems could be explored. Specifically, embedded control systems could be targeted, if they are implemented using higher level languages or frameworks that make use of complex programming abstractions.

Appendix

A Definition of the MidArch Design Method

In this appendix chapter, the definition of the MidArch Design Method is given. Basic definitions used in the remainder of the chapter are given in Section A.1. The roles of stakeholders in the MidArch Design Method are discussed in Section A.2. These roles are assigned to the activities and tasks of the MidArch Design Method, which are described in Section A.3.

A.1 Basic Definitions

Figure A.1 shows the concepts used in the definition of the MidArch Design Method. The definition of the terms "actor", "role" and "activity" follows Acuña et al. [2002]. However, we decompose activities into tasks, and distinguish two kinds of roles. Instead of "process model", we use the term "method".

A *software design method* is considered to consist of *activities*, which are subdivided into *tasks*. By this structure, no assumption of the order and frequency of occasion of the activities and tasks is meant to be made.

Roles differ in the immediacy of involvement with a design method: The immediacy reflects the required knowledge of the overall method. Those roles that are immediately involved with the method are the *core roles*. The others are referred to as *supplemental roles*, which are not responsible for producing the final artefacts of the method. For each task, a *primary role* is identified among its associated core roles.

There may be an arbitrary mapping of roles to actors, i.e. each role may be assumed by several actors, and every participating actor may assume several roles. The mapping does not need to be fixed across the activities.

A.2 Roles

Several roles of participants in the tasks of the MidArch Design Method can be identified. Core roles include the *head MidArch architect*, *application architect* and *style modeller*. Supplemental roles are the *application domain expert* and the *platform expert*.

In addition to the major roles mentioned here, minor additional roles (or refinements of the mentioned roles) may appear in specific applications of the method.

The tasks are described in Section A.3 and are summarised in Figure 8.2.

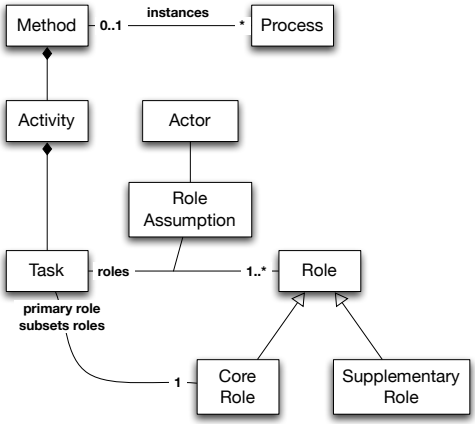

Figure A.1: Design Method Definition Concepts

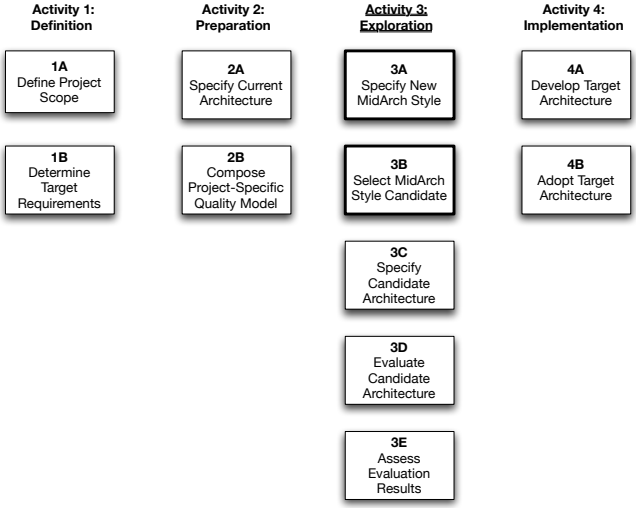

Figure A.2: Activities and Tasks of the MidArch Design Method

Head MidArch Architect The head MidArch architect is responsible for the overall progress of the application of the MidArch Design Method. He supervises the project and coordinates the other participants. We do not identify refined roles of project management that might be delegated to other individuals, since the focus of the discussion of the MidArch Design Method is not on general project management issues.

The head MidArch architect needs to possess profound knowledge of the overall method and is involved in all tasks. He constitutes the primary role of those tasks that do not possess another core role.

Application Domain Expert Application domain experts contribute information on the application domain of the system and its environment. They are primarily involved in the tasks 1A, 1B, 2B and 3D of the MidArch Design Method. The involvement is only indirect, since they serve as interviewees for the head MidArch architect.

Application Architect Application architects are responsible for modelling application-level software architectures for the given requirements. They cooperate with application domain experts (on the problem side) as well as platform experts (on the solution side). They are the primary role of tasks 2B, 3C and 4A. They support the head MidArch architect in selecting style candidates (Task 3B) and evaluating candidate target architectures (Task 3D).

Platform Expert A platform expert[1] is a technical expert who has profound knowledge of a certain middleware platform. Ideally, he has experience in conducting multiple projects using that platform. He is primarily involved in Task 3A, however only indirectly, as he assists the style modeller in that task.

Style Modeller Style modeller and platform expert work hand in hand in modelling MidArch Styles for a given platform. However, the style modeller is the primary role in Task 3A.

[1] As an alternative term for "platform expert", one might consider "platform architect". However, we chose to use the former term, since "platform architect" could imply that this role is concerned with creating the platform or a product that implements the platform. While a platform architect, e.g. the creators of Java EE at Sun Microsystems, is usually also an expert for this platform, the converse is not the case in general. Platform architects in this latter sense are out of the scope of the MidArch Design Method.

A.3 Tasks

In this section, the task associated with each of the activities listed in Section 8.2 are described. For each task, first a general description is given, then some information is given in a structured way:

Internal and external inputs Input artefacts that are used in the task are listed here. Internal inputs are such artefacts that have been produced by other tasks of the MidArch Design Method, while external inputs have been produced independently from the MidArch Method, but are required for its tasks.

Outcome Analogously to the inputs, output artefacts produced by the respective task are listed here.

Roles The roles defined in Section A.2 that participate in the task are listed here.

Dependencies Here, other MidArch tasks the current task depends on in the sense of data dependencies are listed.

Figure A.3 shows the input and output artefacts that are used within the MidArch Design Method.

A.3.1 Activity 1: Definition

The *definition* activity consists of two tasks: definition of the project scope (Task 1A) and determination of the target requirements (Task 1B).

A.3.1.1 Task 1A: Define Project Scope

Internal inputs none
External inputs high-level system documentation of current system and/or its environment
Outcome Project scope definition
Roles Head MidArch architect
Dependencies none

In this task, the overall setup of the project is defined. In particular, the definition of the *system boundary* is part of this task. The system boundary determines which elements of the environment are retained as is but may interact with components of the target system, and which functionality is to be implemented by the target system. In the case of a migration project, the set of components to be replaced is determined. The tasks of this activity are not specific (neither in their goals nor their methods) to the MidArch Design Method, but are part of most software design methods with comparable goals.

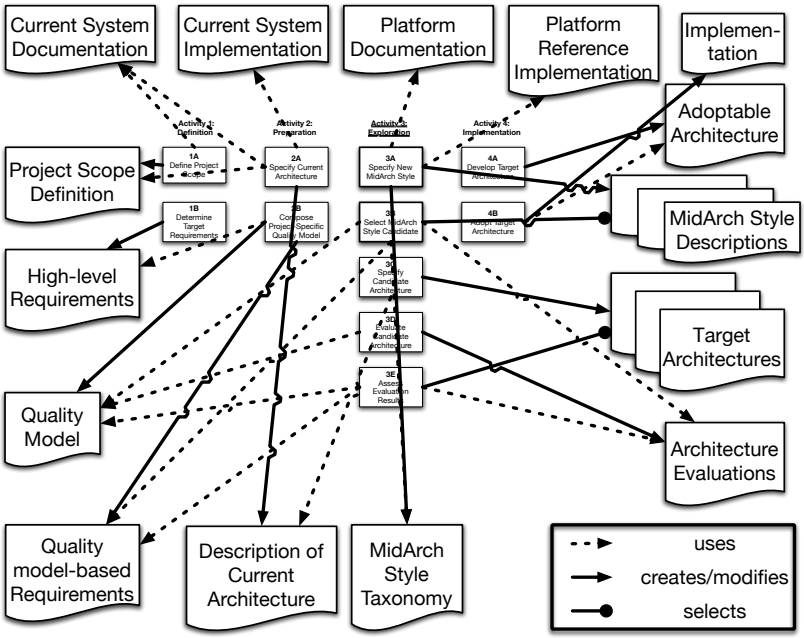

Figure A.3: Artefacts used in the MidArch Design Method

A.3.1.2 Task 1B: Determine Target Requirements

Internal inputs Project scope definition
External inputs none
Outcome List of high-level requirements
Roles Head MidArch architect, Application domain expert
Dependencies Task 1A

In this activity, both functional and non-functional requirements for the target system, as defined by task 1A, are determined. Functional requirements are used in the architecting task 3C for a mapping of functionality towards middleware-oriented components. Non-functional requirements serve as the basis for defining the quality model in task 2B.

A.3.2 Activity 2: Preparation

The tasks of the *preparation* activity are more specific to the MidArch Design Method. They are targeted towards the main *exploration* activity. The first task, the specification of the current architecture (Task 2A) can be found in architecture-based development methods, but the abstraction level and modelling language must be specifically selected to support the style-oriented approach in the *exploration* activity. The second task, the composition of a project-specific quality model (Task 2B), can be found in quality-driven design methods.

A.3.2.1 Task 2A: Specify Current Architecture

Internal inputs Project scope definition
External inputs Existing system documentation & implementation
Outcome Architecture description (current)
Roles Head MidArch architect, Application domain expert
Dependencies Task 1A

The current architecture of the system is reconstructed from available evidence. This task is naturally omitted when building a new system. Otherwise, it will be necessary in most cases, either because no architectural description exists at all, it is not up-to-date and synchronised with the implementation. Even if this is the case, it may be on an unsuitable abstraction level or not described using a notation that is suitable for describing style-based architectures.

A.3.2.2 Task 2B: Compose Project-specific Quality Model

Internal inputs List of high-level requirements
External inputs none
Outcome Project-specific quality model, Quality-model based Requirements
Roles Head MidArch architect, Application domain expert

Dependencies Task 1B

A project-specific quality model is composed in this task, based on the requirements that have been elicited in task 1B. A quality model, however, is a declarative specification of what should be measured (cf. Section 2.4). The quality model is thus not a refinement of the requirements, but the foundation for performing several architectural design tasks on a systematic basis:

1. specify the requirements in terms of the quality model (part of this task),
2. measure software artefacts (Task 3D), and
3. evaluate measurements against the requirements (Task 3E).

We consider GQM-style quality models (cf. Section 2.4) to be used in this task. Whether the full-fledged GQM definition process [van Solingen and Berghout, 1999] is used, remains a decision in a specific MidArch instance. At some point in the interpretation process, expert judgement is required, which is addressed by Task 3E.

Project-specificity of the quality model vs. commensurability of evaluation results There is a trade-off to be made between the project-specificity of the quality model and the commensurability of evaluation results.

We aim at project-specificity of the quality models to maximise the significance of results of architectural evaluations given limited resources. Different software systems show different sets of relevant quality characteristics. This reflects the basic idea of the goal/question/metric approach, which emerged from the unreflected application of large numbers of metrics to software systems, without any idea of how to interpret the results. While the hierarchical structure of the quality models does not enable an automatic interpretation and aggregation of results towards the goal level, it enforces the goal-oriented derivation of the metrics to be applied and provides a guideline for interpretation of metrics-level measurements by experts.

