T U M

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

Software Architecture in Depth

Lars Heinemann, Christian Neumann, Birgit Penzenstadler,
Wassiou Sitou

TUM-I1007
April 10

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-04-I1007-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Preface

The quality of software architecture is one of the crucial success factors for the development of large and/or complex systems. Therefore, a good software architect plays a key role in every demanding project: She or he has the overview of the overall system and sets the framework for the implementation. In order to be successful in this task, software architects need well-founded and encompassing knowledge about design, which exceeds pure programming and specific specialization areas. The master seminar "Software Architecture in Depth" builds on the lecture "Software Architecture" held at Technische Universität München.

The master seminar "Software Architecture in Depth" aims at further deepening of important areas of the domain of software architecture. On one hand, we elaborate areas that were not explicitly considered in the lecture, as for example architecture evaluation and architecture refactoring. On the other hand, we acquire topics of current interest, as for example the REST architectural style.

Lars Heinemann,
Christian M. Neumann,
Birgit Penzenstadler,
Dr. Wassiou Sitou

Contents

1 State of the art of Architectural Approaches

Tankred Hase
hase@in.tum.de

Abstract. This chapter lays the foundation for the master seminar “Software Architecture In Depth”. First, broadly accepted definitions of the software architecture domain will be given, which will introduce a common vocabulary and should ease the understanding of the following chapters. Second, a list of best practices will be discussed, portraying state of the art approaches to the discipline of software architecture.

1.1 Introduction

Software architecture has received growing attention in the past decades and has established itself as an accepted engineering discipline in both research and practice. The term "software engineering" was first coined in 1968 at the NATO Science Committee, where solutions to the so-called "software-crisis" were discussed. This is where Edsger Dijkstra and David Parnas introduced the concept of software architecture. [1]

In the following years, the Waterfall Model was introduced, a development process in which an explicit design phase was placed after an initial requirements analysis phase. But it wasn’t until the V-Model, that the term "software architecture" was used as the product of subsystem decomposition. This phase also followed requirements analysis, but was to be executed before a detailed design phase. This clearly stated, that the architecture consisted of the decomposition into empty shells and that the innards or details of these shells were subject to a later phase. [2]

Today, the architecture itself is represented in different views, namely the functional, logical, and technical architecture. Among other aspects, this shows how far the computer science community has come in the past 40 years. [2]

In order to summarize this development, broadly recognized definitions of the software architecture domain will be cited in the following section. This eases understanding of the following chapters and above all lays a solid foundation for communication about architecture.

1.2 Definitions

Finding broadly accepted definitions for the software engineering domain is quite difficult, as it is a fairly young discipline and the rate of development is very rapid when compared to other sciences. The answer to the question, what a software architecture is, can at best be answered in a multi-tiered fashion.

At the highest level of abstraction, the system can be described using architectural patterns. These patterns define the overall shape and structure of software applications. The next level of abstraction can be described by an instance of one or more architectural patterns, namely by the software architecture itself. The following level, which handles the reuse of components and their interconnections, can be described using design patterns. [3]

In the following, some of the most widely accepted definitions are cited.

- **Component:** "A software Component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [2, p. 1.20]
- **Software Architecture (SA):** "The structure of the components of a program/system their interrelationship, and principles and guidelines governing their design and evolution over time." [2, p. 1.21]
- **Design Pattern:** "A design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved." [4]
- **Architectural Pattern:** "Architectural patterns are software patterns that describe solutions known to work efficiently, to architectural problems in software engineering. It gives description of the elements and relation type together with a set of constraints on how they may be used. An architectural pattern expresses a fundamental structural organization schema for a software system, which consists of subsystems, their responsibilities and interrelations. In comparison to design patterns, architectural patterns are larger in scale." [5]

Now that the foundation has been set by naming some important definitions, the challenges of architecture design will be discussed in the following section.

1.3 Challenges of Architectural Design

Before one can begin to design a system's architecture, one must first understand the fundamentals of the software development process. Typically one of the first action items in any process model consists of requirements elicitation. This is important, because both the functional and non-functional requirements have a strong impact on the resulting architecture. Figure 1.1 depicts this relationship.

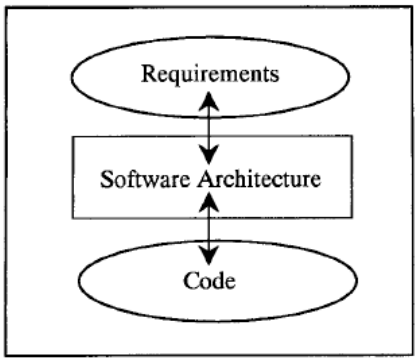


Fig. 1.1: Relationship between the requirements and the architecture [1]

This is a very simplified perspective, as there are usually many other factors and constraints that may influence architecture design. But typically, the problem can be boiled down to the following:

1. There are requirements, that the system is expected to satisfy.
2. Architecture design should consider these requirements.
3. The resulting code should reflect the constructed architecture.

The problem is, that a architecture can hardly meet every requirement perfectly. In requirements-elicitation there are typically two kinds of requirements:

- **Functional requirements**, which "define the functions of a software system or its component. A function is described as a set of inputs, the behavior, and outputs." [6]
- **Non-functional requirement** (or quality attributes), which "specify criteria that can be used to judge the operation of a system, rather than specific behaviors." [7] Examples for non-functional requirements are dependability or maintainability.

The following sections will discuss how these requirements can best be reflected by the architecture.

1.3.1 Mapping functional Requirements to the SA

Defining the functional requirements of a system is typically straightforward. For instance, when designing an electric window for a car, a functional requirement might be that the window should move up when pressing "up" and should move down when pressing "down". When it comes to designing the architecture for this system, ideally spoken, a component is defined, whose sole purpose is to implement this functionality. So one way to map functional requirements to the architecture is done simply by defining components through separation of concerns, and then connecting these components, so that minimal dependencies exist. [2]

There are many different strategies to this approach, but sticking to the rule of thumb called "high coherence, low coupling" will typically be a good idea. This pretty much means, that when trying to define subsystems, one should combine functions that have many dependencies into a component and try to decouple these from functions, that have little or no dependencies. [2]

1.3.2 Considering non-functional Requirements

Trying to respect all given non-functional requirements is often not as simple as dealing with functional requirements. For embedded systems for instance, resource constraints will typically be an important non-functional requirement, since hardware costs play an important role. This might be achieved by implementing the software modules in assembler-code in order to save memory. It may have great results when looking at the performance requirements, but be fatal for portability and maintainability. This is why two architectures with the same functional requirements but differently prioritized non-functional requirements may be totally different. [2]

1.4 Best Practices

It is certainly not trivial to design an appropriate architecture for a certain cause. It can be helpful to learn from the experience of others, because common problems often arise during development. In the following, a list of best practices will be given. These are the results of the sum of experience gained by the software development community over the years. They can be divided into two main types of best practices; process recommendations and product recommendations.

