T U M

I N S T I T U T F Ü R I N F O R M A T I K

The Field of Software Architecture

Christoph Hofmann, Eckart Horn, Wolfgang Keller,
Klaus Renzel, Monika Schmidt

TUM–I9641
November 96

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-96-I9641-250/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1996

Druck: Fakultät für Mathematik und
Institut für Informatik der
Technischen Universität München

The Field of Software Architecture[1]

Christoph Hofmann[2], Eckart Horn[2], Wolfgang Keller[3],
Klaus Renzel[3], Monika Schmidt[2]

E-Mail: {hofmannc, horn, schmidtm}@informatik.tu-muenchen.de
{wolfgang.keller, klaus.renzel}@sdm.de

Edited by Eckart Horn[2] and Monika Schmidt[2]

November 7, 1996

Abstract:

Modern software systems are becoming more and more complex. The importance of distributed systems and networking increases. To guarantee the ability of understanding and maintaining complex systems, some questions concerning the structure of software systems are increasingly important to deal with. Our paper gives a survey of existing approaches in the field of software architecture. With the term software architecture we mean the gross structure of a software system. In this context several questions arise: What are good software structures and how to find them? How to describe software architectures?

This paper starts with a motivation for dealing with software architectures. It gives a short overview of the software development process to show the role of software architecture within this process. After that, we give a survey of some formal approaches by the research community concerning the basic structure of software systems (architectural styles) and the question of how to write them down (architectural description languages). A selection of pragmatic approaches by the software development industry (design patterns, frameworks, domain-specific software architectures and technical standard architectures) is presented thereafter. Their primary goal is to reuse design ideas already known or even available code.

---

[1] This work was sponsored by the *Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie* (BMBF) under the project 'ENTSTAND' and by the *Deutsche Forschungsgemeinschaft* (DFG) under the project 'Bellevue'

[2] TU München

[3] software design & management (sd&m) GmbH & Co. KG, München

Contents

1 Introduction: The Field of Software Architecture 3

2 The Role of Software Architecture 4

2.1 Motivation: Improving the Software Engineering Discipline 4
2.2 Software Development Process 5
2.2.1 Phases of Software Development 5
2.2.2 Models, Views, and Abstraction Levels 7
2.3 Software Architecture in the Software Development Process 8
2.3.1 General Problem and Definition 8
2.3.2 Abstraction Levels and Models 10
2.3.3 Requirements Relevant to Software Architecture 11
2.3.4 Reuse in the Context of Software Architecture 12
2.4 Quality Attributes and Software Design 12

3 A Survey of Existing Approaches 15

3.1 Architectural Styles 16
3.1.1 Examples of Common Styles 17
3.1.2 Conclusion 18
3.2 Architectural Languages 19
3.2.1 Language Criteria 19
3.2.2 Survey of Existing Languages and Environments 20
3.2.3 Conclusion 22
3.3 Design Patterns 23
3.3.1 Design Patterns in the Context of Object-Orientation 24
3.3.2 Classification of Design Patterns 26
3.3.3 How to Describe Design Patterns 27
3.3.4 Conclusion 27
3.4 Frameworks 28
3.4.1 Fundamental Ideas of Frameworks 29
3.4.2 Examples of Application Frameworks 31
3.4.3 Conclusion 31
3.5 Domain-Specific Software Architectures (DSSAs) 32
3.5.1 Domain Architectures for Business Information Systems 33
3.5.2 sd&m Standard Architecture 34
3.5.3 Application Architecture for Insurance Companies (VAA) 37
3.5.4 Conclusion 38
3.6 Technical Standard Architectures 38
3.6.1 Systems Application Architecture (SAA) 39
3.6.2 Open Blueprint 41
3.6.3 Conclusion 43

4 Conclusion and Outlook 44

Acknowledgements 45

Glossary 46

References 48

1 Introduction: The Field of Software Architecture

There is growing interest in software architecture both in industry and in research groups. Development engineers hope to overcome shortcomings in current software engineering practices by concentrating on the structure of software systems.

Building a good software architecture is a very ambitious task. Complex systems have to fulfil many requirements, functional and nonfunctional ones. A well-organized structure of a software system (that is, a good software architecture) reduces complexity and thus leads to a better understanding of how the system works. This is a prerequisite for making it possible to construct a system that fulfils its requirements. The structure of the system is also responsible for making it easier to build and maintain the system. Therefore, the software architecture is relevant to almost the whole development process.

The term software architecture is very popular. This is not surprising because modern software systems have become more and more complex, distribution and networking play an important role. Yet the idea behind software architecture is not a new one. As early as in the late sixties one tried to solve the question of how to describe algorithms. The term 'Programming in the Small' evolved, meaning the algorithmic notion of programming. Later on, the focus of attention moved to the dependencies between different modules. This first bottom-up approaches to software architectures resulted in the first module interconnection languages (MILs) (see [PN86]) and the term 'Programming in the Large'. In the early nineties, there were the first research papers and projects that tried to develop an architectural and structural view of a software system (see [Der91], [GS93]) which is able to capture its gross organisation. The first workshops committed to this problem took place in 1995 (see [Gar95], [GTP95]). However, until now there is no common vocabulary nor a stable theory. There are many open questions and therefore much work has to be invested in this emerging field.

The goal of this paper is to give a survey of this ongoing work in the field of software architecture in an integrated and uniform way. In order to achieve this integration, we start Chapter 2 with a motivation and propose our definition of the term ‘software architecture’ and its role in the software development process. The term ‘software architecture’ is often used to denote the discipline of how to structure software systems. But it is also used when talking about the structure itself. In this paper we use the term only in the latter meaning. We deal with models, views, and abstraction levels that are relevant to the description of software architectures, show the relevant requirements, and try to explain some architectural reuse approaches. The chapter ends with some thoughts on quality attributes influenced by a software architecture.

In Chapter 3, existing and often isolated approaches that lie within this broad field of software architecture are introduced. We give a survey of several theoretical approaches (architectural styles and architectural description languages) and of more practical ones (design patterns, frameworks, domain-specific software architectures, and technical standard architectures).

Chapter 4 summarizes these different approaches and gives some hints on how they are connected and how they could be merged.

Our paper ends with a glossary of important terms used in this survey and the citation index.

2 The Role of Software Architecture

This chapter presents some thoughts on the role of software architecture that we have gained from our studies of the relevant literature, from experiences in software development projects, and from many discussions. Our overall picture has not been confirmed by the software engineering community yet, but it provides an orientation for our further work. In this paper, it serves to give a fundamental understanding of the field of software architecture and gives a framework the existing approaches can be inserted in.

Section 2.1 gives the motivation for formal approaches in the software engineering process, especially for architectural issues. Section 2.2 briefly sketches the development process and introduces

- analysis, design, and implementation,
- models in different phases and on different abstraction levels, and
- possible mappings between models.

This serves as a base for Section 2.3, where we want to show the relationship between the architecture and the development process. We begin with a definition of the term software architecture followed by some hints on architectural models and views, on what kinds of models exist, and when and for what purpose they can be used. The last part of Section 2.3 deals with different approaches to software reuse and shows how they can be supported by software architecture. Finally, in Section 2.4 we reflect upon the characteristics of a good architecture, upon quality factors, and possible design operations to support them.

2.1 Motivation: Improving the Software Engineering Discipline

The term ‘Software Engineering’ was first heard in 1968 (see [NR69]). It incorporated the vision that software - just like in other engineering disciplines - could be produced in a well-organized and planned manner that would ensure the required quality of the product. This should solve the problems of that time, namely that development of software required too much time, that it did not solve the problems users were confronted with, and that it could not be maintained effectively because it was developed in too messy a way, very difficult to understand and to change. Meanwhile, there has been some success. Structured higher programming languages have been developed, which make for easier understanding and description of programs. The importance of a standardization of the development process itself was recognized, leading to elaborated process models. Specification techniques have been developed for describing the functionality of a system. The research community concentrated on formal description techniques, whereas in industry many informal, mostly diagrammatic specification techniques have spread.

But there is still much potential in moving towards the vision of developing software as an engineering discipline. This will become clear when we look at the problems and shortcomings of current software engineering practices: there is a big gap between the two ends of the development process. In the beginning, the software engineer has to understand and describe the application domain (e.g., its semantic entities and their interdependencies, its data and functions) to be able to guarantee that the users get what they want. This domain is the so-called problem space. At the end of the process, the software system in the solution space should be completely implemented in the form of programs and code, fulfilling all - functional

and nonfunctional - requirements. Until now, it is not at all clearly understood how a development process must look like to bridge the gap between problem and solution space and what intermediate models and artifacts are useful.

From our engineering vision of the process, a step towards an improvement of the engineering process is the application of formal methods, in order to work with precise formal models that provide unambiguous semantics. Currently, numerous informal techniques that lack the rigor of formality are applied in nearly all phases of the development process in industry. Formalisation should cover large parts of the development process. Hence besides the task of finding appropriate formal models, we need to integrate all these models, for example by finding mappings between them.

The advantages of dealing with software architecture, or even with a formal approach with respect to software architecture, are manifold. Firstly, the high abstraction level of an architectural description reduces complexity and promotes better understanding of the software system both in development and, especially, during the later maintenance phase. Secondly, a global view of the system as a whole is adequate for an engineer who has to develop a complex product and first works with a model to reduce the risk of failing. Thirdly, formal models that are present in machine-readable format may provide help of a new quality during the further development steps. For example, they support analysis of certain nonfunctional system properties or generation of code stubs. Finally, there is a big potential of reuse. On the one hand, single components can be reused. This componentware approach with plug-in compatible components requires an appropriate infrastructure, e.g. standardized interfaces. On the other hand, it is possible to reuse architectural structures and organization principles which cover whole applications and systems (e.g. frameworks, domain-specific architectures, and technical standard architectures (see Sections 3.4, 3.5, and 3.6)) or certain limited problems (e.g. design patterns (see Sections 3.3)). This reuse approach would be possible, if architectural notions and semantic entities are made explicit in an architectural description language.

2.2 Software Development Process

This section presents a brief introduction to the software development process in order to give a better understanding of the significance of building a software architecture within the software development process.

2.2.1 Phases of Software Development

Each of today's software development methods has its own process model and methodology. People concerned with software development mostly agree to partition the process into different steps, also called phases. In each of these steps, different descriptions of either the *problem* or the *solution* are produced. These development steps may be executed in sequence (classical waterfall model), in iteration (spiral model), or in some derivation of both. In the following, we do not want to fix the order of the different steps. Everything is allowed: iterative, parallel, or incremental proceeding. Even though the steps are defined variably, we can summarize their most important features:

- In the *analysis* (*specification*) *phase,* models of the problem to be solved are created. They cover that part of the real world that is relevant to the software system. Functional requirements determined by the application domain are incorporated into these models, nonfunctional ones (like response time, error frequency, programming language, operating system, hardware) are just written down.
- In the *design* (*construction*) *phase* we create models of the solution. The set of all possible solutions is restricted by the functional and nonfunctional requirements. The analysis models of the problem are mapped to models of the solution space. The main goal of the design phase is to structure the software system under development. This is the phase where the software architecture is determined.
- In the *implementation* (*coding*) *phase* we get a usable solution (in terms of code) out of the design models.

Like in real life, in software development one should take solutions into account that are already known and used successfully. In other words, we want to reuse models and process steps we already made. This can happen in each of the development steps mentioned above (analysis, design and implementation) and on different abstraction levels. For example, one can use specification models of data types or control and data flows of recurring real world problems during analysis, standard architectures or patterns for the design step, and code libraries or frameworks for implementation.

Figure 1 illustrates the relationship between models in different phases. The models of analysis are the base for those of the design phase, which are themselves the base for implementation models. Standards may influence models in all phases and may support reusability of models and process steps. These standards may be guidelines for the models that have to be built and for the choice of appropriate description techniques. They may also provide a methodology for building these models.

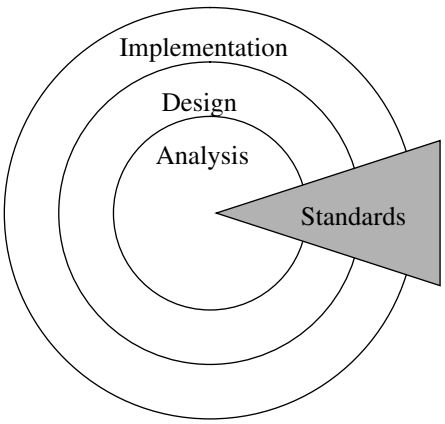

Figure 1 : Reuse in different phases

2.2.2 Models, Views, and Abstraction Levels

As we have seen above, at each of the development steps models of either the problem or the solution are created. The reader may ask what these models describe and what they are intended for. The leading goal of the description is to write down the characteristics of the system under development, to provide a base for communication between customers, users, and developers.

In each of the phases the description of the system under development is quite complex, thus it is described by several views. A view is nothing else than a perspective of looking at the system. Each view describes one or more aspects of the system and thus provides a model. For example, looking at the data structures leads to a data view (data model) and looking at the parts of the system and the control flow between them leads to a control flow view (control flow model).

In our opinion, there are some aspects that are described in each development phase. These are the *static structure* and *dynamic behaviour* of system parts on different abstraction levels (from the whole system to elementary parts). To show the static structure, a description of the system parts and their relationship with each other is needed. For the description of the dynamic behaviour one can imagine data and control flow, state transitions or life cycles.

To describe the different models, several description techniques are needed. Available techniques are algebraic specification, entity-relationship-diagrams, text, automata, class diagrams, message sequence charts, data flow diagrams, etc. One diagram can describe one or more aspects of a model and in turn one model can be described by several diagrams. In this context, the following questions arise: how to check the consistency of two models and do they describe the same facts?