To achieve commensurability of evaluation results, it is not required that the quality models are identical. First, the comparison of evaluation results is indifferent to the structure of the goal and question levels, since the comparison is done on a flat projection of the metrics level only (see Section 2.4). Second, a comparison of measurements can be made on the intersection of the sets of metrics of two different quality models. In that case, no assertion can be made with respect to the metrics outside the intersection.

To support the commensurability of results, a library of metrics should be provided for the definition of quality models: The goal and question levels may remain entirely project-specific, while the metrics level is somewhat harmonised. The metrics used in the case studies (see Chapter 10) provide

an initial basis for this metrics library. Similarly, the metrics from the DMETRICS effort [Eusgeld et al., 2008] may be used for the library.

A.3.3 Activity 3: Exploration

A.3.3.1 Task 3A: Specify New MidArch Style

Internal inputs None
External inputs Platform documentation, Reference implementations
Outcome MidArch Style Description, Updated MidArch Style Taxonomy
Roles Style modeller, Platform expert
Dependencies None

In this task, a new MidArch Style for a platform is specified by a style modeller, who is assisted by a platform expert. The platform expert provides input on the platform in an informal way, which is formalised by the style modeller, who is an expert in modelling styles in the chosen ADL. The details of this task have been elaborated in Chapter 7.

A.3.3.2 Task 3B: Select MidArch Style Candidate

Internal inputs MidArch Style Taxonomy and Evaluations, Project-specific quality model, Quality model based requirements
External inputs None
Outcome Selected MidArch Style
Roles Head MidArch architect, Application domain expert, Application architect
Dependencies Task 1B and 2A

In this task, one or more MidArch Styles that already exist are selected from a MidArch Repository. We assume that existing styles are classified within a MidArch Style Taxonomy and have been evaluated in a previous MidArch instance. The selection process proceeds hierarchically through the MidArch Style Taxonomy.

In this task, the MidArch Repository can be used to perform the selection algorithm described in Section 8.3.4.

Data dependencies Figure A.4 shows the data dependencies of the task, which are more complex than that of most other tasks, since the dependencies must be distinguished into those within the same MidArch application, and those across MidArch applications. Indirect dependencies that exist through other tasks are not shown here.

Within the same MidArch application, there exists a dependency on the specification of the target requirements (Task 1B) and development of the quality model (Task 2A). The target requirements need to be stated in terms of that quality model to serve as an input for matching against the comparative evaluation results.

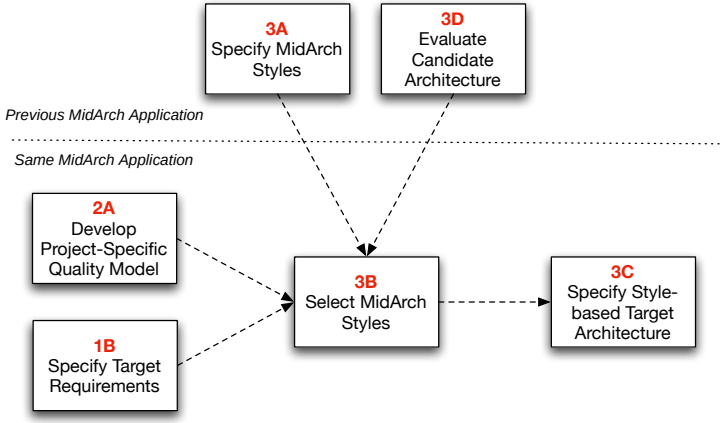

Figure A.4: Data dependencies of the Selection Task

A dependency in the other direction, i.e. a dependency on Task 3B, is that the specification of a candidate architecture based on a certain style (Task 3C) depends on the selection of that style.

To consider a style for selection, it must have been specified first in a previous MidArch instance, and its effects on the quality of conforming architectures must be known to some extent. The style selection task depends on both the style specification task (Task 3A) and the architecture evaluation task (Task 3D). Task 3D is typically performed many times for a style, however it must occur at least once per style. Task 3A, on the other hand, can be considered to occur exactly once for each style. If a style needs to be revised later, the evaluation results that have been acquired so far could be invalidated. Therefore, Task 3A is critical and should be performed with great care.

A.3.3.3 Task 3C: Specify Candidate Architecture

Internal inputs Selected MidArch Style Description, Current architectural description
External inputs None
Outcome Candidate target architectural description
Roles Application architect
Dependencies Task 3A or 3B

In Task 3C, based on the selected MidArch Style Description, which is either a newly modelled style (Task 3A) or a style selected from the repository (Task 3B), a candidate target architectural description is created. If a current

architectural description exists, it is used as a basis for defining the candidate target architectural description. The architecture can be developed through any general software architecture development method. However, the design space is restricted through the chosen MidArch Style.

A.3.3.4 Task 3D: Evaluate Candidate Architecture

Internal inputs Candidate target architectural description, Project-specific quality model
External inputs None
Outcome Candidate architecture evaluation
Roles Head MidArch architect, Application architect
Dependencies Task 3C

In Task 3D, one candidate architecture is evaluated against the project-specific quality model. The methods used for the evaluation are determined by the metrics in the quality model. In this task, only the lowest level, i.e. the metrics level, of the quality model is used, as the interpretation of the evaluation results is left to Task 3E. The metrics may involve appropriate fragments of scenario-based architecture evaluation methods (cf. Section 2.5). However, simpler metrics that can be performed mechanically on an architectural description can also be used.

A.3.3.5 Task 3E: Assessing Evaluation Results

Internal inputs Candidate target architecture evaluation, Project-specific quality model, Quality model based requirements
External inputs None
Outcome Selected target architecture
Roles Head MidArch architect
Dependencies Task 3D

While Task 3D generates architectural evaluation results, this task is decision-oriented like Task 3B. In Task 3B, a preliminary decision is made, i.e. a style is selected for closer inspection. In this task, a final selection decision is made by the head MidArch architect. The goal of this task is to make a decision amongst the candidate architectures that have been modelled in the instances of Task 3C and evaluated in the instances of Task 3D. Typically, a trade-off between the fulfilment of different quality characteristics must be made, so the task involves visualisation of the different evaluations, which allows this trade-off to be made with confidence.

Role of the MidArch Repository As part of this activity, the candidate target architectural descriptions along with the evaluation results are stored in the MidArch Repository. After these basic evaluations have been stored, within the Repository, the evaluations are lifted towards the style level and

propagated upwards within the MidArch Style Taxonomy. Also, the resulting choice and its rationale are stored in the MidArch Repository.

In the MidArch approach, the evaluation of MidArch Styles is performed *indirectly* and *comparatively*. What exactly must be done when adding architecture evaluations to the MidArch Repository is described in Section A.3.3.5.

First, the evaluation is indirect, i.e. the evaluation is not applied directly to the style descriptions, but to artefacts that conform to the styles. An indirect evaluation can either use scenarios that are considered typical for that style, or architectures from genuine software project contexts. In the MidArch Design Method, we chose the second possibility. In our opinion, a direct evaluation of styles is not feasible, i.e., applying some evaluation technique to style descriptions only—without considering other input—would not return meaningful results.

Second, the evaluation is comparative, i.e.:

1. Multiple (at least two) candidate architectures for the same system specification need to be modelled, which conform to different MidArch Styles each (Task 3C),

2. These candidate architectures are then evaluated (Task 3D), which yields evaluation results for each architecture. The metrics defined by the quality model are thus only applied to a single architecture description at once.

3. Now, the evaluation results are compared to determine the commonalities and significant differences between them. These differences are then associated with the MidArch Styles the candidate architectures conform to.

A.3.4 Activity 4: Implementation

Based on Activity 3, the architecture to be implemented is defined (Task 4A), which might involve adapting the chosen target architectures for practical reasons, and the implementation is performed (Task 4B).

A.3.4.1 Task 4A: Develop Target Architecture

Internal inputs Selected target architecture, Current architecture
External inputs Current implementation
Outcome Adoptable target architecture
Roles Application architect
Dependencies Task 3E

Based on the target architecture selected in Task 3E, and possibly the current architecture and implementation, the final architecture that should be adopted is defined. It may be necessary to adapt the selected target

architecture for practical reasons and to deviate from the underlying MidArch Style. These deviations should be documented along with the rationale as part of the description of the adoptable target architecture.

A.3.4.2 Task 4B: Adopt Target Architecture

Internal inputs Adoptable architecture
External inputs Current implementation
Outcome System implementation
Roles Application architect, Software developers
Dependencies Task 4A

In Task 4B, the designated software developers adopt (i.e. implement) the architecture defined in Task 4A, coordinated by the application architect. A suitable architecture-based development method can be used within this task. The task does not specifically make use of MidArch Styles in the basis MidArch Design Method we describe here.

Extension towards Conformity Checks In the thesis, we do not consider checking the conformance of the implementation with the prescribed architecture and the general rules set up by the MidArch Style used. However, for a complete exploitation of the benefits of MidArch Styles, such an implementation-to-architecture conformance check should be performed. To effectively perform an implementation-to-architecture conformance check, it is necessary to define and maintain an explicit mapping of the architecture to the implementation. In defining such a mapping, the style-orientation of the architecture can be exploited. In the context of this thesis, an approach was developed for defining style-based architecture-to-implementation mappings, which is described in Section 11.2.

List of Acronyms

ADL Architectural Description Language

COTS commercial off-the-shelf

CMMI Capability Maturity Model Integration

UML Unified Modelling Language

EAI Enterprise Application Integration

GQM Goal/Question/Metric Paradigm

MDA Model Driven Architecture

OMG Object Management Group

List of Tables

List of Figures

Bibliography

Abi-Antoun, M. and Medvidovic, N. (1999); *Enabling the Refinement of a Software Architecture into a Design*; in: UML99 - The Unified Modeling Language: Beyond the Standard, Second International Conference (edited by France, R. and Rumpe, B.); volume 1723 of *Lecture Notes in Computer Science*; pp. 17–31; Springer, Fort Collins, CO, USA; doi: 10.1007/3-540-46852-8_3. *(cited on pages 43, 64, and 142).*

Abi-Antoun, M.; Aldrich, J.; Garlan, D.; Schmerl, B.; Nahas, N.; and Tseng, T. (2005); *Improving System Dependability by Enforcing Architectural Intent*; in: ICSE 2005 Workshop on Architecting Dependable Systems (WADS 2005). *(cited on page 201).*

Abowd, G.; Allen, R.; and Garlan, D. (1993); *Using style to understand descriptions of software architecture*; SIGSOFT Softw. Eng. Notes; 18(5):9–20; doi:10.1145/167049.167055. Proceedings of SIGSOFT FSE 1993. *(cited on pages 48 and 49).*

Abowd, G. D.; Allen, R.; and Garlan, D. (1995); *Formalizing style to understand descriptions of software architecture*; ACM Trans. Softw. Eng. Methodol.; 4(4):319–364; doi:10.1145/226241.226244. *(cited on pages 48 and 122).*

Acuña, S. T.; de Antonio, A.; Ferré, X.; López, M.; and Maté, L. (2002); *The Software Process: Modelling, Evaluation and Improvement*; in: Handbook of Software Engineering and Knowledge Engineering, Vol. 1 (edited by Chang, S. K.); pp. 193–238; World Scientific; ISBN 981-02-4973-X. *(cited on page 237).*

Agresti, W. W. and Evanco, W. M. (1992); *Projecting Software Defects from Analyzing Ada Designs*; IEEE Trans. Softw. Eng.; 18(11):988–997; doi:10.1109/32.177368. *(cited on page 90).*