1.4.1 Process Recommendations

- "The architecture should be the product of a single architect or a small group of architects with an identified leader.
- The architect (or architecture team) should have both a list of the functional requirements and a prioritized list of the non-functional requirements that the architecture is expected to satisfy.
- The architecture should be well documented, with at least one static view and one dynamic view, using an agreed-on notation that all stakeholders can understand with a minimum of effort.
- The architecture should be circulated to the system's stakeholders, who should be actively involved in its review.
- The architecture should be analyzed for applicable quantitate measures (such as maximum throughput) and formally evaluated for quality attributes before it is too late to make changes to it.
- The architecture should allow incremental implementation through the creation of a "skeletal" system in which the communication pathways are established but which at first has little functionality. This eases the integration and testing of additional functionality.
- The architecture should result in a specific (and small) set of resources contention areas, the resolution of which is clearly specified, circulated and maintained. For example, if network utilization is an area of concern, the architect should produce (and enforce) for each development team guidelines that will result in a minimum of network traffic." [8, p. 15]

1.4.2 Product Recommendations

- "The architect should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns. The information-hiding modules should include those that encapsulate idiosyncrasies of the computing infrastructure, thus insulating the bulk of the software from change should the infrastructure change.

- Each module should have a well-defined interface that encapsulates or "hides" changeable aspects (such as implementation strategies and data structure choices) from other software that uses its facilities. These interfaces should allow their respective development team to work largely independently of each other.
- Quality attributes should be achieved using well-known architectural tactics specific to each attribute.
- The architecture should never depend on a particular version of a commercial product or tool. If it depends upon a particular commercial product, it should be structured such that changing to a different product is straightforward and inexpensive.
- Modules that produce data should be separate from modules that consume data. This tends to increase modifiability because changes are often confined to either the production or the consumption side of data. If new data is added, both sides will change, but the separation allow for a staged (incremental) upgrade.
- For parallel-processing systems, the architecture should feature well-defined processes or tasks that do not necessarily mirror the module decomposition structure. That is, processes may thread through more than one module; a module may include procedures that are invoked as part of more than one process.
- Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of simple interaction patterns. That is, the system should do the same things in the same way throughout. This will aid in understandability, reduce development time, increase reliability, and enhance modifiability. It will also show conceptual integrity in the architecture, which, while not measurable, leads to smooth development." [8, p. 16]

1.5 Summary

In this chapter, the fundamentals of software-architecture design have been discussed. Some well accepted definitions have been laid out, which should ease the understanding of the following chapters. Also a list of generally accepted best practices have been named. This list is neither complete, nor is it suitable for every software project. Yet it may help the reader in his/her next software project.

The important thing to keep in mind is that the definitions and rules discussed here are not set in stone. Software architecture is a very young engineering discipline, and one should always keep in mind, that the rules of today might change in the future, as more and more experience is gained throughout the software development community.

2 The Role of Architecture throughout the Development Process

Young-chul Jung
jungy@in.tum.de

Abstract. Today in many fields of software development the interest in software architecture is increasing, and the good software architecture is an essential element of successful software development. Thus, what the question to be asked is what is software architecture and what role does it play? This report introduces at first the definitions of software architecture and describes its role. Afterwards, the role of software architecture in traditional and modern process models are presented and various software development processes will be compared. Furthermore, from the process model perspective where the software architecture identifies itself within the process will be analyzed.

2.1 Software architecture

In this chapter the definitions of software architecture will be answered for the question: "What is software architecture?" and the general roles of software architecture will be declared in order to explain its imporatance.

2.1.1 What is software architecture?

Currently various definitions of software architecture exist. Hereupon, some well-known definitions of *software architecture* will be introduced.

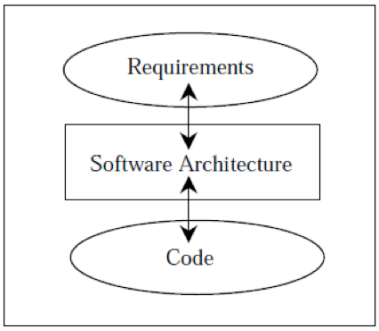
- IEEE-Std-1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems[9]: “Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution”
- Bass, Clements, and Kazman. Software Architecture in Practice 2nd Ed., Addison-Wesley 2003[10]: “The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”
- Garlan and Perry, guest editorial to the IEEE Transactions on Software Engineering, April 1995[11]: “The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.”

Additional definitions exist, however their discrepancies and differences are quite small, thus retaining a common meaning among all contexts. There are two important aspects: High-level structure and Project-wide important design guidelines.

2.1.2 The general roles of software architecture

While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top level design decisions, including things such as how the system is composed of interacting parts, where the main pathways of interaction are, and what the key properties of the parts are. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal. Software architecture typically plays a key role as a bridge between requirements and implementation[1]. By providing an abstract description of a system,

Fig. 2.1: software architecture as a bridge between requirements and implementation



the architecture exposes certain properties, while hiding others. Ideally this representation provides an intellectually tractable guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition. For example, an architecture for a signal processing application might be constructed as a dataflow network in which the nodes read input streams of data, transform that data, and write to output streams. Designers might use this decomposition, together with estimated values for input data flows, computation costs, and buffering capacities, to reason about possible bottlenecks, resource requirements, and schedulability of the computations.
Garlan[1] used the six aspects of software development to explain important roles of software architecture.
1. **Understanding**: Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's high-level design can be easily understood. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices.
2. **Reuse**: Architectural descriptions support reuse at multiple levels. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components and also frameworks into which components can be integrated. Existing work on domain-specific software architectures, reference frameworks, and architectural design

patterns has already begun to provide evidence for this.
3. **Construction**: An architectural description provides a partial blueprint for development by indicating the major components and dependencies between them. For example, a layered view of an architecture typically documents abstraction boundaries between parts of a system's implementation, clearly identifying the major internal system interfaces, and constraining what parts of a system may rely on services provided by other parts.
4. **Evolution**: Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications. Moreover, architectural descriptions separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components, by clearly distinguishing between components and mechanisms that allow them to interact. This separation permits one to more easily change connection mechanisms to handle evolving concerns about performance interoperability, prototyping, and reuse.
5. **Analysis**: Architectural descriptions provide new opportunities for analysis, including system consistency checking conformance to constraints imposed by an architectural style, conformance to quality attributes, dependence analysis, and domain-specific analyses for architectures built in specific styles.
6. **Management**: Experience has shown that successful projects view achievement of a viable software architecture as a key milestone in an industrial software development process. Critical evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies, and potential risks.