There are three important relationships between models:

- **refinement mapping:** if one model is a refinement of another, then it provides a more detailed description of the same aspects. Both models belong to the same development phase but provide descriptions of the same aspects on different levels of abstraction.
- **complementary descriptions:** a set of models that describe different aspects of the system within one development phase. These models have to be consistent.
- **construction mapping:** models of different development phases are related by construction mapping. The original model is extended with regard to additional requirements. An example is a mapping between analysis and design models. In addition to the functional requirements of the analysis, the design models consider the nonfunctional requirements.

In the next section we will have a closer look at models that are appropriate to describe a software architecture.

2.3 Software Architecture in the Software Development Process

We just give a short description of the fundamentals of software development where we saw the need for different models on different abstraction levels in different development phases. To show the importance of software architecture in the software development process and to describe models and requirements that are relevant to software architecture, a definition of that term is needed.

2.3.1 General Problem and Definition

The main goal of any software development process is to produce a software system that has the desired behaviour. Not only functional but also nonfunctional properties are relevant to the behaviour of the system. As stated above, a software architecture has to be described by several models that reflect these relevant properties with respect to behavioural and structural aspects. It is not efficient and practically not possible to work with a monolithic model that does not have an internal structure at all. Thus, the behaviour of the system has to be built of smaller behavioural units, which partition the behaviour and constitute the internal system structure. As these units cooperate, some form of interaction has to take place between them.

We use the following terms for describing these facts (see also Figure 2): the *system* is constructed of *components* which are the internal behavioural units. The behaviour of both the system and its internal components is described by the respective *interface*. An interface may have a static part (comprising properties that do not change over time, e.g. the data types allowed to work with) and a dynamic part (the actual behaviour, e.g. the possible traces of inputs and corresponding outputs). The interaction between these components has to be described in some way. Its static properties (e.g., paths of communication flows) are the *static structure* of the system, whereas certain behavioural properties (e.g., communication protocols) represent the *interaction* of these components. The *system interactions* describe the interactions of the system with its environment, i.e. users or neighbouring systems.

Figure 2 shows two levels of abstraction of a software system. The first level describes the system by its interface and interactions. It is refined to the internal component structure of the system. Taken together, these interacting components have to offer the same external interface and the same interaction behaviour as the system.

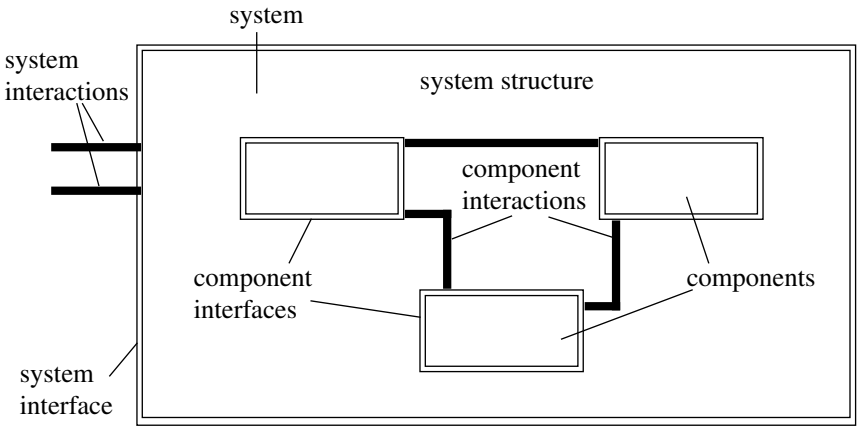

Figure 2 : Architectural System Description

These considerations lead us to the following definition of the term software architecture:

> A software architecture is the structure of a software system. It is described as a set of software components and the relationships between them. For a complete description of an architecture several models and views are needed, with each model describing one or more structural aspects on a certain level of abstraction.

Important examples of structural aspects mentioned in our definition are:

- The **static structure** of the system in terms of related components, which are described by their static interface specification, that means by the properties that do not change during a system run.
- The **dynamic structure** of the system at runtime of the program in terms of interacting components. The description shows the way these components work together to perform the services. Following our definition, we assume that software systems are composed of interacting components, which themselves may be composed of still smaller ones. As a consequence, the behaviour of the system is determined by the behaviour of all its components.
- The **file structure** of the program code in terms of files and include relationships.

The description of a software architecture serves two important purposes. Firstly, it gives a global view of the system and leads to a better understanding. Secondly, the describing models allow us to analyse certain properties of the system without implementing it. Which models are needed to support these purposes depends on the kind of system and on the properties to be analysed.

The definitions of software architecture in literature are not in contradiction with ours, but are often a little bit more specialized. In most cases they only deal with the static and the dynamic structure of a software system and do not consider aspects like code organization. [GS93] treat '*architecture of a specific software system as a collection of computational components - or simply **components** - together with a description of interactions between components - the **connectors***.' [SNH95] state that '*software architectures describe how a system is decomposed into components, how these components are interconnected and how they communicate and interact with each other.*' These two definitions emphasize the description of the software system as a distributed system, mainly concentrating on the aspects of interaction. [BMR+96] also mention nonfunctional properties that have to be represented in an architecture, which is the '*description of the subsystems and components of a software system and the relationships between them, typically represented in different views to show the relevant functional and nonfunctional properties.*' In Section 2.3.3 we will have a closer look at the requirements that influence the decisions concerning the overall system structure.

2.3.2 Abstraction Levels and Models

According to our definition, a software architecture is described by models on certain levels of abstraction. It is left open what these levels of abstraction of the architectural description are. The levels are characterized by the semantic entities available for the description of the components and their interaction.

Concerning the components, the main entities carrying semantic information with respect to behavioural descriptions are the component interfaces which are able to send and receive messages, i.e. data or events. Concerning their interaction, the main semantic entities are the basic protocol mechanisms the components can rely on. There is a broad spectrum of possible levels of abstraction especially with the protocol mechanisms, as can be seen for example in the OSI layered communication model, where at least every layer represents a level of abstraction (see [ISO84]).

A further question in this context is whether there are special types of components or interactions that provide an even higher abstract notion of the respective semantic entity. The approaches at Carnegie Mellon University (CMU) (see [GS93], [AG94]) are based on the fundamental abstraction of ‘components’ and of ‘connectors’, which describe the interaction between the components. In the description language of UniCon [SDK+95], special abstract types for these components and connectors are provided, for example a ‘computation’ component or a ‘procedure call’ connector (for a short description of UniCon see Section 3.2).

The level of abstraction has to be chosen deliberately and there might be several architectural models at different levels. However, there should always be a mapping that describes the refinement relationship between these models and assures consistency. As stated above, the models should allow to analyse certain system properties. To make this possible, the models must have a sufficiently low level of abstraction that depends on the property to be looked at.

The approach by CMU regards an architectural model as a system consisting of components and connectors and uses these terms in a very abstract manner. In contrast, Soni et al. [SNH95] use these terms in a very concrete way. They distinguish between several architectural models, with each model describing different aspects of the software system. Thus they give concrete examples of architectural models. They have models concerning the logical structure of the system, namely the *conceptual architecture* and the *module architecture*. The *conceptual architecture* describes the system by its major design elements and the relationships between these elements. The *module architecture* looks at the functional decomposition and the layers of the system. Furthermore, there exists a model called *code architecture,* which describes how source code is organized in the development environment. Finally, there is the *execution architecture,* which provides a description of the dynamic structure of the software system in terms of resource mapping.

All these models provide a structured description of certain system aspects. For structuring purposes components and connectors are used. However, Soni et al. use these terms only in the conceptual architecture. All other architectural models use terms for structuring that are closer to implementation. For example, the *module architecture* divides the system into the components called subsystems or modules connected by export and import relationships. All models may exist on different levels of abstraction.

Table 1 summarizes the different architectural models proposed by Soni et al. and gives an overview of the entities and their relationships used in these models. The mentioned architectures are interdependent and they complement each other. As a consequence, all these different models have to be consistent in one software system.

Architecture	Examples of Entities	Examples of Relationships
Conceptual Architecture	Components	Connectors
Module Architecture	Subsystems, Modules	Exports, Imports
Code Architecture	Files, Directories, Libraries	Includes, Contains
Execution Architecture	Tasks, Threads	Inter-Process Communication, Remote Procedure Call

Table 1. The Software Architectures of Soni et al.

2.3.3 Requirements Relevant to Software Architecture

Considering requirements relevant to a software system, one naturally finds the functional ones that deal with the problem itself, independent of a possible solution. These functional requirements are traditionally stated with respect to the interface of the system, for example as pairs of possible input and corresponding output descriptions. They model the system as a black box. However, there are also nonfunctional requirements like scalability, portability, performance, or reusability. They take a glass box view and take into account the internal structure of the system in terms of the solution space. This dichotomy of the black box (what happens) and glass box view (how does it happen) of a system (see also [Tra95]) is a very fundamental one which is characteristic of every solution process.

Exactly these nonfunctional requirements have a great influence on the decisions concerning the software architecture. But there is still another point to take into account which is not so easy to solve. There might be technical requirements demanding a certain technology, determining an operating system, data base system, or programming language. These requirements might also be relevant to the overall system structure.

To deliver a suitable model of the software system under development, as many requirements as possible should be considered and incorporated into this model. But is this possible with these technical requirements when one works on a certain abstraction level that does not go into these technical details? Thus, there is a fundamental tension between abstraction level and concrete technical requirements in this architectural approach. Kiczales [Kic92] takes the point of view that architectural abstractions that hide their implementations are not generally possible.

This problem gives use to a general question regarding the abstraction level of an architectural description. To guarantee that the system will be in compliance with the requirements, a sufficiently low abstraction level of the models is needed. If one works on a higher level, the resulting models have to be refined. Then there might emerge contradictions to the technical requirements and a redesign of the high level architectural models is necessary. However, if the architect was experienced, he/she was probably able to foresee these difficulties and to produce an architecture that would satisfy these later technical requirements as well.

2.3.4 Reuse in the Context of Software Architecture

Another question concerning the development process is whether it is possible to reuse some development knowledge embodied in the process and, furthermore, how this could be done. As we have seen in Section 2.2.2, the engineering process maps models of different levels of abstraction to each other, finally yielding the complete software system. Thus, there are some possibilities of reusing models and mappings. The general question emerges how one could distil this knowledge of the development process and define suitable reusable mappings. We think that this requires great experience on the part of the designer, both with the models and the possible mappings between them.

Regarding the architectural models that are built of components, one could reuse some components in a bottom-up manner or even whole architectures and system designs. The first approach is realized, for example, with libraries of reusable components, but there is the general problem of providing the right context for individual components (see [Big94]). Reusing whole architectures generally entails the refinement and customizing of the reused architecture. This requires generic models together with a generator procedure. Refinement mappings instantiate the gross architectural models, for example by inserting concrete components for an abstract component interface. There are some examples of this kind of architectures. Denert [Den91] informally defines standard or reference architectures both for batch and on-line systems (see Section 3.5.2). The 'refinement' of this architecture would be a system development guided by this structure. Luckham [LKA+95] defines reference architectures by behavioural constraints, and the instantiation of this architecture takes place by so-called event pattern mappings, as the behaviour is defined by patterns of events. Object-oriented frameworks (ET++, Microsoft Foundation Classes (MFC), Interviews, OpenDoc) define a system by predefined classes and the flow of control between them. Frameworks can be instantiated by inheriting framework classes already given (see Section 3.4).

The situation is more complex when dealing with construction mappings, as the mapping depends additionally on the new requirements in the solution space and as the two models might have completely different semantic entities and structures. Domain-specific software architectures (DSSAs, see Section 3.5) are an approach for coping with this complexity in the constructive architectural mapping between problem and solution space. Here, the semantic domain of the models is narrowed by looking at a special application domain only. Within this domain, the requirements and the dependencies between the semantic entities are largely known and certain constructive steps are recommended, which have to be represented by the construction mapping. There are two construction mappings in connection with software architecture, namely from specification to architecture (e.g. DSSAs) and from architecture to modules and code. Both require a restricted application domain because a general mapping between any unrestricted models seems to be too complex and not feasible.

2.4 Quality Attributes and Software Design

Sooner or later, the treatment of software architectures will lead to the question of '*good*' software architectures. The usual approach in coping with this question would be searching for evaluation criteria for '*good*' software architectures. Software architectures will then be rated good if they comply with certain criteria for 'good' architecture or have 'good' values for some

metrics that have to be defined. The software architecture will not have an impact on all conceivable quality attributes (see e.g. [Dun90] for a list of quality attributes) of software systems, but it influences many of them.

We think the search for such criteria is doomed to failure. The property ‘good’ can only be expressed in terms of customer requirements. A software system and also its architecture can only be ‘good’ if they match the customer’s requirements.

As stated above, a software architecture is concerned with ‘positive’ nonfunctional properties of a software system. Some of those properties are:

- scalability
- modifiability
- integrability
- portability
- performance
- reliability
- ease of creation
- reusability

The above quality attributes are only a selection of general quality attributes. They can seldom be expressed in metrics. Some other quality attributes, like e.g. usability, cannot be influenced by architectural measures. Yet other attributes (for instance, adaptability, i.e. modifiability, expandability, and portability) are considered as combinations of primitive quality attributes.

A software architecture should be designed to have exactly those of the above quality attributes that are required by the customer. If e.g. the customer does not need a portable architecture, it would in general not be cost effective to build a portable architecture. This can be seen as a ‘design to quality process’ similar to a ‘design to cost’ or a ‘design to quality’ process in general manufacturing.

The above approach has been presented in an article by Kazman and Bass [KB94]. The article lists some possible design operations that can be applied to design an architecture with desired nonfunctional qualities in mind. Design operations can be seen as transformations applied to a set of architectural building blocks, resulting in a new set of building blocks. Kazman lists the following operations, stating that these are far from all possible architectural operations for structuring software systems.

- **Separation** - placement of a distinct piece of functionality into a distinct architectural component that has a well-defined interface to the rest of the model world
- **Abstraction** - the creation of virtual machines
- **Compression** - the removal of undesired software layers
- **Uniform Composition** - the operation of combining two or more system components into a larger component
- **Replication** - the multiplication of an architectural component within an architecture
- **Resource Sharing** - encapsulation of either data or services in a component for shared use by multiple other components

The article then investigates the connection between operations and quality attributes. This results in a matrix of quality factors versus design operations. A plus sign (+) denotes a positive influence of the operation with respect to the desired quality attribute. A minus sign (-) denotes negative influence, the sign (+/-) positive or negative influence, and an empty box means no correlation.