Aldrich, J.; Chambers, C.; and Notkin, D. (2002); *ArchJava: connecting software architecture to implementation*; in: Proceedings of the 24th international conference on Software engineering; pp. 187–197; ACM Press; ISBN 1-58113-472-X; doi:10.1145/581339.581365. *(cited on page 201).*

Alexander, C. (1964); Notes on the Synthesis of Form; Harvard University Press, Cambridge, MA, USA; ISBN 0-674-62751-2. *(cited on page 51).*

—— (1979); The Timeless Way of Building; Oxford University Press; ISBN 0-19-502402-8. *(cited on page 51).*

—— (1999); *The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World*; IEEE Softw.; 16(5):71–82; doi: 10.1109/52.795104. *(cited on page 51).*

Alexander, C.; Ishikawa, S.; and Silverstein, M. (1977); A pattern language : towns, buildings, construction; volume 2 of *Center for Environmental Structure*; Oxford Univ. Press, New York; ISBN 0-19-501919-9. *(cited on pages 50, 51, and 107).*

Allen, R. (1995); *Formalism and Informalism in Architectural Style: A Case Study*; in: Proceedings of the First International Workshop on Architectures for Software Systems (edited by Garlan, D.); volume 95-151 of *CMU-CS Technical Reports*; Carnegie Mellon Univ., Pittsburgh, PA, USA. *(cited on page 48).*

—— (1996); *HLA: a standards effort as architectural style*; in: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops; pp. 130–133; ACM Press, New York, NY, USA; ISBN 0-89791-867-3; doi: 10.1145/243327.243626. *(cited on pages 119 and 194).*

Allen, R. and Garlan, D. (1997); *A formal basis for architectural connection*; ACM Transactions on Software Engineering and Methodology (TOSEM); 6(3):213–249; doi:10.1145/258077.258078. *(cited on pages 61 and 106).*

Almeida, J. P.; Dijkman, R.; van Sinderen, M.; and Pires, L. F. (2004); *On the Notion of Abstract Platform in MDA Development*; in: EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04); pp. 253–263; IEEE Computer Society Press; ISBN 0-7695-2214-9; doi:10.1109/EDOC.2004.16. *(cited on pages 33, 34, 83, and 105).*

Alpert, S. R.; Brown, K.; and Woolf, B. (1998); The design patterns Smalltalk companion; Addison-Wesley Longman; ISBN 0-201-18462-1. *(cited on page 55).*

Antoniol, G.; Casazza, G.; Penta, M. D.; and Fiutem, R. (2001); *Object-oriented design patterns recovery*; J. Syst. Softw.; 59(2):181–196; doi:10.1016/S0164-1212(01)00061-9. *(cited on page 53).*

Atzeni, P.; Ceri, S.; Paraboschi, S.; and Torlone, R. (1999); Database Systems – Concepts, Languages and Architectures; McGraw-Hill; ISBN 0077095006. *(cited on pages 46 and 126).*

Avizienis, A. and Chen, L. (1977); *On the implementation of N-version programming for software fault tolerance during execution*; in: Proc. IEEE International Computer Software & Applications Conference (COMPSAC 77); pp. 149–155. *(cited on page 20).*

Avizienis, A.; Laprie, J.-C.; and Randell, B. (2001); Fundamental Concepts of Dependability; Technical report 739; Department of Computing Science, University of Newcastle upon Tyne; URL `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/739.pdf`. *(cited on pages 19 and 21).*

Babar, M. A. and Gorton, I. (2004); *Comparison of Scenario-Based*

Software Architecture Evaluation Methods; in: APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference; pp. 600–607; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-2245-9; doi:10.1109/APSEC.2004.38. *(cited on page 28).*

Bahsoon, R.; Emmerich, W.; and Macke, J. (2005); *Using real options to select stable middleware-induced software architectures*; Software, IEE Proceedings; 152(4):167–186; doi:10.1049/ip-sen:20045059. *(cited on pages 209 and 234).*

Baker, S. and Dobson, S. (2005); *Comparing Service-Oriented and Distributed Object Architectures.*; in: On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings, Part I (edited by Meersman, R.; Tari, Z.; Hacid, M.-S.; Mylopoulos, J.; Pernici, B.; Babaoglu, Ö.; Jacobsen, H.-A.; Loyall, J. P.; Kifer, M.; and Spaccapietra, S.); volume 3760 of *Lecture Notes in Computer Science*; pp. 631–645; Springer; ISBN 3-540-29736-7; doi:10.1007/11575771_40. *(cited on page 194).*

Balasubramaniam, D.; Morrison, R.; Kirby, G. N. C.; Mickan, K.; and Norcross, S. (2004); ArchWare ADL Release 1 User Reference Manual; Technical report D4.3; ArchWare Project; URL `http://www.dcs.st-and.ac.uk/research/publications/BMK+04.php`. *(cited on pages 106 and 128).*

Baresi, L.; Heckel, R.; Thöne, S.; and Varro, D. (2003); *Modeling and validation of service-oriented architectures: application vs. style*; SIGSOFT Softw. Eng. Notes; 28(5):68–77; doi:10.1145/949952.940082. *(cited on page 117).*

Baresi, L.; Heckel, R.; Thöne, S.; and Varró, D. (2004); *Style-Based Refinement of Dynamic Software Architectures*; in: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04); p. 155; IEEE Computer Society; ISBN 0-7695-2172-X. *(cited on pages 105 and 209).*

—— (2006); *Style-based modeling and refinement of service-oriented architectures*; Software and Systems Modeling; 5(2):187–207; doi:10.1007/s10270-006-0001-4. *(cited on pages 194, 209, and 210).*

Basili, V.; Caldiera, G.; and Rombach, D. (1994); *Goal Question Metric Paradigm*; in: Encyclopedia of Software Engineering, Volume I (edited by Marciniak, J. J.); pp. 528–532; John Wiley & Sons; ISBN 0-471-54001-3. *(cited on pages 26 and 74).*

Basili, V. R.; Briand, L. C.; and Melo, W. L. (1996); *How reuse influences productivity in object-oriented systems*; Commun. ACM; 39(10):104–116; doi:10.1145/236156.236184. *(cited on page 90).*

Bass, L.; Clements, P.; and Kazman, R. (1998); Software architecture in practice; Addison-Wesley Longman; 1. edition; ISBN 0-201-19930-0. *(cited on pages 21, 24, and 40).*

—— (2003); Software architecture in practice; Addison-Wesley; 2. edition; ISBN 0-321-15495-9. *(cited on pages 23, 25, and 213).*

Becker-Pechau, P.; Karstens, B.; and Lilienthal, C. (2006); *Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln.*; in: Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik, 28.-31.3.2006 in Leipzig (edited by Biel, B.; Book, M.; and Gruhn, V.); volume 79 of *LNI*; pp. 27–37; GI; ISBN 3-88579-173-0. *(cited on page 47).*

Bengtsson, P.; Lassing, N. H.; Bosch, J.; and van Vliet, H. (2004); *Architecture-level modifiability analysis (ALMA)*; Journal of Systems and Software; 69(1-2):129–147. *(cited on page 28).*

Biemans, F. P. M.; Lankhorst, M. M.; Teeuw, W. B.; and van de Wetering, R. G. (2001); *Dealing with the Complexity of Business Systems Architecting*; Systems Engineering; 4(2):118–133. *(cited on page 24).*

Bischofberger, W. R.; Kühl, J.; and Löffler, S. (2004); *Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking*; in: Software Architecture, First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004, Proceedings (edited by Oquendo, F.; Warboys, B.; and Morrison, R.); volume 3047 of *Lecture Notes in Computer Science*; pp. 1–9; Springer; ISBN 3-540-22000-3. *(cited on page 47).*

Boehm, B. (2005); Value-Based Software Engineering: Overview and Agenda; Technical report USC-CSE-2005-504; University of Southern California, Computser Science Department. *(cited on page 233).*

Bondarev, E.; Chaudron, M. R.; Zhang, J.; and Klomp, A. (2006); Quality-Oriented Design Space Exploration for Component-Based Architectures; Computer Science Report 2006-09; TU Eindhoven Department of Mathematics and Computer Science. *(cited on page 84).*

Bornhold, J. (2006); Migration eines regionalen Wirtschaftsinformationssystems; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 58, 140, 179, 186, 223, 224, and 225).*

Bostrom, R. P. and Heinen, J. S. (1977); *MIS problems and failures: A socio-technical perspective*; MIS Quarterly; 1(3):17–32. *(cited on page 33).*

Bratthall, L. and Wohlin, C. (2000); *Understanding Some Software Quality Aspects from Architecture and Design Models*; in: IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension; p. 27; IEEE Computer Society; ISBN 0-7695-0656-9. *(cited on page 23).*

Brooks, Jr., F. P. (1995); The mythical man-month (anniversary ed.); Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA; ISBN 0-201-83595-9. *(cited on pages 24 and 25).*

Brown, W. J.; Malveau, R. C.; McCormick, M.; and Mowbray, T. J. (1998); AntiPatterns: Refactoring Software, Architectures, and Projects

in Crisis; John Wiley; ISBN 0-471-19713-0. *(cited on page 57).*

Browne, J. C.; Werth, J.; and Lee, T. (1990); *Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment*; IEEE Trans. Softw. Eng.; 16(2):111–120; doi:10.1109/32.44375. *(cited on page 90).*

Budgen, D.; Brereton, P.; and Turner, M. (2004); *Codifying a Service Architectural Style*; in: 28th Annual International Computer Software and Applications Conference (COMPSAC'04); pp. 16–22; IEEE Computer Society Press. *(cited on page 194).*

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; and Stal, M. (1996); Pattern-Oriented Software Architecture: A System of Patterns; John Wiley & Sons; ISBN 0-471-95869-7. *(cited on pages 50, 51, 55, 102, and 103).*

Cantone, G. and Donzelli, P. (1999); *Goal-oriented software measurement models*; in: Project control for Software quality, Proceedings of the combined 10th European Software Control and Metrics Conference and the 2nd SCOPE conference on Software Product Evaluation (edited by Kusters, R.; Cowderoy, A.; Heemstra, F.; and Veenendaal, E.); pp. 355–364; Shaker Publishing BV, Maastricht, The Netherlands; ISBN 90-423-0075-2. *(cited on page 26).*

Caporuscio, M.; Inverardi, P.; and Pelliccione, P. (2004); *Formal Analysis of Architectural Patterns*; in: EWSA (edited by Oquendo, F.; Warboys, B.; and Morrison, R.); volume 3047 of *Lecture Notes in Computer Science*; pp. 10–24; Springer; ISBN 3-540-22000-3. *(cited on pages 50 and 52).*

Card, D. N.; Church, V. E.; and Agresti, W. W. (1986); *An empirical study of software design practices*; IEEE Trans. Softw. Eng.; 12(2):264–271. *(cited on page 90).*

Carnegie Mellon University (2006); *The Acme Architectural Description Language and Design Environment*. `http://acme.able.cs.cmu.edu/`. *(cited on page 209).*

Cheesman, J. and Daniels, J. (2000); UML components: a simple process for specifying component-based software; Addison-Wesley Longman; ISBN 0-201-70851-5. *(cited on page 40).*

Chen, D.-J. and Lee, P. J. (1993); *On the study of software reuse using reusable C++ components*; J. Syst. Softw.; 20(1):19–36; doi:10.1016/0164-1212(93)90046-Z. *(cited on page 90).*