2.2 A general Software development process

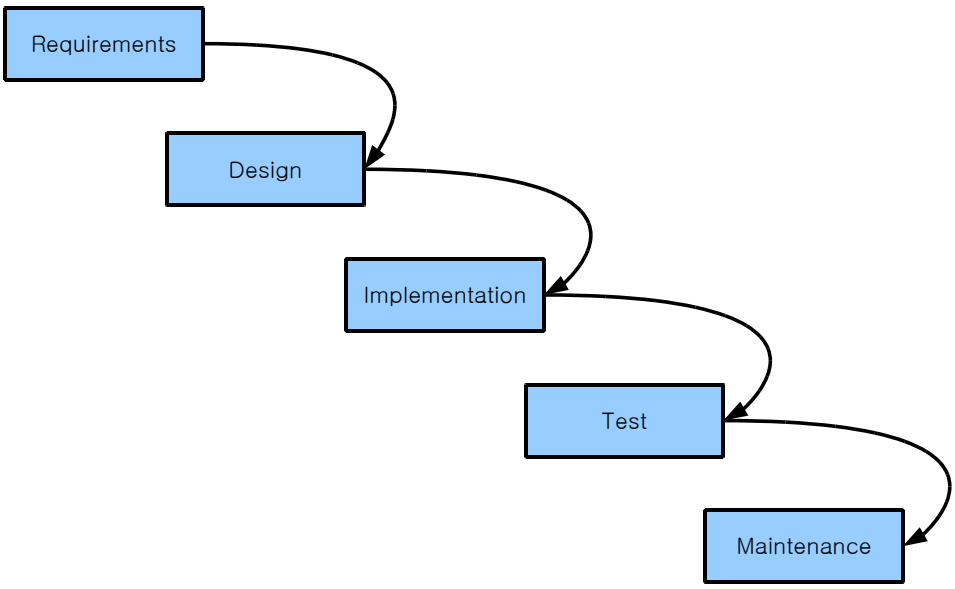
A software development process is a structure imposed on the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. We will now examine the traditional waterfall model as a general software development process. Its activities are also included in other software development processes.

2.2.1 Activities and Structure

The first published model of the software development process was derived from more general engineering processes. This is illustrated in figure 2.2. Because of the cascade from one phase to another, this model is known as the waterfall model.

The principal stages of the model map onto fundamental development activities[12]:
1. **Requirements analysis and definition**: the system's services, constraints and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. **System and software design**: The systems design process partitions the requirements to either hardware or software systems. It establishes an overall system architecture. Software

Fig. 2.2: Waterfall model as a general software development process



design involves identifying and describing the fundamental software system abstractions and their relationships.
3. **Implementation and unit testing**: During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
4. **Integration and system testing**: The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. **Operation and maintenance**: Normally(although not necessarily) this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle. Improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved. The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. During design, problems with requirements are identified: during coding design problems are found and so on. The software process is not a simple linear

model but involves a sequence of iterations of the development activities. Because of the costs of producing and approving documents, iterations are costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured system as design problems are circumvented by implementation tricks. During the final life-cycle phase (operation and maintenance), the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.
The advantages of the waterfall model are that documentation is produced at each phase and that it fits with other engineering process models. Its major problem is its inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.

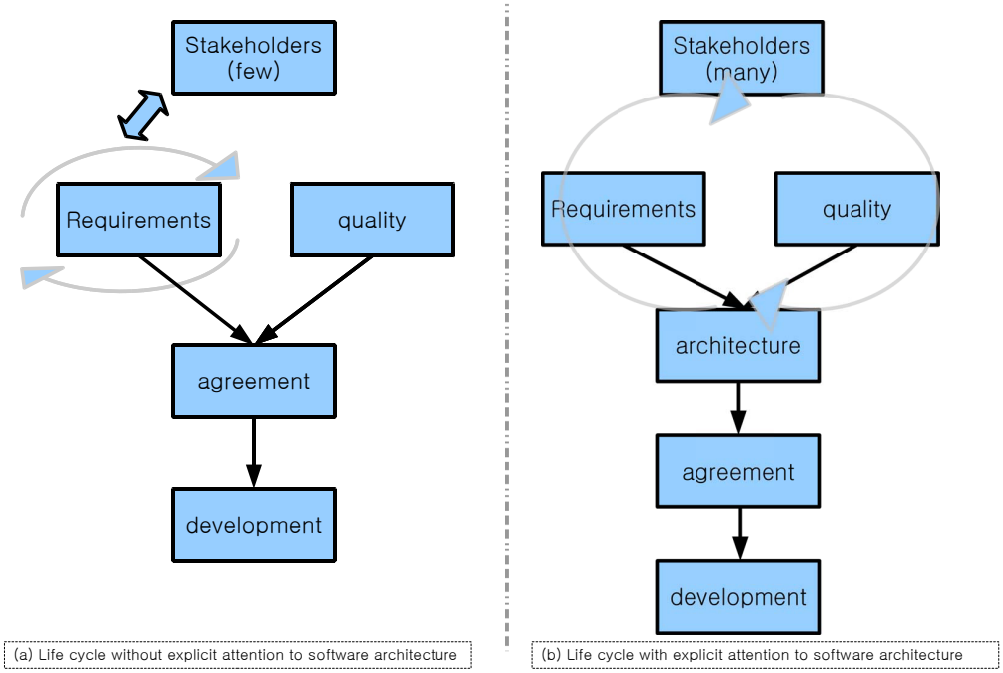
2.2.2 Software architecture in general software development process

The software architecture is defined and developed during the design phase according to its role. All next activities follow the architecture description which is created in the design phase. The design phase is simply split into two subphases[12]: architectural, global design and detailed design. the methods used in these two subphases might be different, but both essentially boil down to a decomposition process, taking a set of requirements as their starting point. both design phase are inward-looking: starting from a set of requirements, derive a system that meets those requirements.

- Architectural, global design: this activity involves designing the software architecture. This includes completing the interface descriptions and updating the software integration plan.
- Detailed design: the software architecture and interface description are further detatiled. The specification and details for each software module, component and database are made.

According to Vliet[12] a 'proper' software architecture phase however has an outward focus as well. It includes the negotiating and balancing of functional and quality(or non-functional) requirements with possible solutions. This means requirements engineering and software architecture are not consecutive phases that are more or less strictly separated, but instead they are heavily intertwined. An initial set of functional and quality requirements is the starting point for developing an initial architecture. This initial architecture results in number of issues that require further discussion with stakeholders. For instance, the envisaged solution may be too costly, integration with existing systems may be complex, maintenance may be an issue because of a lack of staff with certain expertise, or performance requirements cannot be met, these insights lead to further discussions with stakeholders, a revised set of requirements, and a revised architecture. This iterative process continues until an agreement is reached. Only then will detailed design and implementation proceed.
The difference between these two paradigms is illustrated in figure 2.3.