Unit Operation	**Software Quality Factor**							
	Scalability	**Modifiability**	**Integrability**	**Portability**	**Performance**	**Reliability**	**Ease of Creation**	**Reusability**
Separation	+	+	+	+	+/-		+/-	+
Abstraction	+	+	+	+	-		+	+
Compression	-	-	-	-	+		+/-	-
Uniform Composition	+		+				+	
Replication	-	-		-	+/-	+	-	-
Resource Sharing		+	+	+	+/-	-	+	+/-

Table 2. Software Quality Factors and Operations [KB94]

Kazman and Bass then present some case studies. They show which steps could have been used to derive some well-known software architectures, like e.g. the Model View Controller architecture (see [KP88]), from a set of requirements. Starting point for a derivation is a single architectural component that contains all required functionality. The derivation applies a series of operations that promote the desired quality attributes, each transforming the actual set of components into a new set. The article states that pure knowledge of design operations and their impact on quality could transform the building of a software architecture from an apprenticeship to a goal oriented design process. Similar tendencies can be found in the ‘design pattern movement’. The idea of a derivation of an architecture from a problem statement can be found there as well (see [BJ94]). The pattern movement claims to be able to document design decisions, while Kazman claims to be able to derive an architecture by applying the transformations that promote the desired quality attributes.

Even if it is doubted that the desired software architecture can be constructed without further knowledge only by applying operations to a problem until a goal is reached, the article still presents several derivations for known software architectures.

3 A Survey of Existing Approaches

This chapter presents an overview of leading existing approaches to software architecture. Our goal was to collect well-known concepts and to clarify how they are connected. Therefore, this chapter deals with the following fields: architectural description languages, architectural styles, design patterns, frameworks, Domain-Specific Software Architectures (DSSA), and technical standard architectures.

Sections 3.1 and 3.2 focus on architectural styles and architectural description languages, which are rather theoretical attempts by the basic research community. Both attempts deal with the basic structure of software systems and try to find a mathematical foundation of software architectures. The definition of a software architecture as a cooperation of components and connectors on different levels of abstraction is the basis of these considerations. The question of what kinds of components and connectors exist leads to architectural styles, which are an attempt to find and formalize their properties. Architectural description languages provide a description technique for software architectures based on the terms components and connectors. There are several such languages with formal syntax and semantics. They are more precise than box-and-line diagrams which are commonly used to describe software architectures (for example in DSSAs) today.

Sections 3.3 to 3.6 present approaches that have their origin in the practice of software development. They are less abstract than those developed in academic research, thus they do not speak about structures consisting of abstract components and connectors but of concrete interacting software components. Here, the meaning of the term software component is a matter of abstraction. It may be a single class of an object-oriented programming language, a cluster of a few classes, or a complex subsystem. It is these different levels of abstraction that distinguish the different architectural approaches.

Design patterns (Section 3.3) describe essential design ideas. They describe pairs of special software design problems and proper solutions in a special context. In contrast to DSSAs and frameworks, design patterns are not restricted to concrete families of applications. They do not describe a concrete solution but a pattern of a solution, which has to be instantiated each time it is used. As a rule, the design idea consists of only a few interacting components. Their structure and behaviour are described with text, diagrams, and use-cases using message sequence charts. Patterns exist on different levels of abstraction. There are patterns (called architectural frameworks) that describe the fundamental structure of a software system. In this case, each component itself is a subsystem or a complex cluster of classes. There are also patterns dealing with smaller design problems on a lower level of abstraction. Here, each component of the solution description is a class. Even implementation dependent patterns (called idioms) are known, which concern implementation issues. They often depend on a particular programming language. Design patterns are a very successful approach, at least in object-oriented software development. [GHJ+95] was the best selling computer book in 1995.

Frameworks (Section 3.4) provide generic software designs. A Framework can be adapted to specific needs without changing its architecture. Often the term framework is associated with the implementation of the underlying software design. In this sense, frameworks are pre-fabricated software systems that implement the common functionality and the mature flow of control. They have to be adapted to specific functional requirements by writing certain pieces of code and integrating them at particular intended places. This allows the reuse of code and

design. The extent of frameworks may vary from small software components with only a few classes up to ready to use software systems.

DSSAs (Section 3.5) are domain-specific architectures of, for example, business information systems, compilers, etc. This approach tries to reuse results of the whole software engineering process in a specific application domain. Their origin are large projects in government and industry, operating in big and complex application domains. Within this approach, a software architecture means the decomposition of a system into cooperating subsystems. DSSAs are design solutions already proven or design standards for specific application domains. Both DSSAs and frameworks are tailored to specific application domains, but they differ essentially in their abstraction levels. The descriptions of DSSAs consider the basic structure of a software system and are not concerned with implementation issues. In contrast, the abstraction level of frameworks is considerably closer to implementation.

While DSSAs deal with the structure of application software, technical standard architectures (Section 3.6) focus on technical services, which could be used by every application. These technical standard architectures are independent of concrete applications and are restricted to the domain of common services like communication services, data access, or design of user interfaces. They build a high-level interface to the underlying operating system and thus enable different products and technologies to be integrated. In most cases, they include a development environment with a library of basic services. In Section 3.6 we describe the System Application Architecture (SAA) and Open Blueprint as leading representatives of technical standard architectures.

3.1 Architectural Styles

In recent years, software engineers are more and more concerned with building higher programming constructs for a better handling of the development process. This began with the development of high-level programming languages which use more sophisticated notions (procedure calls, while constructs) than there were in assembler languages. In the seventies, a lot of work was done on abstract data types that raised the design level of data structures. Nowadays, system designers recognize the need for even higher levels of abstraction, and terms like ‘client-server’, ‘abstraction layering’ or ‘distributed systems’ are quite common to describe the overall organization of a system. A problem of these descriptions that is becoming apparent is the lack of knowledge of what they really mean. Each of them means some conventionalized interpretation of a box and line diagram. Looking at our definition in Section 2.3.1, these boxes and lines of the overall system structure correspond to components and their interactions in the sense of a software architecture model. All different terms mentioned above mean different types and properties of components and connections. One type of these components together with a matching type of connector forms a so-called architectural style. A reason for research was to understand their properties and to formalize them, so that everyone can use them in the same way and compare different styles.

After an introductory paper by Perry and Wolf [PW92], much of the work in this field has been done at Carnegie Mellon University. In a first survey paper [GS93], Garlan and Shaw outline a number of common architectural styles on which many systems are currently based. In a recent report [AAG95], Abowd, Allan, and Garlan try to formalize styles in order to understand given informal descriptions of software. [BMR+96] relates architectural styles to architectural frameworks.

3.1.1 Examples of Common Styles

To deal with architectural styles one has to define their contents. A recent workshop [GTP95] defined a style as a certain class of architecture by

- defining a vocabulary of design elements (usually called components and connectors),
- imposing configuration constraints on these elements (e.g., 'no cycles'),
- determining a semantic interpretation for the system description,
- making possible a collection of analysis that can be performed on its system.

In spite of this definition, the formalization of styles has not yet been convincing. Only two styles have been semantically based by [AAG95]. The first three styles described below (pipes and filters, data abstraction, events) give the vocabulary but do not handle configuration constraints or analysis. The last three styles (layered systems, repository/blackboard, broker) are very close to architectural frameworks (see Section 3.3). They just impose a configuration and do not care about exact specification of the design elements. Due to this, a distinction between styles is difficult. They may overlap and the differences between them are often not obvious. The following styles are examples taken from [GS93], some of them are also mentioned as architectural frameworks in [BMR+96].

Pipes and Filters

Components of this style are stream processing functions (filters) with defined input/output sets of streams. They are connected via pipes (that may be typed). This style is appropriate, for example, for compilers composed of several independent subtasks. The components preserve an existing order of the data streams and can start the output before the input stream is processed completely. If this is not assured, the style degenerates into a batch architecture.

Data Abstraction / Object-Orientation

Components of this style are abstract data types or objects that encapsulate their internal state. They are connected by function or procedure invocation. One problem literature does not handle is the dynamic behaviour of object-oriented systems. Objects are created and deleted. The open questions are: How to deal with the identity of objects? Are all possible objects known from the beginning or do we need a new diagram when an object is created or deleted?

Events

Components of the event-based style are objects which have an internal state and methods that can alter this state and announce some events. A connector is an n:m relation between an event and the method that should be invoked. Due to this, an event announcement by one component results in a method invocation of another (or the same) component. For example, programming environments for tool integration are structured according to that style.

The main disadvantage of the event-based style is that the order of method invocation cannot be predicted by a component. Another point is the need of a global repository that knows and registers all possible events and forwards them to the appropriate component. Performance may then become a serious issue, because all components access the repository.

Layered Systems

This and the following styles are defined by their configuration constraints. The components (that are not defined more precisely) of this style are arranged in hierarchical layers. The connectors are protocols between adjacent layers. This is suggestive of the ISO/OSI reference model [ISO84] for communication protocols as a good example of a layered style.

[GS93] states that components of the same abstraction level should be ordered in the same layer. Formal or even semi-formal criteria are not given. Another important aspect is the number of layers. A long way from a high-level function down to its low-level realization may take much time, so that performance problems may occur. No advice for the properties of the different layers is given.

Repository / Blackboard

There is a special configuration made up of three kinds of components in this style. Firstly, there is the central data storage that represents the current state, secondly, a control component for access, and thirdly, independent subsystems for certain tasks which operate on the central data storage. Connectors are direct invocations (without data) and can either mean a call of a task by the store (blackboard) or vice versa (repository). This kind of style is used, for example, in a compiler driven by a symbol table (or a syntax tree) as central data storage. A semantic base for the different components has not yet been published.

Broker (Client/Server)

The motivation of this style is the need for a standardized communication for the components of distributed systems that may even work on different platforms. This style is a special configuration of a call-and-return style, just like layered or repository systems. The central part is the broker component, or better, the broker components on different computer systems that have to communicate. Another task is to handle exceptions and to register client and server objects for mapping their names to physical addresses. Other components are client objects and server objects that implement or use services. The role of a component as either client or server may change dynamically. Hints how to group and divide the objects themselves are missing. A semantic base for this style is not given.

3.1.2 Conclusion

A very important aspect of architectural styles is the formalization of common interpretations of box and line diagrams and the classification of different components and connectors. Nevertheless, it is very difficult to give an exact description of a style. Their differences are often not obvious. Styles can be combined and refined, but there has not been any advice on how to do this.

Styles can give a coarse granularity for architectures, but they do not handle the subsystems they impose. There is no help for refinement and treatment of different levels of abstraction. There is no method to show how to deal with one particular style, when to use it, for what kind of application it fits, and what advantages and disadvantages it has. There is no method that gives exact steps leading from a given problem to the architecture.

The fiction held by all the authors mentioned is that after having recognized the common paradigms and having formalized them, it is easy to choose between design alternatives for the system architecture. Styles may serve as a base for software architectures, but a lot of work has to be done identifying their specific advantages and disadvantages and developing a method as well as rules and guidelines for their application. In this context, [BMR+96] mention design patterns (see Section 3.3) and architectural frameworks. They regard architectural frameworks as extended styles and describe styles in pattern notation. The overall structuring principle provides a set of predefined subsystems as well as rules and guidelines for organizing the relationships between them. Different design patterns then describe a basic scheme for structuring these subsystems. They refine the architecture step by step. Seen this way, architectural styles may serve as a basic idea of architectural frameworks.

3.2 Architectural Languages

As stated earlier, a software architecture involves the description of components of which the system is built, of their interaction (connection), and of patterns or constraints that guide their composition. In the last few years, mainly two ways of describing software architectures have been used: module interconnection languages showing definition/use and import/export structures or informal diagrams and idiomatic phrases. Shaw and Garlan evaluate these existing techniques in their report [GS94] and give criteria that architectural description languages should fulfil. A summary of this work is shown in Section 3.2.1.

In Section 3.2.2, we will show some examples of architectural languages and refer to the mentioned literature. Their common goal is the description of a software architecture in terms of components and connectors. Yet they differ in their underlying special syntax and semantics.

3.2.1 Language Criteria

The most important aspects when describing a software architecture are the syntax and semantics of its elements. Different techniques for different architectural models are mentioned in Section 2.2.2.

The language used should support both decomposition (‘divide & conquer’) and composition of the individual parts (to build larger blocks that can be handled easily or used frequently). These composed blocks can then be understood on their own, without other parts of the system; their description is independent of implementation details. When composing different parts, there should be no need for consideration of heterogeneous descriptions (different notations) of algorithms and data structures. They can be composed by the architectural language, independent of their actual description.

Reuse of composed blocks should be possible. An important aspect of finding the suitable block is the need for a description of the properties and required parameters. The constraints of their application also have to be described in the architectural language. Building larger blocks, one has to take care of these composition constraints. The language should allow to reason about the emerging system, not only about individual elements. For example, a description of the configuration with static and dynamic aspects of the system structure should be possible.

Descriptions with different levels of abstraction are needed to hide unnecessary details when they are not wanted. Yet the distinct roles of the elements should be clear. A facility for analysis, checking or even validation of an architecture (with the requirements given) is the last criterion for an architectural description language mentioned in [GS94].

3.2.2 Survey of Existing Languages and Environments

To use a language like the one described above, a development environment is surely needed. This environment allows easy use of the language by offering tools like graphical editors for creating and browsing designs, architectural consistency checkers, code generators, or pattern repositories. Examples of existing experimental systems and languages are given in the following. We start with Rapide as a widespread architectural language environment. The approach presented here is close to the one explained in Chapter 2. We continue with short descriptions of other environments and refer to the mentioned literature.