Chrissis, M. B.; Konrad, M.; and Shrum, S. (2003); CMMI : Guidelines for Process Integration and Product Improvement; Addison Wesley; ISBN 0-321-15496-7. *(cited on page 94).*

Clements, P.; Garlan, D.; Bass, L.; Stafford, J.; Nord, R.; Ivers, J.; and Little, R. (2002); Documenting Software Architectures: Views and Beyond; Pearson Education; ISBN 0201703726. *(cited on pages 38, 44, 99, and 108).*

Clements, P. C. (1996); *A Survey of Architecture Description Languages*;

in: IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design; p. 16; IEEE Computer Society, Washington, DC, USA; ISBN 0-8186-7361-3. *(cited on page 57).*

Compare, D.; Inverardi, P.; and Wolf, A. (1999); *Uncovering architectural mismatch in component behavior*; Science of Computer Programming; 33(2):101–131; doi:10.1016/S0167-6423(98)00006-9. *(cited on page 101).*

Conway, M. E. (1968); *How Do Committees Invent?*; Datamation; 14(4):28–31. *(cited on pages 40 and 106).*

Coplien, J. (1996a); *Broadening beyond objects to patterns and to other paradigms*; ACM Comput. Surv.; 28(4es):152; doi:10.1145/242224.242418. *(cited on page 54).*

—— (1996b); Software Patterns; SIGS, New York; ISBN 1-88484-250-X. *(cited on page 101).*

Dashofy, E.; Garlan, D.; van der Hoek, A.; and Schmerl, B. (2006); *xArch*. `http://www.isr.uci.edu/architecture/xarch/`. *(cited on page 62).*

Dashofy, E. M.; van der Hoek, A.; and Taylor, R. N. (2005); *A comprehensive approach for the development of modular software architecture description languages*; ACM Trans. Softw. Eng. Methodol.; 14(2):199–245; doi:10.1145/1061254.1061258. *(cited on pages 62 and 106).*

Davis, L.; Gamble, R. F.; and Payton, J. (2002); *The impact of component architectures on interoperability*; J. Syst. Softw.; 61(1):31–45; doi:10.1016/S0164-1212(01)00112-1. *(cited on page 101).*

Davis, T. (1993); *The reuse capability model: a basis for improving an organization's reuse capability*; in: Advances in Software Reuse; pp. 126–133; ISBN 0-8186-3130-9; doi:10.1109/ASR.1993.291710. *(cited on page 90).*

Demeyer, S.; Meijler, T. D.; Nierstrasz, O.; and Steyaert, P. (1997); *Design guidelines for tailorable frameworks*; Commun. ACM; 40(10):60–64; doi:10.1145/262793.262805. *(cited on page 91).*

Demeyer, S.; Ducasse, S.; and Nierstrasz, O. M. (2003); Object-oriented reengineering patterns; Morgan Kaufman Publ., San Francisco; ISBN 1-558-60639-4. *(cited on page 55).*

van Dijk, H. W.; Graaf, B.; and Boerman, R. (2005); *On the Systematic Conformance Check of Software Artefacts*; in: Proceedings of EWSA 2005, Second European Workshop on Software Architecture; pp. 204–221; Lecture Notes in Computer Science; Spinger-Verlag, Pisa. *(cited on page 201).*

Dobrica, L. and Niemelä, E. (2002); *A survey on software architecture analysis methods*; IEEE Trans. Softw. Eng.; 28(7):638–653; doi:10.1109/TSE.2002.1019479. *(cited on page 28).*

Dueñas, J. C.; de Oliveira, W. L.; and de la Puente, J. A. (1998); *A Software Architecture Evaluation Model*; in: ESPRIT ARES Workshop (edited by van der Linden, F.); volume 1429 of *Lecture Notes*

in Computer Science; pp. 148–157; Springer; ISBN 3-540-64916-6; doi: 10.1007/3-540-68383-6_22. *(cited on page 24).*

Dyson, P. (1997); Patterns for Abstract Design; PhD thesis; University of Essex. *(cited on page 55).*

Eden, A. H.; Kazman, R.; and Hirshfeld, Y. (2004); Abstraction Strata in Software Design; Technical report CSM-411; Department of Computer Science, University of Essex. *(cited on pages 50 and 52).*

Egyed, A.; Mehta, N. R.; and Medvidovic, N. (2000); *Software Connectors and Refinement in Family Architectures*; in: IW-SAPF-3: Proceedings of the International Workshop on Software A rchitectures for Product Families; pp. 96–106; Springer-Verlag, London, UK; ISBN 3-540-41480-0; doi:10.1007/b78902. *(cited on page 43).*

Eiben, A. (2008); Strategien für die Migration von Client/Server-Architekturen zu Service-orientierten Architekturen; Masterarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 179, 191, 223, 225, and 254).*

Eisenführ, F. and Weber, M. (2002); Rationales Entscheiden; Springer; 4. edition; ISBN 3540440232. *(cited on page 233).*

Emmerich, W. (2000); *Software engineering and middleware: a roadmap*; in: ICSE '00: Proceedings of the Conference on The Future of Software Engineering; pp. 117–129; ACM Press, New York, NY, USA; ISBN 1-58113-253-0; doi:10.1145/336512.336542. *(cited on page 82).*

Engels, G.; Hess, A.; Humm, B.; et al. (2008); Quasar Enterprise; dPunkt Verlag, Heidelberg; ISBN 978-3898645065. *(cited on pages 46 and 47).*

Englisch, J. (2006); Ein Repository für Architektur- und Entwurfsmuster in Eclipse; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 55, 56, 210, 221, and 253).*

Eusgeld, I.; Freiling, F.; and Reussner, R. (editors) (2008); Dependability Metrics; volume 4909 of *Lecture Notes in Computer Science*; Springer; ISBN 978-3-540-68946-1; doi:10.1007/978-3-540-68947-8. *(cited on page 244).*

Floyd, C. (1997); *Autooperationale Form und situiertes Handeln*; in: Cognitio Humana : Dynamik des Wissens und der Werte (edited by Hubig, C.); pp. 237–252; Akademie-Verlag; ISBN 3-05-003109-3. *(cited on page 173).*

Fowler, M. (1997); Analysis patterns: reusable objects models; Addison-Wesley Longman; ISBN 0-201-89542-0. *(cited on page 55).*

Frakes, W. and Terry, C. (1996); *Software reuse: metrics and models*; ACM Comput. Surv.; 28(2):415–435; doi:10.1145/234528.234531. *(cited on pages 88 and 90).*

France, R. B.; Kim, D.-K.; Ghosh, S.; and Song, E. (2004); *A UML-Based Pattern Specification Technique*; IEEE Trans. Softw. Eng.; 30(3):193–206;

doi:10.1109/TSE.2004.1271174. *(cited on page 149).*

Frolund, S. and Koisten, J. (1998); QML: A Language for Quality of Service Specification; HP Labs Technical Report HPL-98-10; Hewlett-Packard Laboratories; URL `http://www.hpl.hp.com/techreports/98/HPL-98-10.html`. *(cited on page 137).*

Fuentes, L.; Troya, J. M.; and Vallecillo, A. (2002); *Using UML Profiles for Documenting Web-Based Application Frameworks*; Ann. Softw. Eng.; 13(1-4):249–264. *(cited on pages 91 and 210).*

Gaffney, J. E. and Durek, T. A. (1989); *Software reuse: key to enhanced productivity: some quantitative models*; Inf. Softw. Technol.; 31(5):258–267; doi:10.1016/0950-5849(89)90005-0. *(cited on page 90).*

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. (1995); Design patterns: elements of reusable object-oriented software; Addison-Wesley Longman Publishing Co., Inc.; ISBN 0-201-63361-2. *(cited on pages 50, 51, 54, 55, 99, and 101).*

Garlan, D. (1995); *What is Style?*; in: Software architectures (edited by Garlan, D.); volume 106 of *Dagstuhl-Seminar-Report*; Saarbrücken, Germany. Proceedings of the Dagstuhl Workshop on Software Architecture. *(cited on pages 48, 49, 53, 99, 106, and 122).*

Garlan, D. and Shaw, M. (1993); *An Introduction to Software Architecture*; in: Advances in Software Engineering and Knowledge Engineering (edited by Ambriola, V. and Tortora, G.); pp. 1–39; World Scientific Publishing Company, Singapore. *(cited on page 104).*

Garlan, D.; Allen, R.; and Ockerbloom, J. (1994); *Exploiting Style in Architectural Design Environments*; in: Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering; ACM Press; doi:10.1145/195274.195404. *(cited on page 48).*

—— (1995); *Architectural mismatch or why it's hard to build systems out of existing parts*; in: Proceedings of the 17th international conference on Software engineering; pp. 179–185; ACM Press; ISBN 0-89791-708-1; doi:10.1145/225014.225031. *(cited on page 13).*

Garlan, D.; Monroe, R.; and Wile, D. (1997); *Acme: an architecture description interchange language*; in: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research; pp. 7–; IBM Press. *(cited on pages 61, 101, and 106).*

Garlan, D.; Monroe, R. T.; and Wile, D. (2000); *Acme: architectural description of component-based systems*; in: Foundations of component-based systems; pp. 47–67; Cambridge University Press, New York, NY, USA; ISBN 0-521-77164-1. *(cited on pages 12, 61, 128, and 142).*

Garlan, D.; Cheng, S.-W.; and Kompanek, A. J. (2002); *Reconciling the needs of architectural description with object-modeling notations*; Sci. Comput. Program.; 44(1):23–49; doi:10.1016/S0167-6423(02)00031-X. *(cited on pages 48, 63, 64, 142, 144, and 211).*

Geppert, B. and Roessler, F. (2001); *Combining Product Line Engineering*

with Options Thinking; in: PLEES '01. *(cited on page 233).*

Ghezzi, C.; Jazayeri, M.; and Mandrioli, D. (2002); Fundamentals of Software Engineering; Prentice Hall PTR, Upper Saddle River, NJ, USA; ISBN 0133056996. *(cited on page 22).*

Giesecke, S. (2006a); *Middleware-induced Styles for Enterprise Application Integration*; in: Proc. 10th European Conference on Software Maintenance and Reengineering (CSMR06), Bari, Italy; pp. 334–340; IEEE Comp. Soc.; ISBN 0-7695-2536-9; doi:10.1109/CSMR.2006.33. *(cited on page 120).*

—— (2006b); *Fundamental Concepts of Dependability*; in: Dependability Engineering (edited by Hasselbring, W. and Giesecke, S.); pp. 11–35; Trustworthy Software Systems; GITO-Verlag; ISBN 3-936771-56-1. *(cited on page 19).*

Giesecke, S. and Bornhold, J. (2006); *Style-based Architectural Analysis for Migrating a Web-based Regional Trade Information System*; in: First International Workshop on Web Maintenance and Reengineering (WMR 2006) (edited by Trentini, A.; Marchetto, A.; and Bellettini, C.); volume 193 of *CEUR Workshop Proceedings*; pp. 15–23; URL `http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-193/paper2.pdf`. *(cited on page 13).*

Giesecke, S. and Hasselbring, W. (2006); *A Taxonomy of Architectural Style Usages*; in: Pattern Languages of Programming (PLoP'06), Portland, OR, USA; URL `http://hillside.net/plop/2006/Papers/Library/PLoP-StyleUsage.pdf`. *(cited on pages 51 and 221).*

Giesecke, S.; Hasselbring, W.; and Matevska, J. (2006); *Extending the ANSI/IEEE Standard 1471 for the Representation of Architectural Rationale*; in: Nordic Workshop on the Unified Modeling Language and Software Modeling (NWUML'06), Grimstad, Norway; URL `http://grimstad.hia.no/nwuml06/Papers/Giesecke_Matevska_Hasselbring.pdf`. *(cited on pages 96 and 222).*

Giesecke, S.; Bornhold, J.; and Hasselbring, W. (2007a); *Middleware-induced Architectural Style Modelling for Architecture Exploration*; in: Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), January 2007, Mumbai, India; IEEE Computer Society Press; ISBN 0-7695-2744-2; doi:10.1109/WICSA.2007.29. *(cited on pages 120 and 150).*

Giesecke, S.; Hasselbring, W.; and Riebisch, M. (2007b); *Classifying Architectural Constraints as a basis for Software Quality Assessment*; Advanced Engineering Informatics; 21(2):169–179; doi:10.1016/j.aei.2006.11.002. Special Issue on Ontology and Epistemology of Systems and Software Engineering. *(cited on pages 52 and 220).*

Giesecke, S.; Marwede, F.; Rohr, M.; and Hasselbring, W. (2007c); *A Style-based Architecture Modelling Approach for UML 2 Component Diagrams*; in: Proceedings of the 11th IASTED International Conference Software Engineering and Applications (SEA 2007), Cambridge, MA, USA; pp. 530–538; ACTA Press, Anaheim, CA, USA; ISBN 978-0-88986-705-5.