Fig. 2.3: software development process without and with explicit attention to software architecture

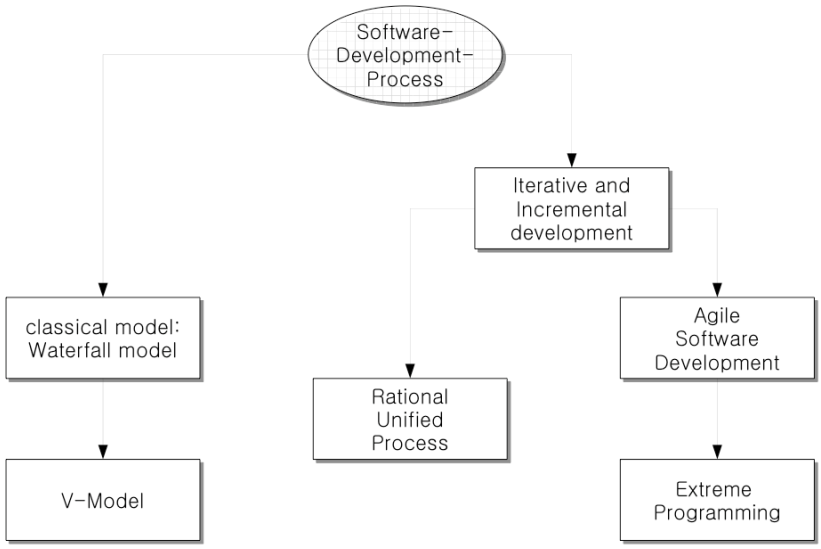


We thus see important differences between traditional process models that do not specific attention to software architecture and process models that do pay attention to software architecture:
-In traditional models, iteration only concerns functional requirements. Once the functional requirements are agreed upon, design starts. In process models that include a software architecture phase, iteration involves both functional and quality requirements. Only when the combined set of functional and quality requirements is agreed upon will development proceed.
-Traditional models involve negotiation with a few stakeholders only. Usually, only the client and end users are involved. Negotiations about architectural solutions may involve a much larger variety of stake holders and include, for instance, the future maintenance organization for the system to be developed and owners of other systems that this system has to interact with.
-In traditional models there is no balancing of functional and quality requirements. Once the functional requirements are agreed upon, development proceeds and it is assumed that quality requirements can be met. If it turns out that the quality requirements cannot be met, the project gets into trouble. Deadlines slip, functionality is skipped, more hardware is bought, etc. in process models that include a software architecture phase, there is a balancing of functional and quality requirements at an early stage.

2.3 Various software Development processes

Nowdays there are many different software development processes in existence. Of course each process is used in various ways and has different advantages and disadvantages. The V-model as an advanced waterfall model, Rational unified process and agile development as kinds of iterative and incremental development, will be introduced in this chapter and compared to each other where for them the role of software architecture will be found.

Fig. 2.4: Category of software development processes

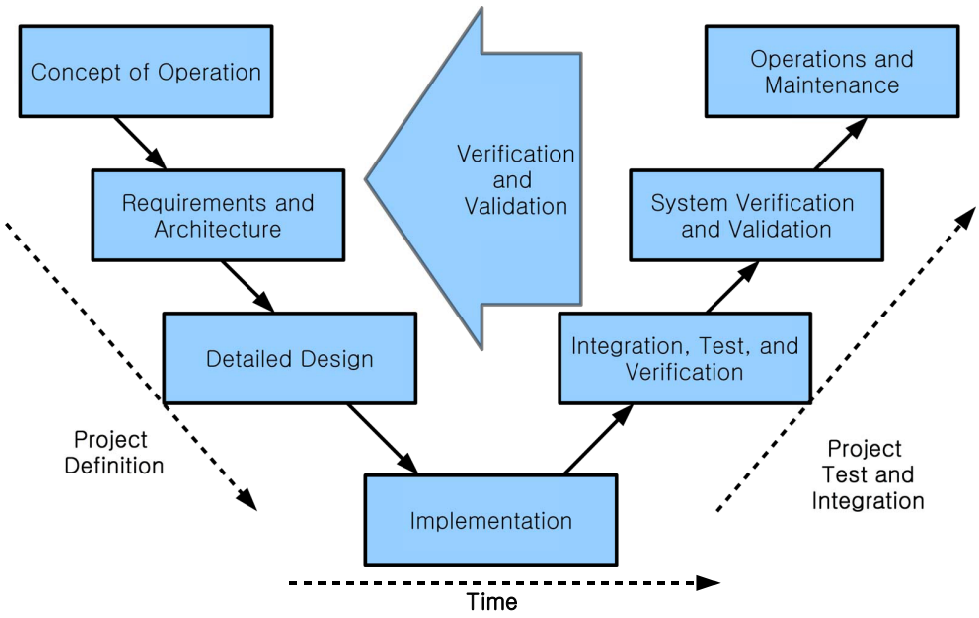


2.3.1 V-Model

The V-model summarizes the main steps of the waterfall model to be taken in conjunction with the corresponding deliverables within computerized system validation framework. The V-Model, also called the Vee-Model, is a product-development process originally developed in Germany for government defense projects. It has become a common standard in software development. The V-Model gets its name from the fact that the process is often mapped out as a flowchart that takes the form of the letter V.
The development process proceeds from the upper left point of the V toward the right, ending at the upper right point. In the left-hand, downward-sloping branch of the V, development personnel define business requirements, application design parameters and design processes. At the base point of the V, the code is written. In the right-hand, upward-sloping branch of the V, testing and debugging is done. The unit testing is carried out first, followed by bottom-up integration testing.