Rapide

The Rapide system (see [LKA+95]) has its origin at Stanford University, CA, USA. It is an event-based, concurrent, object-oriented language designed for prototyping system architectures. It allows testing and validation of architectures, dynamic creation of components, checks of the static interface as well as dynamic checks of system predicates.

In Rapide, a ‘reference architecture’ consists of a set of interfaces, a set of connection rules for communication between these interfaces (e.g. asynchronous, synchronous protocols), and a set of formal constraints for their configuration. An interface is a definition of the features provided to and required from other modules in the architecture. It may contain an abstract definition of the behaviour of modules. By omitting algorithmic details from the behaviour, many different modules may satisfy an interface. The modules are specified in Rapide, but it is intended to use C++ or Ada in future. To get an instantiated architecture, module specifications are assigned to the interfaces. Rapide checks whether the module specification fits the assigned interface. Figure 3 illustrates the concept of Rapide.

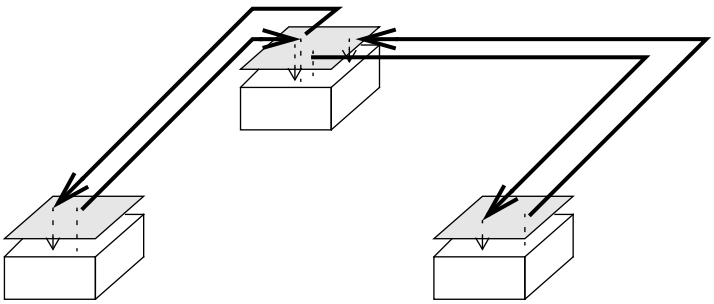

Figure 3 : An instantiated architecture (taken from [LKA+95])

The shaded parallelograms represent interfaces, the bold lines their connections, and the boxes the assigned modules. The architecture is the set of interfaces and connections. The system is a set of modules wired together by the architecture.

Rapide consists of five sublanguages that are compatible as they have the same scoping and naming rules, and underlying execution model. They describe component interfaces (supporting object-oriented and abstract data type styles), constraints on the interface behaviour (by state-machine-like transition rules), event flow between components, the modules themselves (by a set of processes), and event patterns (compositions with possible parameters). The underlying execution model produces a partially ordered event set that represents dependencies between the events (e.g., one event triggers another).

Aesop

In [GAO94], Aesop is described as a system for defining environments for different architectural styles (see also Section 3.1). The description language of Aesop (called Acme) is a kind of C++. It is not really a language but a framework of possible elements that may be used. Types, abstract classes, and methods are predefined and each style is a subset of them. Based on such a style description given by the user (e.g., components, connectors, component ports, connector roles, binding of ports to roles) and a toolkit of common facilities, Aesop produces an editor environment. What Aesop cannot achieve is the combination of different styles in one architectural description.

UniCon

The UniCon system was developed at Carnegie Mellon University, PA, USA, and is documented in [SDK+95]. It is an implementation of an architectural description language together with an environment (compiler, graphical editor, analysis tool invoker). In UniCon, one can define different types of components and different ways of their interaction (connectors). Furthermore, one can incorporate existing code or specifications as components and check configuration constraints.

In UniCon's view, an architecture consists of components, connectors, and their configuration. There are eight predefined component types (module, computation, shared data, sequential file, filter, process, scheduled process, and a general type) with special attributes and players. A player defines a logical point of interaction between the component and its environment. There are seven predefined connector types (pipe, fileio, procedure call, data access, bundler, remote procedure call, and runtime scheduler) with their attributes and roles (e.g., the role of some connector protocol can either be client or server). By mapping the component's players on the connector's roles, the configuration of the system is built.

A complete definition of a component consists of one of the component types mentioned above (with attributes and players), the interface specification, the functionality, and one or more implementations. A composition of components and connections to larger components is possible. The connector’s definition consists of one of the connector types (with attributes and roles), of commitments and assertions (e.g., for time and ordering).

Wright

The concept of the Wright architectural language is very similar to that of UniCon. As stated in [AG94], a system description using Wright consists of three parts: a definition of component and connector types, a set of component and connector instances, and attachments to combine these instances.

A component type is described as a specification of the component's function and a set of ports. Each port defines logical points of interaction between the component and its environment, like the players of UniCon mentioned above. The functionality is defined in Z notation, a mathematical language based on first order logic and set theory.

A connector type definition consists of a set of roles, i.e. the expected local behaviour of an interaction partner (e.g. client or server), and a glue specification. This glue specification describes how these behaviours are coordinated and combined to form a communication protocol. The interaction protocols are described using a variant of CSP (Communicating Sequential Processes). Certain correctness properties, such as absence of deadlock, can be checked. Other tools like editors, repositories etc. are not mentioned by the authors.

To build a system, a set of actual component and connector entities which will appear in the configuration are described. They are combined by describing which component ports are attached to which communication roles and thus they constitute the configuration.

Cham

The Chemical Abstract Machine Model (Cham) referred to in [IW95] considers a computation as transformations (reactions) consuming elements and producing new ones according to certain reaction rules. Since the reactions may take place in any order or even simultaneously, the model allows parallel processing. The elements (molecules) divided into processing, data, and connecting elements are described in terms of syntactic algebra (constants and operations). An example of Cham as an architectural description language with operational semantics is also found in [IW95].

Other approaches

Only recent languages have been mentioned in the sections above. There are many other architectural description languages that have not been investigated in detail yet. Probable candidates for the semantics of connectors (or protocols) are state machines (e.g., I/O-automata, Statecharts), pre- and postconditions, Petri nets, or stream processing approaches.

Module interconnection languages (MILs) are historically grown. Their disadvantage is (according to [AG94]) that they primarily address definition/use (or import/export) relationships of a system and are more concerned with its implementation details. MILs describe the structure of existing code, the code dependencies, and how the component achieves the computation. However, architectural languages claimed in Section 3.2.1 state how components are connected using architectural styles like pipes or shared data.

3.2.3 Conclusion

Most of the criteria given in Section 3.2.1 are fulfilled by today's languages. Especially the aspect of reuse is recognized as very important. Different levels of abstraction are possible through refinement. The composition operators in the languages only differ as to what can be composed to form larger blocks. Some languages compose connectors and components, others only components and some even connectors. It is not clear to what extent heterogeneous formalisms can be used in the different systems. Some systems allow specification of the components by a MIL or abstract data types, others do not mention this.

The varied use of the term software architecture is striking. A distinction between components and connectors is usually made, yet the exact definition of syntax and semantics differs from language to language. As a conclusion, one may draw the analogy to other techniques, for example the one used in requirement specification. There are dozens of formalisms, each with a domain it fits best. The same applies to architectural description languages. Practice and experience with formal and informal description techniques have to point out these domains and the weakness or strength of each language.

3.3 Design Patterns

In daily life there are problems which occur over and over again. Once you have successfully solved a problem, you will try and reuse the essential ideas of this solution when a similar problem occurs. This strategy is followed in the context of software engineering as well. The reuse of a solution for a specific design problem is the central idea of the design pattern approach.

The roots of the design pattern idea go back to the building architect Christopher Alexander. In his two books *A Pattern Language* [AIS+77] and *The Timeless Way of Building* [Ale79], he shows good designs of buildings and communities and how they can be described using pattern languages.

In the context of software engineering, patterns deal with the structure of a software system, that means its architecture. Buschmann proposes a system of patterns with three levels of abstraction [BM95]:

1. **Architectural Frameworks:** they describe the overall structure of large, complex systems in terms of subsystems and the relationships between them. Therefore, they describe the gross architecture of a software system. Architectural frameworks form the highest level of abstraction and are comparable to the architectural styles proposed by Garlan and Shaw [GS93]. For more information about architectural styles see Section 3.1.
2. **Design Patterns:** they describe the structure, i.e. the architecture of subsystems and components of a software system. Thus they can be seen as micro-architectures. The rest of this section will deal with design patterns only.
3. **Idioms:** they form the lowest level of patterns and deal with the implementation of particular components of patterns. They often depend on a particular programming language and are not discussed in this paper. For further information and examples of C++ idioms see [Cop92].

In most cases, design patterns are considered in close connection with object-orientation. Object-oriented languages offer language constructs like abstract classes or inheritance (for further information see [Mey88]), which support the design pattern idea in a very elegant way. This is the reason why the next section describes design patterns in the field of object-orientation. However, it must be emphasizes that the design pattern idea is not limited to this field.

3.3.1 Design Patterns in the Context of Object-Orientation

There is no standardization of the term design pattern in the context of object-oriented software development. Therefore, there is no common definition of that term. As pointed out above, a design pattern describes a solution to a specific design problem with the purpose of reuse in mind.

In [GHJ+95], the term design pattern is characterized by its four essential elements:

1. **Pattern name:** speaking keyword that reflects the design problem.
2. **Problem:** explains the design problem and the context to which the pattern can be applied. Sometimes there is a list of conditions that must be met before it makes sense to use the pattern.
3. **Solution:** describes the elements (the classes and objects) that make up the design, their relationships, responsibilities, and collaborations. A pattern describes how a general arrangement of the design elements solves the problem described in abstract terms. It does not provide a concrete design or an implementation.
4. **Consequences:** outline the results and trade-offs of using the design pattern. For example, space and time requirements, impact on flexibility, extensibility, or portability.

The essential idea of design patterns is best explained by an example. To illustrate the patterns we will use class diagrams. The class diagrams used in this paper are based on Object Model Notation, a notation for depicting the static structure of classes and objects and their relationships as proposed in OMT (Object Model Technique). In this paper the notation is explained only as far as it is used. For more information about OMT and the Object Model Notation see [RBP+91]. Figure 4 gives a brief overview of the notational elements used in this article.

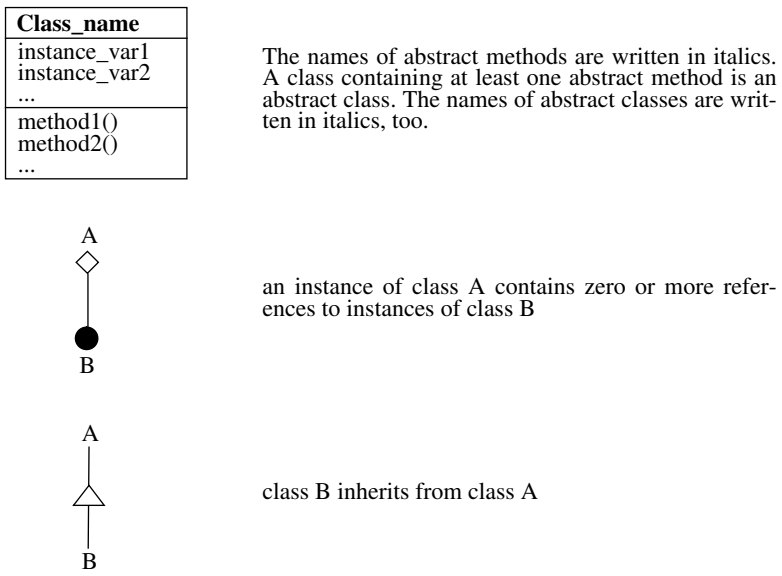

Figure 4 : Overview of the notation used in this article

Now let us have a look at a simple example. In many situations we have elementary objects and want them to be grouped in order to manage them by exactly one object. For example, there is a set of documents which we want to put into a folder. In order to calculate the size of a folder measured in pages, the folder asks each included document how many pages it comprises and adds the answered page numbers to calculate the overall folder size. In this example the folder serves as a container. This leads to the container pattern of Figure 5, where the class **Container** contains zero or more references (contained_item) to instances of another class. In our example, class **Container** represents a folder and class **Component** represents a document.

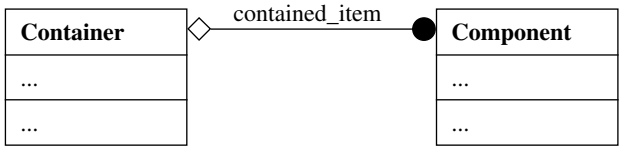

Figure 5 : Container pattern

Note that the container pattern does not imply the inheritance concept, thus it is not bound to object-orientation.

The depicted container pattern is not very flexible because all components must be instances of the same class. Now we want the container to be able to deal with elements of different classes, and it should be possible that an element of a container is a container instance itself. Each element should be treated the same.

In our folder example this means that a folder may contain documents or folders, i.e. a container references zero or more components and a component may be a single component (document) or a container (folder). Now, in order to find out the size of a folder, the folder asks each included element which size it has, regardless of whether it is a document or a folder, and adds up the sizes given. The folder does not have to know which element is a folder and which is not, because each element has to provide a method with a predefined interface that delivers the element size. This allows the treatment of a container object as a single component object. As a consequence, hierarchies of component objects can be built. This leads to the composite pattern depicted in Figure 6.

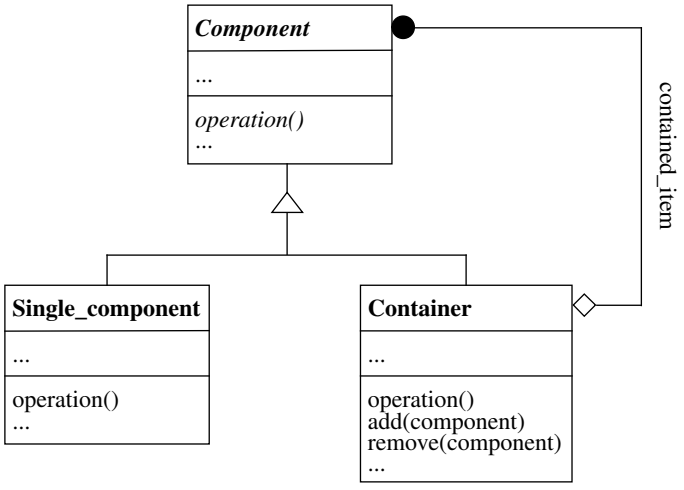

Figure 6 : Composite pattern

Class ***Component*** provides an abstract method called *operation()* (in our example this would be the method that computes the object's own size). Since the classes **Single_component** and **Container** are heirs of ***Component***, both classes have to implement operation(). In addition, **Container** at least has to implement methods to add and remove a contained_item, i.e. a reference to an instance of an heir of ***Component*** (for a detailed description of the composite pattern see [GHJ+95] and [Pre95]).