(cited on page 223).

Goedicke, M. and Zdun, U. (2001); *A Key Technology Evaluation Case Study: Applying a New Middleware Architecture on the Enterprise Scale*; in: EDO 2000 (edited by Emmerich, W. and Tai, S.); volume 1999 of *Lecture Notes in Computer Science*; pp. 8–26; Springer; ISBN 3-540-41792-3; doi:10.1007/3-540-45254-0_3. *(cited on page 81).*

Gottschalk, M. (2007); Abbildung von Architekturbeschreibungen auf Implementierungsartefakte zur Überprüfung von Middleware-orientierten Architekturregeln; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 129, 201, 223, 224, and 225).*

Goulao, M. and e Abreu, F. (2003); *Bridging the gap between Acme and UML 2.0 for CBD*; in: Proceedings of Specification and Verification of Component-Based Systems (SAVCBS 2003), Workshop at ESEC/FSE 2003; URL `http://archives.cs.iastate.edu/documents/disk0/00/00/03/11/00000311-00/SAVCBS03.pdf`. Technical Report #03-11, Department of Computer Science, Iowa State University. *(cited on page 142).*

Grosskurth, A. and Godfrey, M. W. (2005); *A Reference Architecture for Web Browsers*; in: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05); pp. 661–664; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-2368-4; doi:10.1109/ICSM.2005.13. *(cited on page 109).*

Grunske, L. (2006); *Bewertungstechniken - eine allgemeine Übersicht*; in: Handbuch der Software-Architektur (edited by Reussner, R. and Hasselbring, W.); pp. 277–293; dPunkt Verlag; ISBN 3-89864-372-7. *(cited on page 28).*

Hasselbring, W. (2000); *Information system integration*; Commun. ACM; 43(6):32–38; doi:10.1145/336460.336472. *(cited on pages 79 and 122).*

Hasselbring, W. and Giesecke, S. (editors) (2006); Research Methods in Software Engineering; volume 1 of *Trustworthy Software Systems*; GITO-Verlag; ISBN 3-936771-57-x. *(cited on page 69).*

Hasselbring, W. and Reussner, R. (2006); *Toward Trustworthy Software Systems*; Computer; 39(4):91–92; doi:10.1109/MC.2006.142. *(cited on page 24).*

Hilbrands, R. (2006); Vergleich von Architekturbeschreibungssprachen; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 58 and 222).*

Hilliard, R. (1999); *Views and viewpoints in software systems architecture*; in: Software architecture : TC2 First Working IFIP Conference on Software Architecture (WICSA1) (edited by Donohoe, P.); volume 12 of *IFIP*; Kluwer; ISBN 0-7923-8453-9. *(cited on page 39).*

Hirsch, D.; Kramer, J.; Magee, J.; and Uchitel, S. (2006); *Modes for*

Software Architectures; in: 3rd European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA 06); volume 4344 of *Lecture Notes in Computer Science*; pp. 113–126; ISBN 978-3-540-69271-3; doi:10.1007/11966104_9. *(cited on page 45).*

Hofmeister, C.; Nord, R.; and Soni, D. (2000); Applied Software Architecture; Object Technology Series; Addison-Wesley; ISBN 0-201-32571-3. *(cited on page 38).*

Huang, H.; Zhang, S.; Cao, J.; and Duan, Y. (2005); *A practical pattern recovery approach based on both structural and behavioral analysis*; J. Syst. Softw.; 75(1-2):69–87. *(cited on page 53).*

Hudaib, A. and Montangero, C. (2002); *A UML Profile to Support the Formal Presentation of Software Architecture*; in: COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment; pp. 217–223; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-1727-7. *(cited on page 210).*

Humm, B.; Voss, M.; and Hess, A. (2006); *Regeln für serviceorientierte Architekturen hoher Qualität*; Informatik Spektrum; 29(6):395–411; doi:10.1007/s00287-006-0099-3. *(cited on page 32).*

Inverardi, P.; Muccini, H.; and Pelliccione, P. (2005); *DUALLY: Putting in Synergy UML 2.0 and ADLs*; in: WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05); pp. 251–252; IEEE Computer Society; ISBN 0-7695-2548-2; doi:10.1109/WICSA.2005.28. *(cited on pages 58 and 211).*

ISO (2006); Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard 1471-2000, ISO/IEC Standard 42010 (formerly ISO/IEC DIS 25961). *(cited on pages 12, 29, 30, 34, 35, 71, 222, 227, and 253).*

ISO/IEC (1999); Information technology – Open Distributed Processing – Interface Definition Language. ISO/IEC Standard 14750:1999. *(cited on page 101).*

ISO/IEC 14598-1 (1999); ISO/IEC 14598-1: Information technology – Software product evaluation – Part 1: General overview; ISO/IEC. Published standard. *(cited on page 21).*

ISO/IEC 9126-1 (2001); ISO/IEC 9126-1: Software Engineering – Product Quality – Part 1: Quality Model; ISO/IEC JTC 1/SC 7. Published standard. *(cited on pages 20, 21, 22, and 23).*

Ivers, J.; Clements, P.; Garlan, D.; Nord, R.; Schmerl, B.; and Silva, J. R. O. (2004); Documenting Component and Connector Views with UML 2.0; Technical report CMU/SEI-2004-TR-008; School of Computer Science, Carnegie Mellon University; URL `http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tr008.pdf`. *(cited on page 64).*

Jansen, A. and Bosch, J. (2006); *Software Architecture as a Set of Architectural Design Decisions*; in: 5th Working IEEE/IFIP Conference on Soft-

ware Architecture (WICSA'05) (edited by Nord, R.); pp. 109–120; IEEE Computer Society Press; ISBN 0-7695-2548-2; doi:10.1109/WICSA.2005.61. *(cited on page 85).*

Jazayeri, M.; Ran, A.; and van der Linden, F. (2000); Software architecture for product families: principles and practice; Addison-Wesley, Boston, USA; ISBN 0-201-69967-2. *(cited on pages 39 and 120).*

JBoss Labs (2006); *Hibernate*. `http://www.hibernate.org/`. *(cited on page 185).*

Jones, C. (1994); *Economics of software reuse*; Computer; 27(7):106–107; doi:10.1109/2.299437. *(cited on pages 12 and 88).*

Jung, H.-W.; Kim, S.-G.; and Chung, C.-S. (2004); *Measuring Software Product Quality: A Survey of ISO/IEC 9126*; IEEE Softw.; 21(5):88–92; doi:10.1109/MS.2004.1331309. *(cited on page 25).*

Jung, R. (2007); Process-driven Support Environment for the MidArch-Method; Individuelles projekt; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 197 and 225).*

—— (2008); Ontology-based Metadata for MidArch Styles; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on page 223).*

Kacem, M. H.; Kacem, A. H.; Jmaiel, M.; and Drira, K. (2006); *Describing dynamic software architectures using an extended UML model*; in: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing; pp. 1245–1249; ACM Press, New York, NY, USA; ISBN 1-59593-108-2; doi:10.1145/1141277.1141569. *(cited on page 211).*

Kandé, M. M. and Strohmeier, A. (2000); *Towards a UML Profile for Software Architecture Descriptions*; in: UML 2000 – The Unified Modeling Language. Advancing the Standard: Third International Conference; volume 1939 of *Lecture Notes in Computer Science*; pp. 513–527; Springer; doi:10.1007/3-540-40011-7_38. *(cited on pages 141 and 210).*

Kandé, M. M.; Crettaz, V.; Strohmeier, A.; and Sendall, S. (2002); *Bridging the gap between IEEE 1471, an architecture description language, and UML.*; Software and System Modeling; 1(2):113–129; doi:10.1007/s10270-002-0010-x. *(cited on page 227).*

Kazman, R.; Bass, L.; Webb, M.; and Abowd, G. (1994); *SAAM: a method for analyzing the properties of software architectures*; in: ICSE '94: Proceedings of the 16th international conference on Software engineering; pp. 81–90; IEEE Computer Society Press, Los Alamitos, CA, USA; ISBN 0-8186-5855-X. *(cited on pages 28, 38, and 41).*

Kazman, R.; Klein, M.; and Clements, P. (2000); ATAM: A Method for Architecture Evaluation; Technical report CMU/SEI-2000-TR-004; Software Engineering Institute, Carnegie Mellon University. *(cited on*

page 28).

Kazman, R.; Asundi, J.; and Klein, M. (2001); *Quantifying the costs and benefits of architectural decisions*; in: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering; pp. 297–306; IEEE Computer Society; ISBN 0-7695-1050-7. *(cited on page 28)*.

—— (2002); Making Architecture Design Decisions: An Economic Approach; Technical report CMU/SEI-2002-TR-035; Software Engineering Institute, Carnegie Mellon University. *(cited on page 28)*.

KBSt (2008); *V-Modell XT*; URL `http://www.v-modell-xt.de/`. Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung im Bundesministerium des Innern. *(cited on page 177)*.

Keshav, R. and Gamble, R. (1998); *Towards a taxonomy of architecture integration strategies*; in: ISAW '98: Proceedings of the third international workshop on Software architecture; pp. 89–92; ACM Press, New York, NY, USA; ISBN 1-58113-081-3; doi:10.1145/288408.288431. *(cited on page 107)*.

Kirchner, M. and Jain, P. (2004); Pattern-oriented software architecture, vol. 3: Patterns for resource management; Wiley series in software design patterns; Wiley, Chichester; ISBN 0-470-84525-2. *(cited on page 103)*.

Klein, M. and Kazman, R. (1999); Attribute-Based Architectural Styles; Technical report CMU/SEI-99-TR-022; Software Engineering Institute, Carnegie Mellon University. *(cited on page 107)*.

Klein, M.; Kazman, R.; and Nord, R. (2000); *A BASis (or ABASs) for Reasoning About Software Architectures*. Software Engineering Institute. *(cited on page 107)*.

Kollman, R.; Selonen, P.; Stroulia, E.; Systä, T.; and Zündorf, A. (2002); *A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering*; in: WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02); p. 22; IEEE Computer Society, Washington, DC, USA. *(cited on page 140)*.

Koltun, P. and Hudson, A. (1991); *A reuse maturity model*; in: Fourth Annual Workshop on Software Reuse (edited by Latour, L.); pp. 1–4. *(cited on page 90)*.