The extreme upper right point of the V represents product release and ongoing support[12]. The

Fig. 2.5: Activities and Structure of the V-Model



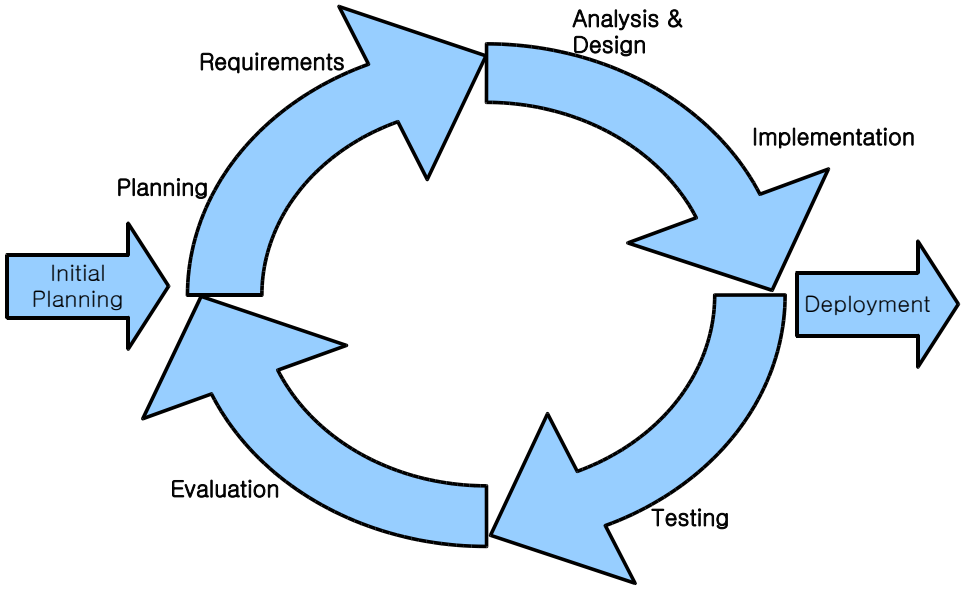
roles of software architecture of the V-model are definitely similar to the waterfall model, and it plays the role of standard criteria for verification and validation of the end/resulting system. Here verification and validation mean different[13]. It is sometimes said that validation can be expressed by the query "Are you building the right thing?" and verification by "Are you building the thing right?". "Building the right thing" refers back to the user's needs, while "building it right" checks that the specifications be correctly implemented by the system. In some contexts, it is required to have written requirements and software architecture for both as well as formal procedures or protocols for determining compliance.

2.3.2 Iterative and incremental development

Iterative and Incremental development is a cyclic software development process developed in response to the weaknesses of the traditional waterfall model. The waterfall model isn't flexible for a changed development environment or changed requirements because it is slightly iterative or not at all. The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, where possible key steps in the process are to start

with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added. It starts with an initial

Fig. 2.6: An iteration of iterative and incremental development



planning and ends with deployment with the cyclic interaction in between[14]. By iterative and incremental development the software architecture can be also flexible changed during iteration. All of the system changes after iteration must be reflected within the software architecture, so the system can remain tractable and you can always have the whole view of the system and access to it as well. The iterative and incremental development is an essential part of the Rational Unified Process, Extreme Programming and generally the agile software development frameworks.

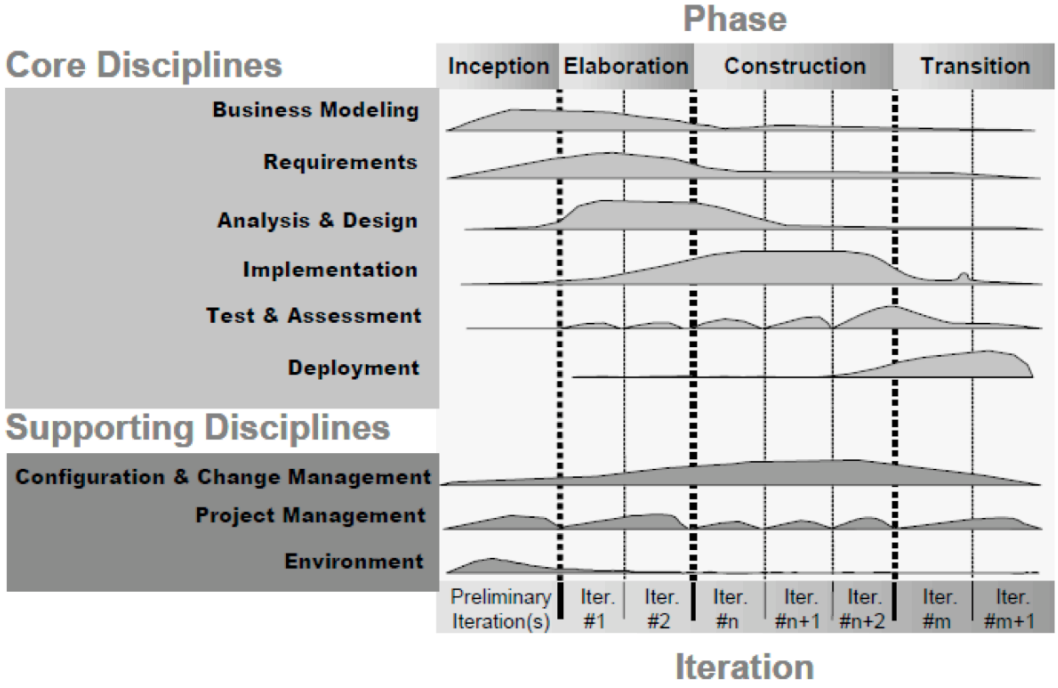
2.3.3 Rational Unified Process

The UP is often known as the Rational Unified Process (RUP) because Rational Software Corporation was the most successful vendor of UP. (The corporation was acquired by International Business Machines [IBM] in February 2003).

As shown in Figure 2.7, the RUP defines four phases[15].

- **The Inception Phase**: addresses the project's scope and objectives.

Fig. 2.7: Activities and Structure of Rational Unified Process, from IBM



- **The Elaboration Phase**: addresses major risks, builds an architecture, and evolves project plans.
- **The Construction Phase**: addresses detailed design, implementation, and testing.
- **The Transition Phase**: addresses fine-tuning functionality, performance, and overall quality.

The phases are made up of iterations of core disciplines associated with the key technical activities of software development: requirements, analysis and design, implementation, testing and assessment, deployment, and some other disciplines that are supporting in nature (configuration management, change management, project management, and the provision of an appropriate environment), and the software architecture is designed and analyzed in the Elaboration Phase, so as to be ready for the Construction Phase.
Jacobson[16] refers to the Unified Development Process as "iterative," "use-case-driven," and "architecture-centric." We briefly provide examples of each of these characteristics. The iterative nature of the RUP is illustrated in figure 2.7. The software is "grown" in a set of small iterative activities, not merely "developed" in one shot. The weight of the various disciplines is different in each phase and each iteration. This iterative process permits different things to be implemented in each pass through the disciplines. For instance, about 80 percent of the use cases are developed by the end of the Elaboration Phase after several iterations, leaving some requirements (captured eventually as added use cases) for later phases. Thus, change, especially

additional requirements, is recognized as a needed and natural part of software development.
The RUP is a use-case-driven process. Use cases are used to express the functional requirements of the system in a way that is understandable to the stakeholders.
A final important point to be made about the RUP is that it is architecture-centric. we use the list of characteristics of architectural centricity developed by Bass, Clements, and Kazman and reproduced in figure 2.8[10].