Another example of an application of the composite pattern is a graphical editor. It allows to build diagrams out of components like lines and texts. The user may group components to form larger components, which in turn may be grouped to form still larger ones. In this example there are two kinds of single components: lines and text. So instead of class **Single_component** there are the two classes **Line** and **Text**. Both classes as well as the **Container** class have to implement the method operation(). Let us assume this method draws something onto the screen. In class **Line** it draws a line, in class **Text** it writes a text onto the screen and in class **Container** it calls the drawing method for each contained_item. Using the composite pattern, the caller of the drawing method does not have to know whether the call refers to a single or a container object.

The composite pattern depicted above is not only suitable for solving the problem of how to handle recursive folder structures or graphic objects as considered in the examples, but it also describes how to deal with an object hierarchy, where each instance has to be treated the same regardless of whether it is a container or a primitive object.

This leads to the conclusion that a pattern describes the basic idea behind the solution of a particular design problem, but it does not provide a solution to the application in full detail. Each instantiation of a design pattern looks different because of the different contexts in which it is used.

3.3.2 Classification of Design Patterns

In order to reuse a design pattern, it is necessary to know that a suitable design pattern exists and how to find it. Therefore, a classification scheme would be helpful as a guide when searching for an appropriate pattern for a given situation. There are several different classification schemes, but there is no standardized way of classifying design patterns. To show this fact we present three proposals for characterizing design patterns found in literature.

In [GHJ+95], design patterns are classified by two criteria: purpose and scope. The purpose reflects what the pattern does and the scope specifies whether the design pattern applies primarily to classes or objects. Both criteria are orthogonal and can be subdivided. Table 3 gives an overview of the classification system.

Purpose	Creational	concern the process of object creation
	Structural	deal with the composition of classes or objects
	Behavioral	characterize the way in which classes or instances interact and distribute responsibility.
Scope	Class	describe the relationships between classes and their subclasses (static relationships)
	Object	describe relationships between objects (dynamic relationships)

Table 3. Classification system used in [GHJ+95]

Another possibility of classification is found in [Gam92], where design patterns are distinguished according to their application:

- structuring and organization of class hierarchies
- organization of class interactions
- organization of class interfaces
- evolutionary improvement of class hierarchies

Yet another classification scheme is found in [BM95]. Besides the classification by three levels of granularity (architectural frameworks, design patterns, and idioms) there are two other categories: functionality and structural principles. The category of functionality distinguishes between the creation of objects, guiding communication between objects, access to objects, and organizing the computation. The structural principles can be subdivided into abstraction, encapsulation, separation of concerns, as well as coupling and cohesion. Granularity, functionality, and structural principles form a three-dimensional classification scheme.

All classification schemes described above share one insufficiency: the assignment of a design pattern to a category is not definitely clear. Therefore, it is useful to list the pattern in each category indicating its properties.

3.3.3 How to Describe Design Patterns

The design idea represented by a specific design pattern can only be reused if it is found and recognized as the solution of the problem to be solved. Therefore, we must have a look at the question of how to describe design patterns. [GHJ+95] provides a template for the description of design patterns. It comprises:

- the pattern's name and classification
- intent
- motivation in form of a scenario that illustrates the design problem and how the design pattern solves it
- applicability that states in which situations this design pattern can be used
- a graphical representation of the structure
- participants
- collaboration
- consequences
- hints for implementation and sample code
- references to concrete examples
- references to related patterns

The template proposed in [BM95] is a very similar one. The main differences are the additional description of the dynamic behaviour of the pattern in form of a scenario-based illustration and the description of the methodological steps for constructing the pattern.

3.3.4 Conclusion

The purpose of the design pattern approach is the reuse of design ideas. Design patterns describe the basic structure and behaviour of the elements that are part of a solution of a particular design problem, but they do not provide the solution itself in full detail. Although the implementation of design patterns is not bound to object-oriented programming languages (e.g.

container pattern), these languages support the implementation in an elegant way through the concept of inheritance (e.g. composite pattern). To reuse a design pattern, you first have to know that an appropriate pattern exists and then how to find it. Therefore, classification schemes are helpful, but currently there is no standard. To identify the fitting pattern, an exact description is needed.

3.4 Frameworks

The term framework is used in many different contexts. This is because the term framework means nothing else than skeleton or frame. The term is used in all cases where one addresses the structure of something. This may be the skeleton, for example, of a ship or an aircraft which makes up its shape especially in the process of building, or the structure and management organization of a team of development engineers who have to solve a certain complex problem. In the environment of object-oriented software development, the term framework exists as well, and again it deals with that part of the structure that makes up the construction of the software to be developed.

Johnson and Foote [JF88] define a framework as follows: *'A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and support reuse at a larger granularity than classes.'* The substantial idea of frameworks is that they do not deal with an accumulation of independent classes (like a class library), but already define a design for the solution to a specific problem. This is achieved by determining not only the classes that are part of the solution but especially the interactions among the classes. Therefore, the concept of frameworks is very similar to that of design patterns, but frameworks are implemented in a programming language so they are more concrete than design patterns. Mature frameworks reuse several design patterns.

Substantial components of frameworks are abstract classes. This kind of class contains abstract methods which the development engineer has to substantiate in derived classes. In such a way the framework can be adapted to special requirements. This allows the reuse of code and design.

Frameworks can be divided into frameworks in the large and frameworks in the small:

- Frameworks in the small offer a solution to a specific problem. In most cases they consist of only a few interacting classes and represent a micro-architecture of a few components. Small frameworks are closely related to design patterns that are described in Section 3.3. As we will see in the next section, those design patterns that contain abstract classes support the framework approach best.
- Frameworks in the large are generic applications and, therefore, they are called *application frameworks*. The idea behind such frameworks is that software development often does not deal with single applications but with a family of applications with a common basic functionality. The framework implements common functionality and mature flow of control. The concrete application is achieved by adapting the framework to one's own functional requirements by implementing derivations of abstract classes.

In many cases it is difficult to determine which category a framework belongs to, because it is not possible to draw a sharp line between the two categories. The concepts behind both kinds of frameworks are the same. Therefore, we will not make any distinction between them in the next section.

3.4.1 Fundamental Ideas of Frameworks

The primary goal of the framework approach is the reuse of code and design. Class libraries support the reuse of code on a low level. Connections between classes have to be designed each time a development engineer uses the classes. This is not true for frameworks, because the structure and flow of control are pre-built in framework classes.

An application framework defines the structure of a complete application and consists of ready-to-use and semi-finished building blocks. Application frameworks are often called generic applications because they implement the mature flow of control of an application, but they do not include application-specific functionality. Adaptation to the specific application is provided by overriding abstract methods. The flow of control of an application framework is fixed in the framework classes. This leads to a reuse of code and design.

The difficulty of designing a framework is how to identify those aspects of an application domain that have to be kept flexible, because they vary from application to application. In [Pre95], these aspects are called hot spots, while the aspects that are not designed for adaptation are called frozen spots.

Implementation of frameworks relies heavily on the dynamic binding of methods. In the context of frameworks, two kinds of methods are to be distinguished [Pre95]:

- template methods: they call at least one other method
- hook methods: they could be abstract or concrete methods or template methods itself

Figure 7 illustrates the concept of template and hook methods by an example.

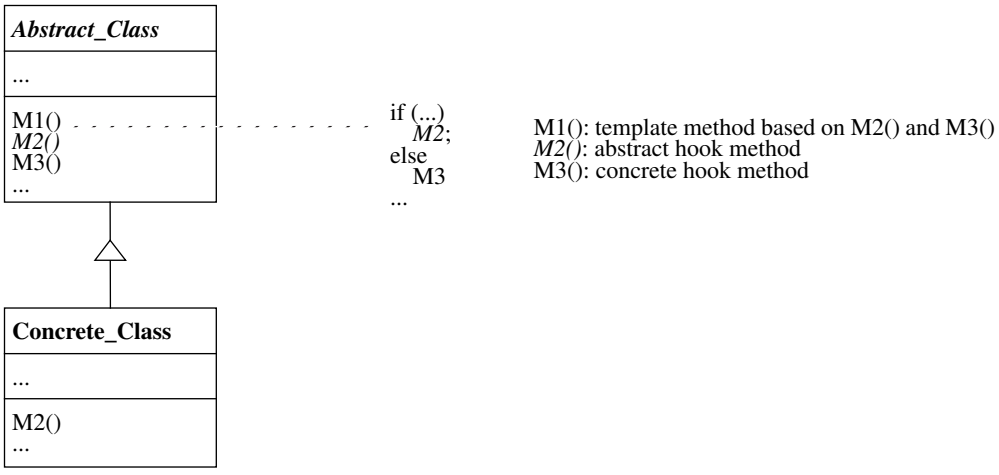

Figure 7 : Example of template and hook methods

Abstract_class includes an abstract method *M2*, therefore it is not possible to create an instance of that class. **Concrete_class** is an heir of ***Abstract_class*** and inherits method M1 and M3 and the interface specification of *M2*. In **Concrete_class**, the behaviour of M1 is adapted to specific needs only by implementing M2 and possibly reimplementing M3, but not touching the program text of M1. This leads to the following conclusion: template methods represent the frozen spots and define the abstract behaviour or the generic flow of control, or the relationship between objects. Hook methods are designed to be adapted to specific needs and implement the hot spots of an application framework. As a consequence, when using a framework,

programmers do not have to care about the connection of classes, but may fully concentrate on the application. That means they only have to implement the hook methods which are called by the framework.

In some situations it is necessary to change the behaviour of a template method by providing a different hook method at runtime. In the example above, ***Abstract_class*** contains both, the template method M1 and the hook methods *M2* and M3. In order to change the behaviour of the template method M1, the hook methods *M2* and M3, respectively, have to be reimplemented in a subclass. This is the reason why the template method cannot be changed at runtime. To solve this problem, template and hook methods are separated into different classes. The template class contains a reference to the hook class. At runtime, an instance of a hook class can be created and associated with an instance of a template class.

Pree [Pre95] introduced the term *metapatterns* for those patterns that capture and communicate the design of frameworks. Pree presents seven metapatterns that describe how to compose template and hook classes. The composition is influenced by the following questions:

- May one instance of a template class refer to exactly one instance of a hook class or to an arbitrary number of instances?
- Is there an inheritance relationship between template and hook classes?
- Are the classes abstract or concrete classes?

Nearly all metapatterns imply a typical structure of template methods. In [Pre95], these impacts on template methods are described, and a guide is given when to choose a certain metapattern.

The development of a new framework differs significantly from conventional software development. A framework arises bottom-up through several steps of iteration. First the domain-specific hot spots have to be found. To do this, thorough knowledge of the application domain is necessary. In the next step, the programmer designs the template and hook classes and their interaction, using several metapatterns. The next time the framework is reused to develop a new application, the programmer has to adapt the hook methods through inheritance. If this is not possible, because a template method appears to be too rigid, the template method must be redesigned, using additional hook methods. If, on the other hand, the programmer must implement similar flow of control repeatedly while adapting a framework, additional template methods should be defined. In this way, the experiences gained using a framework permanently flow back into the framework design.

By using of frameworks for the development of applications, a high level of reuse of code and design is reached. The programmer only has to implement the hot spots of the application and does not have to think about how to structure the application system, because the structure and mature flow of control are already decided. This means the software architecture together with the implementation of the control parts of the software system are already available. This leads to lower cost and a shorter development period.

The idea of [Sch96] goes one step further. In this article, frameworks are classified according to the way in which an application is created:

- **White-box framework:** using a white-box framework, the application is created as described above. That is, the developer has to implement the hot spots by deriving from abstract classes.

- **Black-box framework:** this kind of framework contains for each hot spot a set of alternative classes. The developer of an application selects a fitting alternative for each hot spot and thus configurates the framework.

Obviously, the development of a black-box framework is more expensive than that of a white-box framework, because all alternatives of a hot spot have to be provided. The spectrum between white-box and black-box frameworks may be continuous. The frequently used alternatives of a hot spot may be predefined and used in black-box mode, whereas the others, especially the unforeseen cases, have to be created in white-box mode. The more hot spots may be instantiated in black-box mode, the easier is the use of the framework and the higher is the degree of reuse of code and design.

3.4.2 Examples of Application Frameworks

ET++ is an extensive C++ class library together with an application framework for GUIs. ET++ was developed at the University of Zurich and the Union Bank of Switzerland Informatics Laboratory by Erich Gamma and Andre Weinand [Gam92].

Another example are Taligent Frameworks. Until the end of 1995, Taligent, Inc., was an independent joint venture of Apple Computer, Inc., IBM Corporation, and Hewlett Packard Company. Taligent intended to develop application-level frameworks (like GUI), system-level frameworks, and development environment-level frameworks (like debuggers, browsers, and other tools). Now Taligent is part of the IBM software development laboratories. In this new role, Taligent's product direction is shifting from the delivery of a complete programming model to integrating Taligent's framework technology into IBM products. Taligent offers an Internet home page [Tal96] with a variety of information about their framework approach.

3.4.3 Conclusion

The primary motivation for the framework approach was a high level of reuse of code and design. The benefits of using frameworks cannot be enjoyed immediately. The programmer who wants to adapt a framework has to know a lot about the structure and the design of the given framework, which requires a great deal of learning. The more flexible a framework is, the more difficult is it to learn. In addition, a framework is developed through steps of iteration. As a consequence, the benefits of reuse (like lower costs and shorter development periods) are long-term benefits, not immediate ones.

If a matured framework for an application domain is available, the software engineer can reuse the architecture of the software system as well as substantial parts of the implementation. The engineer can adapt the framework to special requirements by only implementing new functionality and integrating it at the proposed hot spots (white-box framework) or by selecting already implemented alternatives for the hot spots (black-box framework). In the context of frameworks, software architecture means the structure of software on an abstraction level close to implementation.