Koning, H. and van Vliet, H. (2006); *A Method for Defining IEEE Std 1471 Viewpoints*; Journal of Systems and Software; 79(1):120–131. *(cited on page 38)*.

Kruchten, P. (1995); *The 4+1 View Model of Architecture*; IEEE Softw.; 12(6):42–50; doi:10.1109/52.469759. *(cited on pages 38, 39, and 40)*.

—— (2005); *Casting Software Design in the Function-Behavior-Structure Framework*; IEEE Software; 22(2):52–58. *(cited on page 27)*.

Krueger, C. W. (1992); *Software reuse*; ACM Comput. Surv.; 24(2):131–183; doi:10.1145/130844.130856. *(cited on page 101)*.

Lim, W. C. (1994); *Effects of Reuse on Quality, Productivity, and Economics*;

IEEE Softw.; 11(5):23–30; doi:10.1109/52.311048. *(cited on page 90).*

Liu, A. and Gorton, I. (2003); *Accelerating COTS Middleware Acquisition: The i-Mate Process*; IEEE Softw.; 20(2):72–79; doi:10.1109/MS.2003.1184171. *(cited on pages 212 and 215).*

Liu, Y. and Gorton, I. (2004); *An Empirical Evaluation of Architectural Alternatives for J2EE and Web Services*; in: APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04); pp. 10–17; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-2245-9; doi:10.1109/APSEC.2004.25. *(cited on page 212).*

Liu, Y.; Gorton, I.; Bass, L.; Hoan, C.; and Abanmi, S. (2006); *MEMS: A Method for Evaluating Middleware Architectures*; in: Second International Conference on the Quality of Software Architectures (QoSA 2006); volume 4214 of *Lecture Notes in Computer Science*; pp. 9–26; ISBN 978-3-540-48819-4; doi:10.1007/11921998_6. *(cited on pages 12, 78, 80, 212, 213, 214, and 215).*

Lorenz, D. H. (1997); *Tiling design patterns. A case study using the interpreter pattern*; in: OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Obje ct-oriented programming, systems, languages, and applications; pp. 206–217; ACM Press, New York, NY, USA; ISBN 0-89791-908-4; doi:10.1145/263698.263737. *(cited on page 55).*

Losavio, F.; Chirinos, L.; Lévy, N.; and Ramdane-Cherif, A. (2003); *Quality Characteristics for Software Architecture*; Journal of Object Technology; 2(2):133–150; URL `http://www.jot.fm/issues/issue_2003_03/article2`. *(cited on page 23).*

Luckham, D. C.; Kenney, J. J.; Augustin, L. M.; Vera, J.; Bryan, D.; and Mann, W. (1995); *Specification and Analysis of System Architecture Using Rapide*; IEEE Trans. Softw. Eng.; 21(4):336–355; doi:10.1109/32.385971. *(cited on page 60).*

Magee, J.; Dulay, N.; Eisenbach, S.; and Kramer, J. (1995); *Specifying Distributed Software Architectures*; in: Proceedings of 5th European Software Engineering Conference (ESEC 95); volume 989 of *Lecture Notes in Computer Science*; pp. 137–153; ISBN 978-3-540-60406-8; doi:10.1007/3-540-60406-5_12. *(cited on page 60).*

Maheshwari, P. and Pang, M. (2005); *Benchmarking message-oriented middleware: TIB/RV versus SonicMQ*; Concurr. Comput. : Pract. Exper.; 17(12):1507–1526; doi:10.1002/cpe.v17:12. *(cited on page 212).*

Mahoney, M. S. (2004); *Finding a History for Software Engineering.*; IEEE Annals of the History of Computing; 26(1):8–19. *(cited on page 29).*

Maier, M. W.; Emery, D.; and Hilliard, R. (2001); *Software Architecture: Introducing IEEE Standard 1471*; Computer; 34(4):107–109; doi:10.1109/2.917550. *(cited on page 29).*

Marwede, F. (2007); Beschreibung von MIDARCH-Regelsätzen mit UML-Profilen; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by

Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 58, 142, 145, and 223).*

May, D. and Taylor, P. (2003); *Knowledge management with patterns*; Commun. ACM; 46(7):94–99; doi:10.1145/792704.792705. *(cited on page 52).*

Mayring, P. (2003); Qualitative Inhaltsanalyse : Grundlagen und Techniken; number 8229 in UTB für Wissenschaft; Beltz, Weinheim; 8. edition; ISBN 3-8252-8229-5. *(cited on page 229).*

Medvidovic, N. (1996); *ADLs and dynamic architecture changes*; in: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops; pp. 24–27; ACM Press, New York, NY, USA; ISBN 0-89791-867-3; doi:10.1145/243327.243340. *(cited on page 45).*

—— (2002); *On the role of middleware in architecture-based software development*; in: Proceedings of the 14th international conference on Software engineering and knowledge engineering; pp. 299–306; ACM Press; ISBN 1-58113-556-4; doi:10.1145/568760.568814. *(cited on pages 105 and 106).*

Medvidovic, N. and Rosenblum, D. S. (2000); *Bridging Heterogeneous Software Interoperability Platforms*; in: Proceedings of the Fourth International Software Architecture Workshop (ISAW-4); pp. 77–83; Limerick, Ireland; URL `http://www.seat.utulsa.edu/papers/MGR00.pdf`. *(cited on page 43).*

Medvidovic, N. and Taylor, R. N. (2000); *A Classification and Comparison Framework for Software Architecture Description Languages*; IEEE Trans. Softw. Eng.; 26(1):70–93; doi:10.1109/32.825767. *(cited on pages 36, 41, 44, 57, 58, 59, 60, and 63).*

Medvidovic, N.; Rosenblum, D. S.; and Taylor, R. N. (1999); *A language and environment for architecture-based software development and evolution*; in: ICSE '99: Proceedings of the 21st international conference on Software engineering; pp. 44–53; IEEE Computer Society Press, Los Alamitos, CA, USA; ISBN 1-58113-074-0. *(cited on pages 72 and 105).*

Medvidovic, N.; Rosenblum, D. S.; Redmiles, D. F.; and Robbins, J. E. (2002); *Modeling software architectures in the Unified Modeling Language*; ACM Trans. Softw. Eng. Methodol.; 11(1):2–57; doi:10.1145/504087.504088. *(cited on pages 141 and 211).*

Medvidovic, N.; Dashofy, E. M.; and Taylor, R. N. (2003); *The Role of Middleware in Architecture-based Software Development*; International Journal of Software Engineering and Knowledge Engineering; 13(4):367–393. *(cited on pages 105, 106, and 209).*

Mehta, N.; Medvidoviç, N.; and Rakiç, M. (2000a); Why Consider Implementation-Level Decisions in Software Architectures?; Technical report USC-CSE-2000-500; University of Southern California, Computer Science Department; URL `http://sunset.usc.edu/publications/`

TECHRPTS/2000/usccse2000-500/usccse2000-500.pdf. *(cited on pages 31, 46, and 105).*

Mehta, N. R. and Medvidovic, N. (2002); Distilling Software Architecture Primitives from Architectural Styles; Technical report USC-CSE-2002-509; University of Southern California, Computer Science Department; URL http://sunset.usc.edu/publications/TECHRPTS/2002/usccse2002-509/usccse2002-509.pdf. *(cited on pages 63, 106, and 211).*

—— (2003); *Composing architectural styles from architectural primitives*; in: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering; pp. 347–350; ACM Press; ISBN 1-58113-743-5; doi:10.1145/940071.940118. *(cited on pages 63, 106, 128, and 211).*

—— (2004); Toward Composition Of Style-Conformant Software Architectures; Technical report USC-CSE-2004-500; University of Southern California, Center for Software Engineering; URL http://sunset.usc.edu/publications/TECHRPTS/2004/usccse2004-500/usccse2004-500.pdf. *(cited on page 211).*

Mehta, N. R.; Medvidovic, N.; and Phadke, S. (2000b); *Towards a taxonomy of software connectors*; in: ICSE '00: Proceedings of the 22nd international conference on Software engineering; pp. 178–187; ACM Press, New York, NY, USA; ISBN 1-58113-206-9; doi:10.1145/337180.337201. *(cited on page 43).*

Meszaros, G. and Doble, J. (1997); *A pattern language for pattern writing*; Pattern languages of program design, vol. 3; pp. 529–574. *(cited on page 55).*

Mohagheghi, P.; Conradi, R.; Killi, O. M.; and Schwarz, H. (2004); *An Empirical Study of Software Reuse vs. Defect-Density and Stability*; in: ICSE '04: Proceedings of the 26th International Conference on Software Engineering; pp. 282–292; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-2163-0. *(cited on page 90).*

Monroe, R. T. (2000a); Capturing Software Architecture Design Expertise with Armani; Technical report CMU-CS-98-163; School of Computer Science, Carnegie Mellon University; URL http://www.cs.cmu.edu/~bmonroe/Armani%20LRM%20TR%20v2.32.pdf. Version 2.3. *(cited on page 139).*

—— (2000b); Capturing Software Architecture Design Expertise with Armani; Technical report CMU-CS-98-163; Carnegie Mellon University, School of Computer Science. Version 2.3. *(cited on pages 61 and 106).*

Monroe, R. T. and Garlan, D. (1996); *Style-Based Reuse for Software Architectures*; in: Proceedings of the 4th International Conference on Software Reuse (ICSR'96); p. 84; IEEE Computer Society; ISBN 0-8186-7301-X. *(cited on page 48).*

Monroe, R. T.; Kompanek, A.; Melton, R.; and Garlan, D. (1997); *Architectural Styles, Design Patterns, and Objects*; IEEE Software; 14(1):43–

52; doi:10.1109/52.566427. *(cited on pages 49, 102, 104, 107, 231, and 232).*

Moore, M.; Kazman, R.; Klein, M.; and Asundi, J. (2003); *Quantifying the value of architecture design decisions: lessons from the field*; in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering; pp. 557–562; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-1877-X. *(cited on page 28).*

Moriconi, M. and Qian, X. (1994); *Correctness and composition of software architectures*; in: SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering; pp. 164–174; ACM Press, New York, NY, USA; ISBN 0-89791-691-3; doi:10.1145/193173.195403. *(cited on pages 41 and 42).*

Morisio, M.; Ezran, M.; and Tully, C. (2002); *Success and Failure Factors in Software Reuse*; IEEE Trans. Softw. Eng.; 28(4):340–357; doi:10.1109/TSE.2002.995420. *(cited on page 90).*

le Métayer, D. (1996); *Software architecture styles as graph grammars*; in: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering; pp. 15–23; ACM Press; ISBN 0-89791-797-9; doi:10.1145/239098.239105. *(cited on page 137).*

Nada, N. and Rine, D. C. (2000); *Three empirical evaluations of a software reuse reference model*; Ann. Softw. Eng.; 10(1-4):225–259. *(cited on page 90).*

Nazareth, D. L. and Rothenberger, M. A. (2004); *Assessing the cost-effectiveness of software reuse: a model for planned reuse*; J. Syst. Softw.; 73(2):245–255; doi:10.1016/S0164-1212(03)00248-6. *(cited on page 90).*

Di Nitto, E. and Rosenblum, D. (1999); *Exploiting ADLs to specify architectural styles induced by middleware infrastructures*; in: Proceedings of the 21st international conference on Software engineering; pp. 13–22; IEEE Computer Society Press; ISBN 1-58113-074-0. *(cited on pages 14, 72, 105, 106, 112, 117, 119, and 209).*