Fig. 2.8: architecture-centric development

Architecture-centric development involves iteratively

- creating the business case for the system
- understanding the requirements
- creating or selecting the software architecture
- documenting and communicating the software architecture
- analyzing or evaluating the software architecture
- implementing the system based on the software architecture
- ensuring that the implementation conforms to the software architecture

In RUP the 4+1 View framework is used to construct of software architecture[15], and the views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are logical, development, process and physical view. In addition selected use cases or scenarios are utilized to illustrate the architecture serving as the 'plus one' view.

- Logical view: The logical view is concerned with the functionality that the system provides to end-users.
- Development view: The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view.
- Process view: The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc.
- Physical view: The physical view depicts the system from a system engineer's point-of-view. It is concerned with the topology of software components on the physical layer, as well as communication between these components. This view is also known as the deployment view.

- Scenarios: The description of an architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype.

As we have seen in this chapter, software architecture plays a central role in RUP. With software architecture the resulting system can remain tractable and consistent during many iterations. Thus RUP is also suitable for big projects.

2.3.4 Agile development

By the end of the nineties, several methodologies began to attract public attention in general. Each one of these methodologies was a combination of old ideas, new ideas and variations of old ideas. But all of them had in common[12]: they emphasized the collaboration between programmer teams and business experts; they promoted direct face-to-face communication (as a more efficient way of communicating than through written documentation); frequent releases of new business functionality; self organized development teams; ways to structure source codes and development teams so that the most important requirements did not fall into crisis. By the beginning of 2001, the pioneers and creators of these methodologies gathered everything their methodologies had in common in a unique figure. By using the term "Agile" as a definition that represented them all, they created what they became to call the Manifesto for Agile Software Development. Some of the most important principles defined in this manifesto are[1]:

- Personal interaction of individuals over processes and tools.
- Working with the code over written documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The term "agile" does not describe a particular methodology, instead, it represents the existence of a family of methodologies that share the foundations and guidelines, which include the ones listed before. As examples of agile methodologies, we can mention Scrum, Feature Driven Development, DSDM, Crystal, XP, among many others.
From the perspective of change, the role of architecture in agile software development becomes quite clear: a good architecture is responsive and will support agility; a poor architecture will resist it and reduce it. With a clearer understanding of what we mean by architecture(as well as what we want from it), it is easy to see how very wrong claims along the lines of "We're agile; we don't need architecture!" are, as well as how harmful they can be in practice. What is not needed are lots of unreadable documents, long phases in a lifecycle dedicated to building infrastructure to the exclusion of all other development, and elite roles divorced from developers and their work. But although these associations exist in the minds and practices of many, none captures

[1] http://www.agilemanifesto.org/

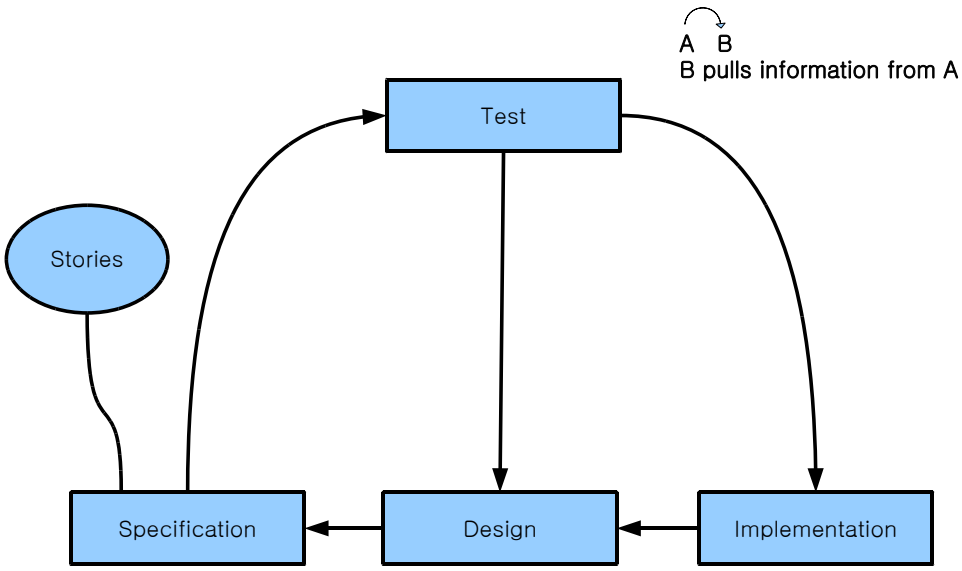
what architecture is about. Architecture involves the critical decisions for a system and a shared understanding of internal structure and development style. Architecture, in essence, captures the day-to-day (and today-tomorrow) communication between members of a development team. It is a medium for collaboration and work definition. Architecture constrains or enables different development approaches. Thus, architecture is more than just some optional consideration for agile development; it is essential.

2.3.5 eXtreme Programming

The XP(abbr. eXtreme Programming) method has quite a following. We chose it as an example of the agile development because it is one of the most mature (dating to the mid-1990s) and best known of the agile development process. It got the name "extreme" from its tendency to "turn up the volume" of its practices. The XP method is based on four values[17]:
1. **Communication** emphasizes person-to-person talking, rather than documents that explain the software. Also, several of the XP practices call for an on-site customer, so a lot of time is spent communicating to that customer.
2. **Simplicity** is an XP value that calls for the solution of the customer's problem to be unpretentious. Both developers and customers easily understand the software solution.
3. **Feedback** means that everything done is evaluated with respect to how well it works. How well it works is indicated through the feedback gained from exercising every working part of the solution.
4. **Courage** means that developers are prepared to make important decisions that support XP practices while building and releasing something of value to the customer each iteration.

XP describes four basic activities that are performed within the software development process[17].
1. **Implementation**: The advocates of XP argue that the only truly important product of the system development process is code - software instructions a computer can interpret. Without code, there is no work product. Implementation can also be used to figure out the most suitable solution. For instance, XP would advocate that faced with several alternatives for a programming problem, one should simply code all solutions and determine with automated tests which solution is most suitable. Implementation can also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem and finding it hard to explain the solution to fellow programmers might code it and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also coding their thoughts.
2. **Test**: One can not be certain that a function works unless one tests it. Bugs and design errors are pervasive problems in software development. Extreme Programming's approach is that if a little testing can eliminate a few flaws, a lot of testing can eliminate many more flaws.
There are 2 types of test.
Unit tests determine whether a given feature works as intended. A programmer writes as many automated tests as they can think of that might "break" the code; if all tests run successfully, then the coding is complete. Every piece of code that is written is tested before moving on to

Fig. 2.9: Activities and Structure of the eXtreme Prgoramming



the next feature.
Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements.
3. **Specification**: Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Understanding of his or her problem.
4. **Design**: From the point of view of simplicity, of course one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design structure that organizes the logic in the system. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system.