3.5 Domain-Specific Software Architectures (DSSAs)

The approach of domain-specific software architectures is based on the following assumption [CT92]: '*Development of systems in a well understood domain is a process of matching requirements to known approaches to providing solutions in that domain.*' Thus one expects that within a specific application domain (which means restricted problem and possibly restricted solution space), different developments of software systems are quite similar. Development activities that are concerned with special requirements of a single system could be isolated from others that are essentially the same. This would result in reusing many developed artefacts and process knowledge all over the software engineering process in that domain.

The most famous example of DSSAs is the *Avionics Domain Application Generation Environment, (ADAGE)* (see [CTB+93]) which covers nearly the whole development process for the domain of avionics. Even code generation features are integrated.

In literature, different definitions of DSSAs with different central concerns are to be found. [Hay94] for example emphasizes the software artefact itself, which naturally incorporates previous engineering steps (architecture, code generation): '*A DSSA is an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across the domain, composed in a standardized structure (topology) effective for building successful applications.*' [Hid90] places the early architectural specification in the centre of interest: '*A DSSA is a context for patterns of problem elements, solution elements, and situations that define mappings between them.*'

To summarize, the decisive properties of DSSAs are:

- a restricted problem and/or solution domain
- genericity, in order to be able to adapt a DSSA to a special development
- suitable level of abstraction to cover the whole domain
- fixed reusable elements of the development process that are typical of the respective domain

Having understood the importance and utility of DSSAs, the question is how to find one. It may be stated that it has to be done by very experienced people who know the special application domain very well, which means that they have to know both the problem and the solution space. They have to find the appropriate abstractions to achieve genericity and reusability. We think that in general DSSAs can be developed in an evolutionary way, always searching for better abstractions all over the development process. This can be done simply by working with the DSSAs, gaining more and more experience and validating the existing DSSAs.

Domain architectures are found in a wide variety of application domains. There exist domain architectures for such different application domains as:

- Business Information Systems
- User Interface Management Systems (UIMS)
- Compilers
- Artificial Intelligence Systems (Blackboards etc.)
- Movement Control Systems
- Robotics

Domain architectures may be composed hierarchically. A business information system will typically include a user interface management system that is constructed conforming to a specialized domain architecture for User Interface Management Systems.

The following sections deal with DSSAs for business information systems.

3.5.1 Domain Architectures for Business Information Systems

Most Business Information Systems (IS) are rated as simple, database driven information systems. Table 4 gives some characteristics of this application domain in relation to other application domains.

	Engineering Systems	**Technical and Advanced Systems**	**Business Information Systems**
Example Applications	- Engineering - Scientific - Design Automation - Simulation	- Groupware - Multimedia - Network Administration - Imaging - Document Management	- Finance - Order Processing - Accounting
Characteristics	- Complex - Object Oriented - Single User or Small Workgroups	- Mission Critical - Database Oriented - Larger Workgroups	- Mission Critical - Database Oriented - ACID Transactions - Simple

Table 4. Examples of Business Application Domains (adapted from [Wad94])

Even in the IS domain there exists a wide variety of different domain architectures depending on the particular kind of business the architecture is intended for, e.g. banking, financial applications, insurance applications. These architectures differ less in their general architectural approach - usually a three layer Seeheim model [Pfa83] - than in their inclusion of domain-specific knowledge of the specific business domain.

Many application architectures in the business field consist of two parts:

- a reference model that is a conceptual business model for an ideal enterprise in the specific business domain for which the model is intended.
- a domain-specific software architecture for simple, database driven business information systems. The respective software architectures are rather similar in most cases.

Reference models are conceptual models of certain business domains. They specify business processes, business objects, and business requirements. Reference models deal with functional requirements of some ideal enterprise. Reference models are often not clearly separated from domain architectures. The term domain architecture is often used where the term reference model would be more appropriate.

FAA (Financial Application Architecture [IBM92]) and IAA (Insurance Application Architecture [IBM91]) are two examples of a whole set of IBM reference models for different fields of business. They are mostly based on Zachman's work on information system architecture (see [Zac87], [ZS92], [ÖS94]). Thus the term 'application architecture' is

somewhat misleading. Reference data model would be a more appropriate term. Reference models like the above may be seen as a means of exchanging information about the functional requirements in a certain business domain. They are not intended to lead to standard software products or reusable components in the short term. The focus of these models is less on software architecture but more on the reference model side, and there especially on the data model side. As a consequence we do not talk about FAA or IAA in the following.

Instead we present some examples of domain architectures for business information systems. We discuss two general architectures for simple information systems: the sd&m[1] standard architecture and the 'Anwendungsarchitektur der Versicherungswirtschaft' (VAA), an Insurance Application Architecture initiated by a group of German insurance companies.

3.5.2 sd&m Standard Architecture

As an example of a more general domain-specific software architecture we will present the sd&m standard architecture for business information systems (see [Den91]). The architecture is used for mainstream information systems. In the following we will use an object-oriented variant of the architecture. The architecture does not contain any business domain-specific knowledge.

The sd&m standard architecture is a refinement of the three layer Seeheim model. Similar software architectures can be found in the whole domain of simple, database driven business information systems. The architecture follows the component model of software architecture - in other refined views of the architecture connectors are also shown. Connectors are defined as possible call relations.

The sd&m architecture consists of three layers:

- user interface
- application kernel
- database (represented as instance data)

The database layer is not explicitly depicted in Figure 8. We assume that application kernel objects are persistent. Persistence can be implemented by various means - one possible storage system to implement persistence is a relational database. Other storage systems like OODBMS or flat files are also feasible.

Basic functionality is being used across layers and denoted as common services. The sum of common services is not explicitly depicted in Figure 8. Common services include e.g. exception handling, data types, and security.

If the architecture is intended to be used for client/server systems, it has to be refined with a client/server cut and some middleware concept.

[1] software design & management (sd&m) GmbH & Co. KG, München

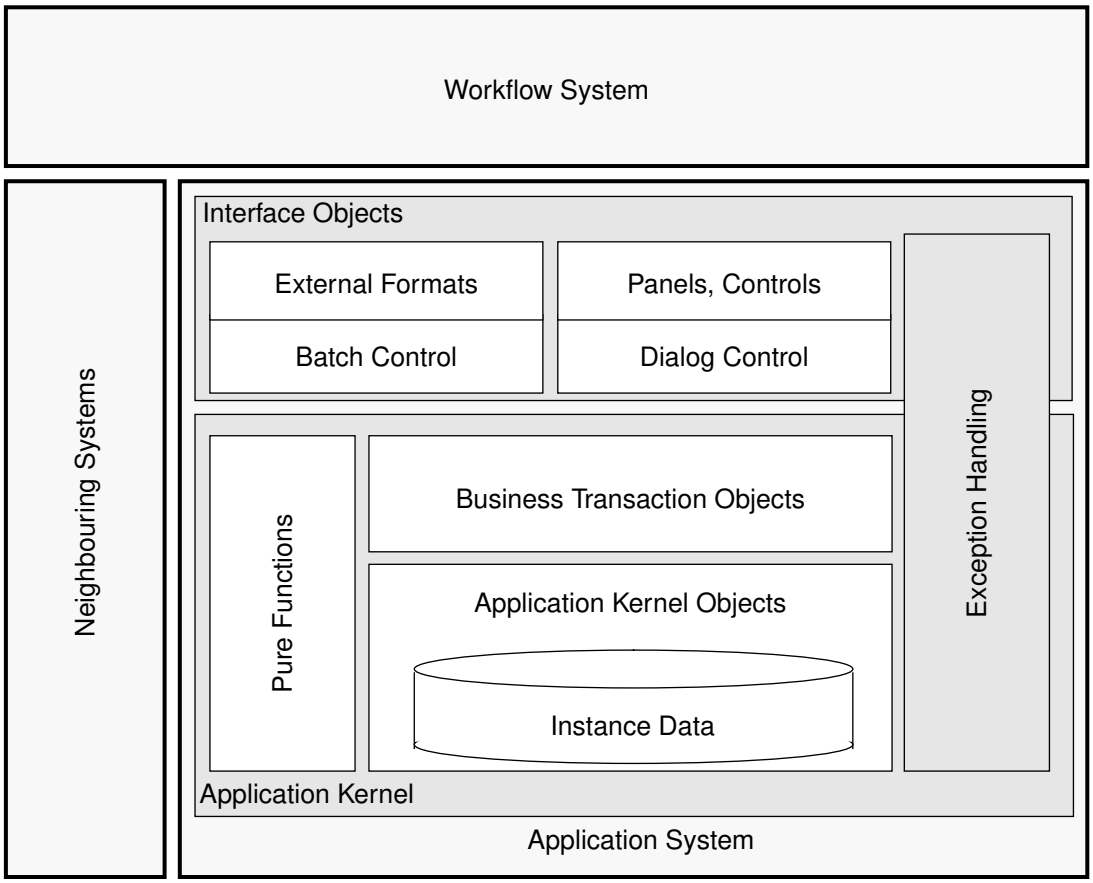

Figure 8 : sd&m Standard Architecture

The following paragraphs contain a brief description of the architecture shown above. First we will describe the three large blocks: workflow system, neighbouring systems, and application system.

Workflow System

The workflow system is a specialized application system that implements workflow metaphors like business processes, process steps, folders and documents flowing through an organization, hierarchy and delegation, and many more.

The workflow system acts as glue or as a giant spider above all application systems of an enterprise, integrating all these applications. If there is an atomic process step to be performed, the work can be done manually by a clerk or supported by one of the application systems. In this sense, the workflow system is yet another application system with its own architecture.

Neighbouring Systems

Neighbouring systems are other application software systems that may have the same architecture as the application system of interest. If the neighbouring system is well designed, it should have some interface classes that export services to other application

systems. These interface classes may be called by the application kernel objects of an application system.

Application System

Application systems consist of an interface and a set of application kernel objects that are manipulated via the interface. Application kernel objects are made persistent using a relational database or some other form of database.

Now we will have a closer look at the components of the application system. These are, as mentioned above, the user interface, the common services (e.g. exception handling), and the application kernel.

Interface Objects

Interface objects constitute the user interface to the application kernel objects. There are two suites of interfaces.

- The **dialog interface** allows manipulation of application kernel objects via dialogs (GUI or terminal type dialogs). The interface objects are typically partitioned into presentation objects (panels, controls) and dialog control objects that manage the flow of control in a dialog.
- The **batch interface** allows manipulation of application kernel objects via batch processes. Batch processing is often neglected. This is why batch processing is an explicit part of the sd&m standard architecture. Batch objects are also partitioned into two layers. External formats denote list, files, etc. that are generated by batches or used as batch input. The batch control constitutes the heart of a batch program.

Exception Handling (Common Services)

Exception handling is one common service provided by the standard architecture. The box shall give the impression of an elevator. Exceptions ride upwards in this elevator until an upper layer can treat them or until the interface layer has to make them visible to the user as no layer was able to treat them. In practice, exception handling consists of several exception classes and protocol classes plus a component that maps exceptions to some meaningful sort of message for the user of a system.

Application Kernel

The application kernel consists of several components. These components are sets of similar objects. Their subcomponents are grouped by structural similarity - not by functional requirements - at least in the architecture.

- **Business Transaction Objects**: it may be observed that one meaningful atomic step of a business process (business process step) or even of a dialog (dialog step) requires several dependent actions from business objects (application kernel objects) that can be seen as one transaction. Business transaction objects are used to control such a sequence of actions by business objects and treat these actions as one atomic transaction. Business transaction objects typically contain no persistent data.

- **Application Kernel Objects**: application kernel objects model the core of the business model of the enterprise. They are usually derived relatively straightforward from the object-oriented analysis model of the business. If the application kernel consists of more than some 20-30 application kernel object types, it makes sense to apply some form of clustering or subsystem formation to the application kernel.
- **Pure Functions**: practical information systems, especially in the financial system domain, contain a lot of (mathematical) functions. These functions could be modelled as objects - but this is seldom natural. Therefore they are grouped as pure functions. Functions typically have a narrow interface, perform a lot of complex calculations, and do not possess any persistent data. A typical example is a function for calculating the effective interest rate, given an array of payments.

3.5.3 Application Architecture for Insurance Companies (VAA)

VAA, the 'Anwendungsarchitektur der Versicherungswirtschaft' [VAA95], is an ongoing activity that has been initiated by a number of German insurance companies. It was started with the experience in mind that IAA will not lead to a market for standard software components that can easily be installed by a single insurance company.

The long-term goal is to set up a market for reusable software components for the insurance business.

Therefore VAA consists of two parts:

- a technical architecture that is intended to be developed to become a technical framework for insurance applications.
- a set of application specifications (reference models) consisting of business process, data, and function models for core insurance applications like claims processing, car insurance, partner systems, and others.

We will not discuss the application specifications but concentrate on the technical architecture. The VAA technical architecture (see Figure 9) is again based on the three layer Seeheim model. The technical principles are similar to the sd&m standard architecture of Section 3.5.2, but there are several major differences:

- VAA is an architecture for insurance companies. Therefore, certain technical concepts that are specific for insurance applications are incorporated into the architecture. Examples of such components are a security system or history concepts for enterprise data.
- VAA does not yet use object-oriented terminology but is more based on structured analysis and design methods.
- Presentation services and dialogs are integrated into the workflow-layer on top of the architecture.
- The common services contain more services - especially insurance-specific basic services.