Noble, J. (1998); *Classifying Relationships between Object-Oriented Design Patterns*; in: ASWEC '98: Proceedings of the Australian Software Engineering Conference; p. 98; IEEE Computer Society, Washington, DC, USA; ISBN 0-8186-9187-5. *(cited on page 55).*

Odenthal, G. and Quibeldey-Cirkel, K. (1997); *Using Patterns for Design and Documentation*; in: ECOOP (edited by Aksit, M. and Matsuoka, S.); volume 1241 of *Lecture Notes in Computer Science*; pp. 511–529; Springer; ISBN 3-540-63089-9. *(cited on page 52).*

OMG (2001); *MDA Guide*; URL `http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf`. OMG Document omg/03-06-01. *(cited on pages 34 and 46).*

OMG (2004); Common Object Request Broker Architecture (CORBA) Core Specification 3.0.3; Object Management Group. OMG Specification formal/2004-03-01. *(cited on page 73).*

OMG (2004a); UML Profile for Patterns Specification; Object Management Group. OMG Specification formal/04-02-04. *(cited on pages 149 and 227).*

—— (2005c); UML 2.0 Superstructure Specification; Object Management Group. OMG Specification formal/05-07-04. *(cited on pages 143 and 210).*

—— (2007b); UML 2.1.1 Superstructure Specification; Object Management Group. OMG Specification formal/07-02-03. *(cited on pages 63 and 143).*

Oquendo, F. (2006); *Formally modelling software architectures with the UML 2.0 Profile for π-ADL*; SIGSOFT Softw. Eng. Notes; 31(1):1–13; doi:10.1145/1108768.1108773. *(cited on page 211).*

Oquendo, F.; Warboys, B.; Morrison, R.; Dindeleux, R.; Gallo, F.; Garavel, H.; and Occhipinti, C. (2004); *ArchWare: Architecting Evolvable Software*; in: Software Architecture, First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004, Proceedings (edited by Oquendo, F.; Warboys, B.; and Morrison, R.); volume 3047 of *Lecture Notes in Computer Science*; pp. 257–271; Springer; ISBN 3-540-22000-3. *(cited on page 106).*

Pérez-Martínez, J. E. (2003); *Heavyweight extensions to the UML metamodel to describe the C3 architectural style*; SIGSOFT Softw. Eng. Notes; 28(3):5–5; doi:10.1145/773126.773140. *(cited on page 211).*

Perry, D. E. and Wolf, A. L. (1992); *Foundations for the study of software architecture*; SIGSOFT Softw. Eng. Notes; 17(4):40–52; doi:10.1145/141874.141884. *(cited on pages 29, 49, 52, and 131).*

Philippow, I.; Streitferdt, D.; and Riebisch, M. (2003); *Design Pattern Recovery in Architectures for Supporting Product Line Development and Application*; in: Modelling Variability for Object-Oriented Product Lines (edited by Riebisch, M.; Coplien, J. O.; and Streitferdt, D.); pp. 42–57; BookOnDemand, Norderstedt. *(cited on page 53).*

Ploski, J. and Giesecke, S. (2005); *When Small Outgrows Beautiful – Experiences from a Development Project*; in: Net.ObjectDays 2005; pp. 367–380; tranSIT; ISBN 3-9808628-4-4. *(cited on pages 11, 13, and 91).*

Posch, T.; Birken, K.; and Gerdom, M. (2004); Basiswissen Softwarearchitektur : verstehen, entwerfen, bewerten und dokumentieren; dpunkt-Verlag, Heidelberg; ISBN 3-89864-270-4. *(cited on page 64).*

Prechelt, L.; Unger-Lamprecht, B.; Philippsen, M.; and Tichy, W. F. (2002); *Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance*; IEEE Trans. Softw. Eng.; 28(6):595–606; doi:10.1109/TSE.2002.1010061. *(cited on page 52).*

Prieto-Diaz, R. and Neighbors, J. M. (1986); *Module interconnection languages*; J. Syst. Softw.; 6(4):307–334; doi:10.1016/0164-1212(86)90002-6. *(cited on page 57).*

Raje, R. R. and Chinnasamy, S. (2001); *eLeLePUS – a language for specification of software design patterns*; in: SAC '01: Proceedings of the 2001 ACM symposium on Applied computing; pp. 600–604; ACM Press, New York, NY, USA; ISBN 1-58113-287-5; doi:10.1145/372202.372480.

(cited on pages 50, 52, and 102).

Ran, A. (1998); *Architectural structures and views*; in: Proceedings of the third international workshop on Software architecture; pp. 117–120; ACM Press; ISBN 1-58113-081-3; doi:10.1145/288408.288438. *(cited on page 39).*

—— (1999); *Software isn't built from Lego blocks*; in: Proceedings of the 1999 symposium on Software reusability; pp. 164–169; ACM Press; ISBN 1-58113-101-1; doi:10.1145/303008.303079. *(cited on page 39).*

Ran, S.; Palmer, D.; Brebner, P.; Chen, S.; Gorton, I.; Gosper, J.; Hu, L.; Liu, A.; and Tran, P. (2002); *J2EE Technology Performance Evaluation Methodology*; in: Distributed Objects and Applications 2002 (DOA'02), Proceedings (addendum); pp. 13–16. *(cited on page 212).*

Randell, B. (1975); *System structure for software fault tolerance*; IEEE Transactions on Software Engineering; SE-1(2):220–232. *(cited on page 20).*

Randell, B. and Buxton, J. (editors) (1970); Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels; NATO Scientific Affairs Division, Brussels; URL `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`. *(cited on page 29).*

Rapide Design Team (1994); Guide to the Rapide 1.0 Language Reference Manuals; Technical report; Program Analysis and Verification Group, Computer Systems Lab., Stanford University; URL `http://pavg.stanford.edu/rapide/lrms/overview.ps`. *(cited on pages 60 and 106).*

Reussner, R. H.; Schmidt, H. W.; and Poernomo, I. H. (2003); *Reliability prediction for component-based software architectures*; J. Syst. Softw.; 66(3):241–252; doi:10.1016/S0164-1212(02)00080-8. *(cited on page 153).*

Richter, J.-P. (2005); *Wann liefert eine Serviceorientierte Architektur echten Nutzen?*; in: Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen (edited by Liggesmeyer, P.; Pohl, K.; and Goedicke, M.); volume 64 of *LNI*; pp. 231–242; GI; ISBN 3-88579-393-8. *(cited on page 22).*

Riehle, D. (1997); *Composite design patterns*; SIGPLAN Not.; 32(10):218–228; doi:10.1145/263700.263739. *(cited on page 54).*

Riehle, D. and Züllighoven, H. (1996); *Understanding and Using Patterns in Software Development*; Theory and Practice of Object Systems; 2(1):3–13. *(cited on pages 47 and 104).*

Robinson, H.; Hall, P.; Hovenden, F.; and Rachel, J. (1998); *Postmodern Software Development*; The Computer Journal; 41(6):363–375; doi:10.1093/comjnl/41.6.363. *(cited on page 173).*

Roh, S.; Kim, K.; and Jeon, T. (2004); *Architecture Modeling Language based on UML2.0*; in: 11th Asia-Pacific Software Engineering Conference (APSEC'04); pp. 663–669; doi:10.1109/APSEC.2004.32. *(cited on page 210).*

ROHR, M. (2005); *Example of Empirical Research: N-version Programming*; in: Research Methods in Software Engineering : A Seminar Reader (edited by GIESECKE, S. and HASSELBRING, W.); Carl von Ossietzky University, Oldenburg, Germany. *(cited on page 20).*

ROZENBERG, G. and SALOMAA, A. (editors) (2004); Handbook of Formal Languages: Vols. 1 - 3; Springer; ISBN 978-3540614869. *(cited on page 57).*

SANDÉN, B. I. (2003); *Entity-Life Modeling: Modeling a Thread Architecture on the Problem Environment*; IEEE Softw.; 20(4):70–78; doi:10.1109/MS.2003.1207459. *(cited on page 38).*

SARKAR, S. and THONSE, S. (2004); *EAML – Architecture Modeling Language for Enterprise Applications*; in: CEC-EAST '04: Proceedings of the E-Commerce Technology for Dynamic E-Business, IEEE International Conference on (CEC-East'04); pp. 40–47; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-2206-8; doi:10.1109/CEC-EAST.2004.37. *(cited on page 227).*

SCHMERL, B. and GARLAN, D. (2004); *AcmeStudio: Supporting Style-Centered Architecture Development (Research Demonstration)*; in: ICSE '04: Proceedings of the 26th International Conference on Software Engineering; pp. 704–705; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-2163-0. *(cited on pages 61 and 147).*

SCHMIDT, D. C. and BUSCHMANN, F. (2003); *Patterns, frameworks, and middleware: their synergistic relationships*; in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering; pp. 694–704; IEEE Computer Society; ISBN 0-7695-1877-X. *(cited on pages 82, 83, and 253).*

SCHMIDT, D. C.; JOHNSON, R. E.; and FAYAD, M. (1996); *Software Patterns*; Communications of the ACM; 39(10):37–39; doi:10.1145/236156.236164. Special Issue on Patterns and Pattern Languages. *(cited on pages 104 and 107).*

SCHMIDT, D. C.; STAL, M.; ROHNERT, H.; and BUSCHMANN, F. (2000); Pattern-oriented software architecture, vol. 2: Patterns for concurrent and networked objects; Wiley series in software design patterns; Wiley, Chichester; ISBN 0-471-60695-2. *(cited on page 103).*

SCHWANKE, R. W. (2001a); *Layers, Decisions, Patterns, Styles, and Architectures*; in: WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01); p. 137; IEEE Computer Society, Washington, DC, USA; ISBN 0-7695-1360-3; doi:10.1109/WICSA.2001.948423. *(cited on page 211).*

—— (2001b); *Toward a Real-time Event Flow Architecture Style*; in: Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01); pp. 94–; doi:10.1109/ECBS.2001.922410. *(cited on page 211).*

SCHWARTZ, E. S. and TRIGEORGIS, L. (editors) (2001); Real Options and In-

vestment Under Uncertainty: Classical Readings and Recent Contributions; The MIT Press; ISBN 978-0262194464. *(cited on page 233).*

Sefika, M.; Sane, A.; and Campbell, R. H. (1996); *Monitoring compliance of a software system with its high-level design models*; in: ICSE '96: Proceedings of the 18th international conference on Software engineering; pp. 387–396; IEEE Computer Society; ISBN 0-8186-7246-3. *(cited on pages 36 and 201).*

Selby, R. W. (2005); *Enabling Reuse-Based Software Development of Large-Scale Systems*; IEEE Trans. Softw. Eng.; 31(6):495–510; doi:10.1109/TSE.2005.69. *(cited on page 90).*

Selic, B. (2005); *Architectural Modeling Capabilities of UML 2.0*; in: UML 2 glasklar (edited by Rupp, C.; Hahn, J.; Queins, S.; Jeckle, M.; and Zengler, B.); chapter 9; Carl Hanser, München. *(cited on page 142).*

Selonen, P. and Xu, J. (2003); *Validating UML models against architectural profiles*; in: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering; pp. 58–67; ACM Press, New York, NY, USA; ISBN 1-58113-743-5; doi:10.1145/940071.940081. *(cited on page 210).*

Shaw, M. (1995a); *Comparing Architectural Design Styles*; IEEE Softw.; 12(6):27–41; doi:10.1109/52.469758. *(cited on page 49).*