XP does offer few practices to help communicate and construct an architecture though[18]. Architecture shows up mainly in spikes(like as prototypes), the metaphor(like as a story

of everyone, customers, programmers and managers about how the system works) and the first iteration. Spikes are used to test out a preliminary architecture. Instead of validating an architecture for the right non-functional patterns, quality attributes are quickly tested out in a quick throw-away solution (e.g. a simulation). In this way, spikes serve as a first deployment. The metaphor is an XP practice that is supposed to provide a common base of understanding about the key concepts in the system and their interactions. In this way it acts as a simplified architecture. However, metaphor is one of the least applied XP practices because it is not easy to come up with a suitable metaphor. First Iteration is the implementation of a first set of user stories, Kent Beck explains, that are chosen to force create the whole system. It is supposed to be a skinny, installable and configured version of the software. First iteration is also expected to deliver the initial architecture of the system. Having an initial architecture is not sufficient, especially using a process that embraces changes so much as XP does. In XP, the small releases practice should detect and fix architectural errors quickly, because of early feedback in real use. Also, XP encourages larger refactorings[17]: changes in the software that do not affect functionality. This follows from the XP observation that you cannot do everything right from the beginning and the principle that coders should be designers as well (and vice versa). The latter principle ensures that programmers won't try to hack their way around the design and that designers are always in touch with the code. Refactoring avoids introducing new bugs by relying on continuous and automated testing and integration.

What about quality attributes in XP? XP believes it covers these in its user stories. These stories can be compared to use cases. XP's frequent releases ensure that the developers and customers can validate these attributes and solve problems early in the development process. It is clear that the XP process has certain benefits for small and medium sized software development projects. On the other hand, no large system can be constructed without an architecture that has been made explicit. Quality attributes are useful for projects of all sizes. The weakness of XP is perhaps that it does not forbid architecture, but that it does not encourage architecture. Unfortunately, the advantages of XP are difficult to transfer to architecture. XP relies on the interplay of an number of practices and principles that are easy to implement for code based development and difficult to apply on anything else. Shifting the balance of design and coding can easily disturb the efficiency of XP.

2.4 Summary

In some software development processes (e.g. RUP) the software architecture definitely plays an important role. On the other hand in some other processes (e.g. XP) it is neglected. However nobody will disagree with the fact that software architecture is an essential element of the software development process which supplies many advantages that have been mentioned within this report. Finally referring to the table 2.1 one can identify the differences between the V-model, RUP and XP. Although the software architecture in XP is almost neglected, better performance can be shown in smaller projects as opposed to RUP. Due to this characteristic, XP is not suitable for big projects in which many changes and numerous stakeholders may exist. Without software architecture, the resulting system cannot easily remain tractable after many changes have been

implemented by many people. Thus RUP is suitable for big projects. On the other hand it does not fit for smaller projects because it has many fundamental disciplines. Executions of all fundamental disciplines in all phases are very costly for small projects. If non-functional requirements are more involved from customers at the specification phase or a certain defined communication medium is used between programmers and customers e.g. strength of software architecture during an XP development process, then much time could be saved and customer satisfaction will improve.

Tab. 2.1: Overview of differences between the V-model, RUP and XP

	V-Model	**RUP**	**XP**
Scale of Project	small to big	small to big	small to medium
Duration of iteration	n/a	2 weeks to 6 months	about 2 weeks
Deal of SW architecture	standard	architecture-centric model, 4+1 view framework for architecture construct	neglected or burdensome
Position of software architecture	in Design phase and system test phase	from Inception to end of Elaboration	in Spikes, Metaphor and first iteration(not explicit as software architecture)
Role of SW architecture	main criteria for verification and validation of resulting system	make large, complex system tractable for change, supplying of view and access to the resulting system	supplying agility and responsiveness

3 Decomposition Criteria for Architecture

Robert Franz
franz@in.tum.de

Abstract. Many criteria are inevitable in developing a new software architecture. Especially decomposition criteria are vital to each software architecture. They are so important, because decomposing a software architecture thus means reducing complexity. So it is possible to create a software architecture that fulfills as many requirements as possible. A software architecture is influenced by a huge number of drivers. Almost each of them influences the development process when decomposing a software architecture. Decomposition Criteria is powered by four different parts: *Organisational Factors*, *Technical Factors*, *Non-functional Requirements* and *Functional Requirements*. Each of them contributes information that affects a software architecture. This information can be gathered from various sources. Furthermore, these criteria influence each other. This can be in a positive, as well as a negative way.

3.1 Introduction

Quality is one of the most important concerns when creating a new software architecture. Thus ensuring, that the quality of a software architecture fits underlying needs is very important. There are many factors to pay attention to, when creating a new software architecture. One relevant factor is called decomposition criteria. In order to reduce the complexity of a piece of a software architecture it is essential to split up this software architecture into modules. The manner in which a software architecture is split up, is called *Decomposition Criteria*. These decomposition criteria are crucial for operating the developed software architecture later.

This chapter handles decomposition criteria as well as their attributes. Furthermore there are examples, how they occur and how they interact with its environment. Moreover there is a description of all the sources, that provide information about decomposition criteria. In addition, there is a description of all the reciprocations of decomposition criteria and their effect on the development process. Finally, there is a section that handles choosing and prioritising decomposition criteria.

3.2 Decomposition Criteria

This section covers all the different decomposition criteria. Furthermore this section describes the characteristics a decomposition criteria has. In addition different architectural structures are handled. In detail there are organisational factors, technical factors, non-functional requirements and functional requirements [19], [10], [20], [21].

3.2.1 Architectural Structure

Before handling *Decomposition Criteria* it is necessary to determine the structure of a software architecture. When looking at an architectural structure, this can be divided into three groups. *Module*, *Component-and-Connector* and *Allocation* [10].

Module

A module represents a unit, that is responsible for a functional area of the whole system. A module can be one of the following:

- Decomposition
- Layer
- Use
- Class

Decomposition e.g. declares a modules module, submodules etc ... A *use* represents the usage of a module. This can be in versatile way. A *layer* is often designed in the form of an abstraction. It makes a system easier to understand. A *class* pictures the relation of a number of modules, the way how they work together and what kind of entities are affected.

Component-and-Connector

On the other hand a *Component-and-Connector* can be split up into:

- Process
- Concurrency
- Shared Data
- Client-Server

A process shows how component-and-connector work together and concurrency shows what kind of things can be parallelised. Shared data represents the kind of information that is held in common. At last the client-server describes how two components of a system work together.