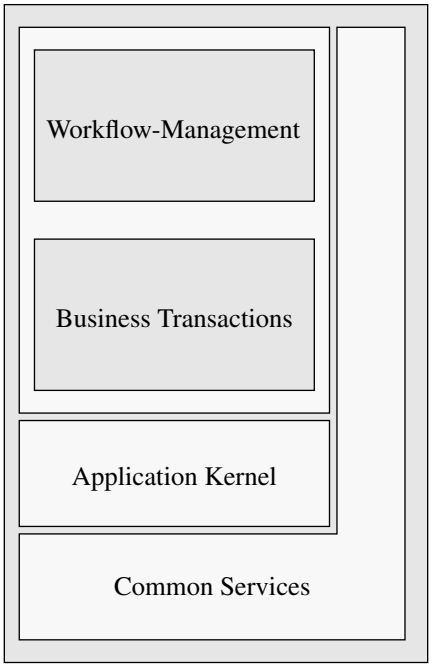

Figure 9 : VAA - Technical Architecture

3.5.4 Conclusion

As we have seen, different DSSAs exist nowadays. Most of the big companies try to excerpt their recent development know-how and incorporate it into a standard architecture most of their projects can be designed with.

VAA is still in a state of work in progress. There is a lot of work to be done. The current state of this architecture is a reference model for implementing architectures for the respective insurance companies and not a framework. The combination of both a technical and a business reference model can also be called a DSSA (Domain-Specific Software Architecture). The idea to create an open market for application components for a business area is ambitious and fascinating. The future will show whether this approach will be successful.

3.6 Technical Standard Architectures

The domain of technical standard architectures is that of technical services which build a high-level interface to the underlying operating system and all kinds of hardware. This interface (in general there exist different interfaces) is designed to reach the following goals:

- providing a common platform/environment for application development
- easing application development by abstractions, reuse, and tools (the services should hide all technical subtleties from the application programmer)
- increasing portability of applications
- protecting applications from changes in the underlying hardware and technology
- adaptability (to new technologies, extensions)
- uniform look and feel across all applications
- faster development

Technical architectures evolved in the eighties. Today the importance of technical architectures is still increasing. This is not surprising, because a lot of technologies and products for networking, interoperability, and client/server computing push the market. Thus there is a need for standard architectures, making possible integration of different products and technologies.

To sketch out an architecture for a software system, you have to consider both the application architecture and the technical architecture. The technical architecture influences main aspects of a software system, such as:

- performance
- possible distribution of the software system
- characteristics of the user interface
- portability
- security

All application architectures previously described are based on the assumption that there exists some environment on which an application can be built. The structure, services, interfaces, and protocols of this environment are described by a technical architecture. For example, the ISO-OSI reference model is a technical architecture. For a basic infrastructure which covers all aspects needed for application development, a broad architecture is necessary where the ISO-OSI model may be only one component.

In the following sections we present two representatives of technical architectures, namely SAA and Open Blueprint, both developed by IBM. There are also some other technical architectures, but we have chosen the IBM architectures because

- they are the most well-known examples of technical standard architectures,
- they are sufficient to illustrate the concept of technical architectures.

3.6.1 Systems Application Architecture (SAA)

The ***S**ystems **A**pplication **A**rchitecture* (SAA) is an architecture defined by IBM. It was first announced in March 1987. The early vision of SAA is described in [WG88]. This article is only one of many in a special IBM Systems Journal about SAA. Throughout the years the specifications of SAA have changed, and today SAA is a bit outdated. The new vision of IBM on distributed computing is now described in an architecture called Open Blueprint. This architecture, which can be seen as a successor to SAA, is treated in the following Section 3.6.2. Nevertheless it is worthwhile to have a look at the concepts of SAA because Open Blueprint has evolved from it.

It is important to note that SAA does not deal with the interior architecture of concrete applications. Thus the term SAA is a little misleading. The scope of SAA is to define standardized interfaces which serve as a common base for the development of applications and encapsulate the functionality of the underlying operating system. Within SAA, the architecture of an application is a black box, whereas application architectures like VAA (see Section 3.5.3) are concerned with the structure of this black box for a specific application domain. SAA defines basic services which can be used by every application.

In early operating systems such services were very low-level. However, these services have become more and more abstract and their functionality has increased. Today high-level

languages and graphical CASE-Tools support application development. Examples of applications which are based on SAA are AD/Cycle and Office Vision from IBM.

What is the structure of SAA? Roughly speaking, SAA is a collection of selected software interfaces, conventions, and protocols. It defines a framework for the development of integrated applications. SAA unifies various application environments across different product families to achieve cross system compatibility and to provide support for distributed processing and distributed data. The concept is visualized in the following diagram, which has become the official icon (also called 'the fireplace') of SAA and can be found in nearly every book on SAA.

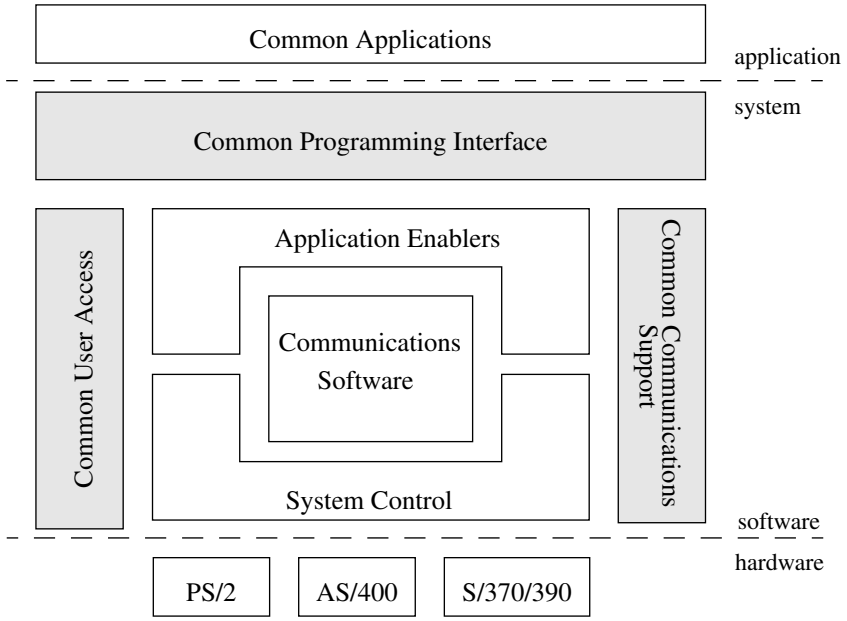

Figure 10 : The Systems Application Architecture

The Systems Application Architecture consists of three standardized interfaces (grey-shaded boxes in Figure 10) that are described in the remaining part of this section:

- **C**ommon **U**ser **A**ccess (CUA)
- **C**ommon **C**ommunication **S**upport (CCS)
- **C**ommon **P**rogramming **I**nterface (CPI)

As one can see in Figure 10, these three components encapsulate various hardware/software platforms in order to give them a uniform look to the outside world: CUA defines the interface for the system users, CPI defines the interface for the programmer writing SAA applications, and CCS defines the interface to other machines.

The interfaces are built around a three-part core called the software base. This software base is a layered architecture that describes the general software structure of an operating system. The bottom layer is the software which is related to the hardware and manages the physical system resources. In the centre there is the communications software, which entails communication protocols and network management. The top layer represents the software for writing and executing applications. This includes services such as programming languages, applications generators, databases and transaction management, data presentation, and dialog management.

Common User Access (CUA)

CUA defines rules and guidelines for the design of (graphical) user interfaces, so that every application which conforms to this standard offers a consistent view to the application users. This means that every application has the same look and feel on different computing environments. Consistency has three dimensions:

- *physical consistency*: keyboard layout, placement of keys, etc.
- *syntactic consistency*: visible part of the interface, appearance of items on the screen (look), sequences of user requests (feel)
- *semantics*: meaning of elements in the interface

Common Programming Interface (CPI)

The CPI represents the developer's view of a SAA environment. It is a standardized API, which is available on all IBM-operating systems and is used to develop applications. It has two layers: languages (e.g. REXX, C, Cobol, Fortran) and services (e.g. database services, dialog services, presentation services). The main objectives of CPI are:

- providing consistency for the end users (through the dialog and presentation interface),
- more productivity on the part of application developers,
- supporting distributed processing,
- an enterprise-wide application development environment.

Common Communication Support (CCS)

CCS specifies a set of protocols (about thirty separate protocols) for interconnection and communication among SAA-systems. It is primarily based on elements of SNA (**S**ystem **N**etwork **A**rchitecture). The services of CCS are accessible for applications through the CPI. Like the ISO-OSI model, CCS consists of different layers.

3.6.2 Open Blueprint

Open Blueprint [IBM95] is a kind of a new SAA. It describes a common infrastructure for distributed systems. Like other architectures it is illustrated by a diagram (see Figure 11), with different layers of the architecture and different components within each layer.

In contrast to SAA, Open Blueprint integrates new technologies (e.g. object-oriented technology) and new standards which are not as proprietary as the standards within SAA. Therefore Open Blueprint supports the interoperability of applications from different vendors, whereas SAA mainly concentrates on the (application-) consistency across different IBM-platforms.

For an application (or user) there is only the view of one net-system. The application (or user) does not know that there is a heterogeneous network of systems. For some of the components, the diagram additionally lists the different protocols, standards, or models which are supported by the specific component. But the diagram has no precise semantics. It is only an illustration of the different components and their level of abstraction. It contains no statement about the real relationships between these components. For example, the position of the Data Access Services above the Distribution Services does not mean that functions of the Data Access Services can only call functions of the Distribution Services.

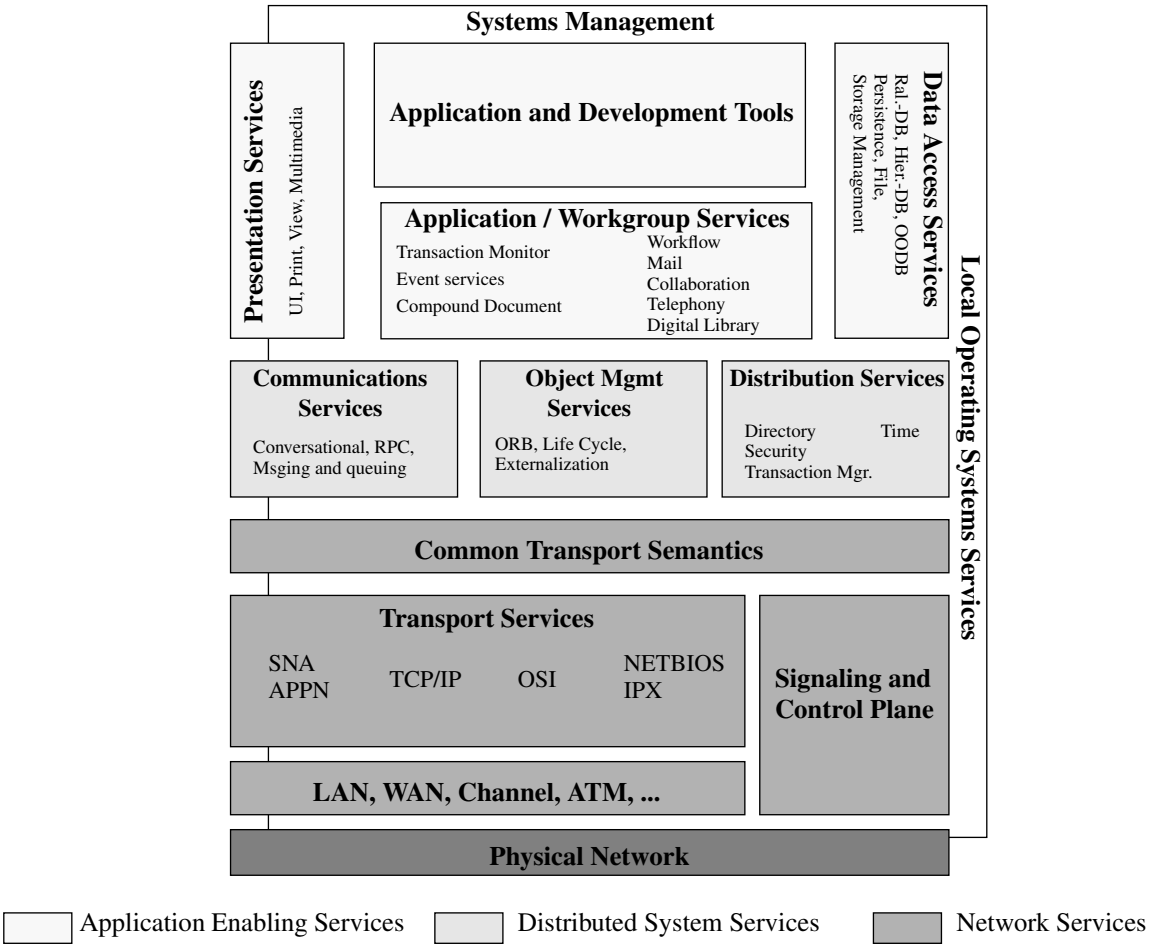

Figure 11 : Open Blueprint components [IBM95]

As you can see from Figure 11, Open Blueprint not only deals with the infrastructure of the execution environment (e.g., local operating system services like memory management), but also with the infrastructure for application development, tools, and system management. The architecture specifies and details the following service groups:

Network Services

- *Common Transport Semantics* unifies the different communication protocols which are used by the transport services.
- *Transport Services* support different communication protocols (e.g. SNA/APPN, TCP/IP, OSI).
- *Signalling and Control Plane* enables subnetworking connections and is based on the ISDN Control Plane.
- *Subnetworking* contains services for different kinds of networks (e.g. LAN, WAN, ATM).

Distributed System Services

- *Communication Services* are needed for any communication between distributed applications. Open Blueprint supports three models, which specify how components can communicate with each other: Messaging and Queuing, Conversational, and Remote Procedure Call.
- *Object Management Services* are common object services (for example the services of an Object Request Broker).
- *Distribution Services* comprise different common services which support a single system view of the network (e.g. Directory and Naming Service, Security Service).

Application Enabling Services

- *Presentation Services* deal with the interaction between applications and users.
- *Application/Workgroup Services* are common higher-level services like workflow management and electronic mail.
- *Data Access Services* support applications to read and write data.