—— (1995b); *Architectural issues in software reuse: it's not just the functionality, it's the packaging*; SIGSOFT Softw. Eng. Notes; 20(SI):3–6; doi:10.1145/223427.211783. *(cited on pages 49 and 104).*

—— (1996); *Some Patterns for Software Architecture*; in: Pattern Languages of Program Design, Vol. 2 (edited by Vlissides, J. M.; Coplien, J. O.; and Kerth, N. L.); pp. 255–269; Addison-Wesley, Reading, MA, USA. *(cited on page 49).*

Shaw, M. and Clements, P. (1996); *How Should Patterns Influence Architectural Description Languages?*; in: Working Paper for DARPA EDCS community. *(cited on page 51).*

Shaw, M. and Clements, P. C. (1997); *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*; in: 21st International Computer Software and Applications Conference (COMPSAC '97), 11-15 August 1997, Washington, DC, USA; pp. 6–13; IEEE Computer Society; ISBN 0-8186-8105-5. *(cited on pages 49 and 223).*

Shaw, M. and Garlan, D. (1996); Software architecture: perspectives on an emerging discipline; Prentice-Hall, Inc.; ISBN 0-13-182957-2. *(cited on pages 12, 29, 101, and 104).*

Shaw, M.; DeLine, R.; Klein, D. V.; Ross, T. L.; Young, D. M.; and Zelesnik, G. (1995); *Abstractions for Software Architecture and Tools to Support Them*; IEEE Trans. Softw. Eng.; 21(4):314–335; doi:10.1109/32.385970. *(cited on pages 49 and 59).*

Smith, G. (2000); The Object-Z Specification Language; Advances in Formal

Methods; Kluwer; ISBN 0-7923-8684-1. *(cited on page 85).*

Software Engineering Institute (2008); *Software Architecture for Software-intensive Systems – How do you define software architecture?*; URL `http://www.sei.cmu.edu/architecture/definitions.html`. *(cited on page 77).*

van Solingen, R. and Berghout, E. (1999); The goal/question/metric method : a practical guide for quality improvement of software development; McGraw-Hill; ISBN 0-07-709553-7. *(cited on pages 26, 186, and 243).*

Sparks, S.; Benner, K.; and Faris, C. (1996); *Managing Object-Oriented Framework Reuse*; Computer; 29(9):52–61; doi:10.1109/2.536784. *(cited on page 91).*

Stachowiak, H. (1973); Allgemeine Modelltheorie; Springer, Wien; ISBN 3-211-81106-0. *(cited on pages 30 and 31).*

Sullivan, K.; Chalasani, P.; and Jha, S. (1997a); Software Design Decisions as Real Options; Technical report 97-14; University of Virginia, Department of Computer Science. *(cited on page 233).*

Sullivan, K.; Chalasani, P.; Jha, S.; and Sazawal, V. (1999); *Software Design as an Investment Activity: A Real Options Perspective*; in: Real Options and Business Strategy: Applications to Decision Making (edited by Trigeorgis, L.); chapter 10; Risk Books, London. *(cited on page 233).*

Sullivan, K. J.; Socha, J.; and Marchukov, M. (1997b); *Using formal methods to reason about architectural standards*; in: ICSE '97: Proceedings of the 19th international conference on Software engineering; pp. 503–513; ACM Press, New York, NY, USA; ISBN 0-89791-914-9; doi:10.1145/253228.253433. *(cited on pages 117 and 120).*

Sullivan, K. J.; Griswold, W. G.; Cai, Y.; and Hallen, B. (2001); *The structure and value of modularity in software design.*; in: ESEC / SIGSOFT FSE; pp. 99–108. *(cited on page 233).*

Szyperski, C. (2002); Component software : beyond object-oriented programming; Addison-Wesley Longman; 2nd edition; ISBN 978-0201745726. *(cited on pages 40, 91, and 124).*

Taibi, T. (editor) (2007); Design pattern formalization techniques; Idea Group, Hershey, PA, USA; ISBN 1-599-04219-3. *(cited on pages 50 and 52).*

Taylor, R. N.; Medvidovic, N.; Anderson, K. M.; E. James Whitehead, J.; Robbins, J. E.; Nies, K. A.; Oreizy, P.; and Dubrow, D. L. (1996); *A Component- and Message-Based Architectural Style for GUI Software*; IEEE Trans. Softw. Eng.; 22(6):390–406; doi:10.1109/32.508313. *(cited on page 105).*

Teeuw, W. B. and van den Berg, H. (1997); *On the Quality of Conceptual Models*; in: Proc. ER'97 Workshop on Behavioral Models and Design Transformations (edited by Liddle, S. W.); URL `http://osm7.cs.byu.`

edu/ER97/workshop4/tvdb.html. *(cited on page 24).*

Tracz, W. (1993); *Parametrized programming in LILEANNA*; in: SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing; pp. 77–86; ACM Press, New York, NY, USA; ISBN 0-89791-567-4; doi: 10.1145/162754.162815. *(cited on page 57).*

Tyree, J. and Akerman, A. (2005); *Architecture Decisions: Demystifying Architecture*; IEEE Softw.; 22(2):19–27; doi:10.1109/MS.2005.27. *(cited on page 227).*

Ulbts, J. (2008); Ein Repository zur Unterstützung der Auswahl von Architekturstilen im Rahmen der MidArch-Methode; Diplomarbeit; Carl von Ossietzky University Oldenburg, Department of Computing Science, Software Engineering Group. Supervised by Wilhelm Hasselbring and Simon Giesecke. *(cited on pages 197, 224, and 225).*

University of California at Irvine (); *xADL 2.0 – A Highly Extensible Architecture Description Language for Software and Systems*. http://www.isr.uci.edu/projects/xarchuci/index.html. *(cited on page 62).*

Van Der Aalst, W. M. P.; Ter Hofstede, A. H. M.; Kiepuszewski, B.; and Barros, A. P. (2003); *Workflow Patterns*; Distrib. Parallel Databases; 14(1):5–51; doi:10.1023/A:1022883727209. *(cited on page 55).*

van Emde Boas, P. (1997); Resistance is Futile; Formal linguistic observations on design patterns; Technical report ILLC-CT-97-02; ILLC, FWINS, Universiteit van Amsterdam; URL http://www.wins.uva.nl/research/illc/ResearchReports/CT97-02.text.ps.gz. *(cited on page 102).*

Vogel, D. R. and Wetherbe, J. C. (1984); *MIS Research: A Profile of Leading Journals and Universities*; DATA BASE; 16(1):3–14. *(cited on page 69).*

W3C (2007); Web Services Description Language (WSDL) 2.0; URL http://www.w3.org/TR/wsdl20/. *(cited on page 101).*

Wagelaar, D. and Jonckers, V. (2005); *Explicit Platform Models for MDA*; in: Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings (edited by Briand, L. C. and Williams, C.); volume 3713 of *Lecture Notes in Computer Science*; pp. 367–381; Springer; ISBN 3-540-29010-9; doi:10.1007/11557432_27. *(cited on page 231).*

Weinberg, G. M. (1998); The psychology of computer programming (silver anniversary ed.); Dorset House Publishing Co., Inc., New York, NY, USA; ISBN 0-932633-42-0. *(cited on page 40).*

Withall, S. (2007); Software Requirement Patterns; Best Practices; Microsoft Press; ISBN 0735623988. *(cited on page 55).*

Wohlfarth, S. and Riebisch, M. (2006); *Evaluating Alternatives for Architecture-Oriented Refactoring*; in: 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06); pp. 73–79; IEEE Comp. Soc. Pr.; doi:10.1109/ECBS.2006.40. *(cited on page 230).*

World Wide Web Consortium (2006); *Extensible Markup Language (XML)*. `http://www.w3.org/XML/`. *(cited on page 62).*

Zarras, A.; Issarny, V.; Kloukinas, C.; and Nguyen, V. (2001); *Towards a Base UML Profile for Architecture Description*; in: Proceedings of ICSE 2001 Workshop for Describing Software Architecture with UML; pp. 22–26; IEEE Computer Society. *(cited on page 211).*

Zdun, U. and Avgeriou, P. (2005); *Modeling architectural patterns using architectural primitives*; in: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications; pp. 133–146; ACM Press, New York, NY, USA; ISBN 1-59593-031-0; doi:10.1145/1094811.1094822. *(cited on page 210).*

Zelesnik, G. (1996); *The UniCon Language Reference Manual*; URL `http://www.cs.cmu.edu/~UniCon/reference-manual/Reference_Manual.html`. *(cited on page 59).*

Zendler, A. and Schwartzel, H. G. (1998); *From logical to physical software architectures*; IETE technical review; 15(5):355–369. *(cited on page 46).*

Zhu, Y.; Huang, G.; and Mei, H. (2005); *Modeling diverse and complex interactions enabled by middleware as connectors in software architectures*; in: 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005); pp. 37–46; IEEE Computer Society Press; ISBN 0-7695-2284-X; doi:10.1109/ICECCS.2005.63. *(cited on page 120).*

Zimmer, W. (1995); *Relationships between Design Patterns*; in: Pattern languages of program design (edited by Coplien, J. O. and Schmidt, D. C.); pp. 345–364; Addison-Wesley, Reading; ISBN 0201607344. Proc. PLoP'94. *(cited on pages 55 and 103).*

Zimmermann, O.; Gschwind, T.; Küster, J. M.; Leymann, F.; and Schuster, N. (2007); *Reusable Architectural Decision Models for Enterprise Application Development*; in: Proceedings of QoSA 2007 (edited by Overhage, S.; Szyperski, C. A.; Reussner, R.; and Stafford, J. A.); volume 4880 of *Lecture Notes in Computer Science*; pp. 15–32; Springer; ISBN 978-3-540-77617-8; doi:10.1007/978-3-540-77619-2_2. *(cited on page 209).*

Zimmermann, O.; Zdun, U.; Gschwind, T.; and leymann, F. (2008); *Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method*; in: WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008); pp. 157–166; IEEE Computer Society, Washington, DC, USA; ISBN 978-0-7695-3092-5; doi:10.1109/WICSA.2008.19. *(cited on page 213).*

Lebenslauf von Simon Giesecke

Persönliche Daten

Akademischer Grad	Diplom-Informatiker
Geburtstag und -ort	1980-04-18 in Herne
Staatsangehörigkeit	Deutsch

Schulausbildung

08/1986–07/1990	Grundschule in Recklinghausen
08/1990–05/1998	Gymnasium in Recklinghausen
05/1998	Abitur

Studium

10/1998–07/2001	Studium der Mathematik mit dem Abschlussziel Diplom an der Carl von Ossietzky Universität, Oldenburg
04/1999–09/2003	Studium der Informatik mit dem Abschlussziel Diplom an der Carl von Ossietzky Universität, Oldenburg
07/2001	Vordiplom in Mathematik
09/2003	Abschluss als Diplom-Informatiker

Arbeitsverhältnisse

10/2000–02/2002	Studentische Hilfskraft an der Universität Oldenburg
03/2003–09/2003	Studentische Hilfskraft am OFFIS Institut für Informatik
10/2003–06/2004	Stipendiat im DFG-Graduiertenkolleg "'Software für mobile Kommunikationssysteme"' an der RWTH Aachen, Prof. Dr.-Ing. M. Nagl
07/2004–08/2007	Wissenschaftlicher Mitarbeiter in der Abteilung Software Engineering, Carl von Ossietzky Universität Oldenburg, Prof. Dr. W. Hasselbring
07/2007–	Wissenschaftlicher Mitarbeiter am OFFIS Institut für Informatik, Oldenburg

Oldenburg, 24. Juni 2008