3.6.3 Conclusion

Concept and motivation for technical standard architectures do not differ very much from domain-specific standard architectures. The technical architectures also try to partition functionality into different components which are organized in a couple of abstraction layers. The examples show that these architectures can be very broad, consisting of components which themselves can be seen as subarchitectures (e.g. Open Blueprint contains CORBA, which defines a broker architecture). Other typical components are certain standardized protocols. In contrast to domain-specific architectures, standards for many components of technical architectures already exist, because technical architectures depend on technologies and products from different vendors. Thus there is even more interest and pressure for standardization and stable architectures.

4 Conclusion and Outlook

The complexity of modern software systems is increasing. Distributed applications are becoming more and more popular. To manage these new challenges, the whole software development process has to be improved. One very important aspect supporting this is dealing with the structuring of software, that is to say, its architecture.

As we have seen in Chapter 2, the software engineering process provides multiple models of software in each of the development phases. These models include architectural models that describe the overall system structure.

There are two different approaches in the field of software architecture: formal attempts by the basic research community and pragmatic attempts by the software development industry.

Efforts by basic research concentrate on a formal description of the structure of software systems under development. There are two ideas that base on the definition of software architecture as a cooperation of components and connectors on different levels of abstraction:

- **Architectural Styles:** they formalize the properties of different kinds of components and connectors. Since most of the proposed styles are actually not based on formal semantics, a lot of work has to be done.
- **Architectural Description Languages:** they provide different description techniques with different syntax and semantics. Most of the languages and the supporting tools are not yet in a state ready for public use.

Present industrial approaches are pragmatic. Their primary goal is the reuse of already known solutions. These may be general design concepts, particular design ideas, or even code already available. The reuse approach can be summarized as follows:

- **Patterns:** they focus on the reuse of design ideas. These may vary from general concepts for the overall structure of a software system (architectural frameworks) to particular design ideas (design patterns) and how to implement them (idioms).
- **Frameworks:** they go one step further by reusing design and implemented code. This code is designed to be adaptable to specific needs. The size of frameworks may vary from small building-blocks to complete adaptable software systems (application frameworks).
- **Domain-Specific Software Architectures (DSSAs):** they deal with the structure of complete software systems, thus they are quite similar to application frameworks, but they differ in their level of reuse. DSSAs focus on the reuse of informal concepts and designs. They do not deal with the reuse of code.
- **Technical Standard Architectures:** they form a base for application architectures by providing a unique development environment. They build common technical services and high-level interfaces to the underlying operating system and they provide protocols for different environments and hardware platforms. This supports portability of applications in different hardware and software environments.

At first sight, attempts by the basic research community have nothing in common with those by the software development industry. However, they share a common goal. Both try to write down the structure of software in order to reuse incorporated design ideas, but the means differ. The

informal descriptions of patterns, frameworks, DSSAs, and technical standard architectures often lack precision. To make an end to interpretation, formal (but easy to handle) description techniques would be helpful. Pragmatic approaches concentrate on the question of what is a good software architecture, i. e., what to write down, whereas the formal approaches focus on the question of how to write it down. Thus both approaches could complement each other.

Acknowledgements

Our special thanks go to Peter Brössler, Manfred Broy and Peter Scholz for discussions and for their review of earlier versions of this paper. They were always a valuable resource for feedback. We are also grateful to Heike Philp and Gabriele Turner, who read an earlier version of this paper and provided us with many useful comments leading to improvement.

Glossary

Application Framework

An application framework is a framework (→ Framework) that provides a generic application for a specific domain. It implements the common functionality and mature flow of control for a family of applications. It consists of ready-to-use and semi-finished building blocks that can be adapted to specific needs by object-oriented mechanisms like inheritance in combination with polymorphism and dynamic binding.

Architectural Framework

An architectural framework is a pattern (→ Pattern) that describes the overall structure of a large, complex system in terms of subsystems and the relationships between them. Architectural frameworks are comparable to architectural styles (→ Architectural Style).

Architectural Language

An architectural language is a description technique with formal syntax and semantics for the description of software architectures.

Architectural Style

An architectural style defines a certain class of architecture by

- defining a vocabulary of design elements (usually called components and connectors),
- imposing configuration constraints on these elements (i.e., “no cycles”),
- determining a semantic interpretation for the system description,
- making possible a collection of analysis that can be performed on its system.

Business Information System

A Business Information System is an application software system for the electronic administration of large volumes of business-relevant data. The aim is to support business processes and to make evaluations possible that are of great importance for management decisions. Examples of business information systems are a system for the administration of stock-keeping or a system for book-keeping.

Design Pattern

Design patterns describe essential design ideas in form of pairs of software design problems and proper solutions in a special context. The solutions are described in terms of the components (usually classes and objects) that are part of the solution, the behaviour of the components, their relationships, responsibilities, and collaborations. Furthermore, the consequences of using a particular design pattern are outlined.

Domain-Specific Software Architecture (DSSA)

A DSSA is a standard software architecture for a specific application domain that provides an organizational structure tailored to a family of applications. In most cases DSSAs include a specific reference model (→ Reference Model).

Framework

A framework is a set of classes that embodies an abstract design for a solution of a family of problems. Frameworks can be adapted to specific needs by applying object-oriented programming concepts. Framework is a generic term for

- application frameworks (→ Application Framework) and
- small frameworks, which consist of only a few interacting classes.

Idiom

An idiom is a pattern (→ Pattern) that deals with the implementation of particular components of a design pattern (→ Design Pattern) or with the implementation of the relationships between components. Idioms are often dependent on a particular programming language.

Pattern

A pattern describes a software design problem together with a scheme for its solution, consisting of the components and their relationships. [BM95] distinguish patterns on three levels of abstraction:

- architectural frameworks (→ Architectural Framework)
- design patterns (→ Design Pattern)
- idioms (→ Idiom)

Reference Model

Reference models are conceptual models for certain business domains. They often specify business processes, business objects, and business requirements for some ideal enterprise. Reference models are seldom clearly separated from DSSAs (→ Domain-Specific Software Architecture).

Software Architecture

A software architecture is the structure of a software system. It is described as a set of software components and the relationships between them. For a complete description of an architecture several models and views are needed, with each model describing one or more structural aspects on a certain level of abstraction.

Technical Standard Architecture

Technical standard architectures are independent from concrete applications and are restricted to the domain of common services like communication services, data access, or design of user interfaces. They build a high-level interface to the underlying operating system and thus make integration of different products and technologies possible.

References

[AAG95] G. Abowd, R. Allen, D. Garlan: *Formalizing Style to Understand Descriptions of Software Architecture*, Technical Report CMU-CS-95-111, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, Jan. 1995.

[AG94] R. Allen, D. Garlan: *Formal Connectors*, Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1994.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: *A Pattern Language*, Oxford University Press, New York, 1977.

[Ale79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, New York, 1979.

[Big94] T. Biggerstaff: *The Library Scaling Problem and the Limits of Concrete Component Reuse*, IEEE International Conference on Software Reuse, Nov. 1994.

[BJ94] K. Beck, R. Johnson: *Patterns Generate Architectures*, Proceeding ECOOP 1994.

[BM95] F. Buschmann, R. Meunier: *A System of Patterns*, in: J. Coplien, D. Schmidt (eds.): Pattern Languages of Program Design, Addison-Wesley, 1995.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern Oriented Software Architecture*, Wiley & Sons, 1996.

[Cop92] J. Coplien: *Advanced C++ Programming Styles and Idioms*, Addison Wesley, 1992.

[CT92] L. Coglianese, W. Tracz: *Architecture-Based Development Process Guidelines for Avionics Software*, Technical Report ADAGE-IBM-92-02, IBM Federal Systems Company, Dec. 1992.

[CTB+93] L. Coglianese, W. Tracz, D. Batory, M. Goodwin, S. Shafer, R. Smith, R. Szymanski, P. Young: *DSSA-ADAGE, Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE)*, Technical Report ADAGE-IBM-93-09, IBM Federal Sector Company, Owego, May 1993.

[Den91] E. Denert: *Software-Engineering*, Springer-Verlag Berlin Heidelberg, 1991.

[Der91] J. McDermid: *In Praise of Architects*, Information and software technology, 33(8): 566-574, Oct. 1991.

[Dun90] R. H. Dunn: *Software Quality - Concepts and Plans*, Prentice-Hall, 1990.

[Gam92] E. Gamma: *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*, Springer-Verlag, Berlin, 1992.

[GAO94] D. Garlan, R. Allen, J. Ockerbloom: *Exploiting style in architectural design environments*, In Proceedings of SIGSOFT '94, ACM Press, Dec. 1994.

[Gar95] D. Garlan: *ICSE-17 First International Workshop on Architectures for Software Systems, Workshop Summary*, ACM Software Engineering Notes, 20(3): 84-89, July 1995.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley professional computing series, Reading, Massachusetts u.a., 1995.

[GS93] D. Garlan, M. Shaw: *An Introduction to Software Architecture*, in V. Ambriola, G. Tortora (eds.): Advances in Software Enginnering and Knowledge Engineering, Vol. I, World Scientific Publishing Company, 1993.

[GS94] D. Garlan, M. Shaw: *Characteristics of Higher-level Languages for Software Architecture*, Technical Report CMU-CS-94-210, School of Computer Science and Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, Dec. 1994.

[GTP95] D. Garlan, W. Tichy, F. Paulisch: *Dagstuhl Software Architecture Workshop Summary*, ACM Software Engineering Notes, 20(3): 63-83, July 1995.

[Hay94] R. Hayes-Roth: *Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*, Technical Report, Teknowledge Federal Systems, Oct. 1994.

[Hid90] *Proceedings of the Workshop on Domain-Specific Software Architectures*, Technical Report CMU/SEI-88-TR-30, Software Engineering Institute, Hidden Valley, PA, July 9-12, 1990.

[IBM91] IBM (Eds.): *Insurance Application Architecture - Data Model Reference Manual*, IBM Corp.; No. GE19-5644-0, 1991.

[IBM92] IBM (Eds.): *Financial Application Architecture - Introduction*, IBM Corp.; No. GC31-3932- 0, 1992.

[IBM95] IBM (Eds.): *Open Blueprint - Introduction to the Open Blueprint: A Guide to Distributed Computing*, IBM Corp.; No. G326-0395, 1995.

[ISO84] International Standards Organisation: *International Standard ISO7498: Information Processing Systems* - Open Systems Interconnection - Basic Reference Model, 1984.

[IW95] P. Inverardi, A.L. Wolf: *Formal specification and analysis of software architectures using the chemical abstract machine model*, IEEE Transactions on Software Engineering, 21(4), April 1995.

[JF88] R. E. Johnson, B. Foote: *Designing Reusable Classes*, Journal of Object-Oriented Programming, 1(2): 20-35, June/July 1988.

[KB94] R. Kazman, L. Bass: *Toward Deriving Software Architectures From Quality Attributes*, Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-94-TR-10, 1994.

[Kic92] G. Kiczales: *Towards a New Model of Abstraction in the Engineering of Software*, in IMSA'92 (Workshop on Reflection and Meta-level Architectures), 1992.

[KP88] G. Krasner, S. Pope: *A Cookbook for Using Model-View-Controller User Interface Paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, 1(3): 26-49, August/September 1988.

[LKA+95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, W. Mann: *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, 21(4): 336-355, April 1995.

[Mey88] Bertrand Meyer: *Object-oriented Software Construction*, Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[NR69] P. Naur, B. Randell (Eds.): *Software Engineering. Report on a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7th to 11th Oct. 1968. Bruxelles: NATO Scientific Affairs Division, 1969.

[ÖS94] H. Österle, J. Sanche: *Systementwicklung mit Applikationsplattformen - Erfahrungen bei der Lufthansa und der Schweizerischen Kreditanstalt*, Wirtschaftsinformatik, Vol. 36, pp. 145-154, 1994.

[Pfa83] G. E. Pfaff (Ed.): *User Interface Management Systems*, Proceedings, Workshop on User Interface Management Systems, Seeheim, Nov. 1983; Springer Verlag, 1983.

[PN86] R. Prieto-Diaz, J. Neighbors: *Module Interconnection Languages*, The Journal of Systems and Software, 1986.

[Pre95] W. Pree: *Design Patterns for object-oriented Software Development*, ACM Press books, Addison-Wesley Publishing Company, Wokingham, England u.a., 1995.

[PW92] D.E. Perry, A.L. Wolf: *Foundations for the study of software architecture*, ACM SIGSOFT, Software Engineering Notes, 17(4), pp.40-52, Oct. 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson: *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Sch96] H. A. Schmid: *Design Patterns for Constructing the Hot Spots of a Manufacturing Framework*, Journal of Object-Oriented Programming, 9(3): 25-37, June 1996.

[SDK+95] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik: *Abstractions for Software Architectures and Tools to Support Them*, IEEE Transactions on Software Engineering, 21(4): 356-372, April 1995.

[SNH95] D. Soni, R. Nord, C. Hofmeister: *Software Architectures in Industrial Applications*, in Proceedings of the 17th International Conference on Software Engineering, pp. 196-207, 1995.

[Tal96] On Taligent's Internet home page *http://www.taligent.com* there is a variety of information and documents about Taligent and their framework approach.

[Tra95] W. Tracz: *DSSA (Domain-Specific Software Architecture) Pedagogical Example*, ACM Software Engineering Notes, 20(3): 49-62, July 1995.

[VAA95] GDV: *VAA - Die Anwendungsarchitektur der Versicherungswirtschaft*, Version 1.3, Gesamtverband der Versicherungswirtschaft, Bonn, Germany, 1995.

[Wad94] D. Wade: *Options for persistent Objects*, Proceedings ObjectWorld 94, London, 1994.

[WG88] E. F. Wheeler, A. Ganeck: *Introduction to Systems Application Architecture*, IBM Systems Journal, 27(3): 250 - 263, 1988.

[Zac87] J. A. Zachman: *A Framework for Information Systems Architecture*, IBM Systems Journal, 26(3): 276-292, 1987.

[ZS92] J. A. Zachman, J. F. Sowa: *Extending and Formalizing the Framework for Information Systems Architecture*, IBM Systems Journal, 31(3): 590-615, 1992.