Allocation

Allocation includes the following:

- Deployment
- Implementation
- Work assignment

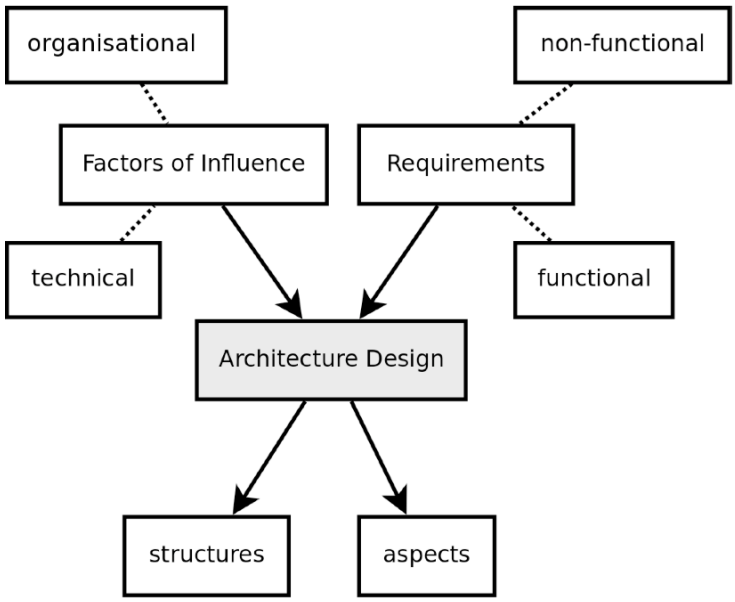
The deployment describes how a structure is assigned to hardware entities and the communication infrastructure. The implementation points out the way how software interacts with its environment, and the work assignment exposes the manner responsibilities within a software project are assigned.

Fitting together

In order to create a software architecture it is necessary to pay attention to the requirements. The requirements that are collected during the software development process lead to all the modules that have to be created in a software architecture. The decomposition criteria are directly affected by the requirements that are defined in the beginning of the software development process. [21]

Figure 3.1 shows how everything works together. The upper part of this diagram describes the factors that influence a software architecture. They can be distinguished between *Factors of Influence* and *Requirements*. The diagram shows how they are located and how they effect the design of architecture. This leads to aspects and structures of an architecture.

Fig. 3.1: Architecture Design



3.2.2 Organisational Factors

When developing a new software architecture, organisational issues impact the design of the software architecture. As different people take part during the development of a software architecture, each of them may have different aims. These different people are known as stakeholders. For example, the best technical solution does not have to be the best political solution as well. The reason for this effect is the fact that every stakeholder prefers different assets. So a perfect

technical solution does not fit the needs, because it is not possible to achieve this solution caused by political reasons. Thus it is very important, to find out the aim of each stakeholder. The following part describes a number of different factors concerning the organisational structure that influences a software project and therefore the software architecture. These factors are split up into *Organisation and Structure*, *Resources*, *Organisational Standards* and *Juristic Concerns*. [21]

Organisation and Structure

The following list describes the most important things that can be classified within *Organisation and Structure*:

Decision Maker Do the decision makers have experience in making similar projects? Are they willing to try something new? Do the decistion makers wish some special peculiarity for the software project?

Client May change responsibilities? May the person in charge change? On the client side it is possible, that the responsible person changes, so this may have influence on the software project.

Cooperations Does a cooperation demand some special requirements? Maybe the company has a contract or an agreement with another company, that influences the software architecture.

Product or own usage Is the new software system used for internal purpses only or shall it be sold as a product? It differs, whether the software is used within the company or used by costumers.

Internal or External Development Should the software product be developed within the company or by an external company? If the software is developed externally this might influence the software architecture.

The way how a company is organised, the manner how processes run and how a company solves its tasks is described as *Organisation and Structure*. Everything that is influenced by these aspects, affects the software architecture. This can result in a different way how software is developed, or a software architecture is designed. E.g. there is a cooperation with another company. In such a case, there might be an agreement, that some things must be done a predefined way. Or some regulations must be fit. In such a case, the development process of a software architecture must be modified. Another possibility is, that the software architecture must contain modules that can handle all the tasks that cope with these regulations, agreements or contracts. Either a software architecture must be extended in order to be compatible, or it must be adapted. For all the other points on the upper list, the same considerations are regarded.

Resources

Time How flexible is the time schedule? What do the stakeholders demand? If there is a tight time schedule, possibly a sophisticated architecture is not possible.

Effort Is there a limit? Paid per effort? Caused, by the maximum effort maybe a *Quick'n'Dirty* solution is the best, in a specific situation.

Release Plan Are there some special needs? Is it required to deliver a release of the software on a special date? Maybe a number of development stages require a suited strategy.

Team What kind of abilities do the members of the development team have? Are the members of the team availible continuously? abilities of the team members lead to a different software architecture.

Another important thing, that influences a software architecture, are all the available resources. As described above, limitations within a company affect a software architecture. Everyone of these limitations lead to modifications concerning the decomposition criteria that affects a software architecture. When there is a team, that has some specific abilities, this maybe has an affect on the software architecture directly. In case a team member is an expert in a very sophisticated issue, it is possible to create a software architecture that uses a sophisticated mechanism. Normally nobody would invest time in order to introduce this issue, but it is possible to use it, as a team member already has the knowlege necessary to deal with it.

Organisational Standards

Quality Does the company have to take notice of a quality standard? In case there is a quality standard, that requires a customised strategy when developing the software architecture or running the software.

Tools What kind of tools are availible? What kind of software development is possible? Which tools are allowed for use? Do the tools support processes and procedures that influence the software architecture?

SLAs Are there some special service level agreements that must be paid attention to? E.g. availability or performance. A system that has to be highly available requires another architecture opposed to a system that doesn't must fit this requirement.

Procedure Model Internal documentation, standards, implementation.

Furthermore fundamential things that exist within a company are also vitial concerning decomposition criteria. Normally in each company, there exist standards and processes, that must be observed. E. g. a company is certified in being able to comply with a quality management standard, this must be respected when developing a new software architecture. But also tools that are used within the company influence the possibilities when developing software. Maybe some processes are very efficient, in case there is a software solution, compared to a company that doesn't use such a solution. In case, there is a service level agreement that requires the company to provide a minimum level of service, this has an impact on a software architecture. In case a special mechanism has to be installed in order to guarantee this level. A much more precise description is delivered in section 3.2.3.

Juristic Concerns

Liability Is the company liable when offering a software system? Are human beings threatened? This can affect the requirements concerning reliability, safety, etc ...

Privacy Does the software system store confidential information? In case there is information that represent a company's business secrets, that endanger a company when this information is published.

Compulsory Verification Is it necessary to be able proof the flow of some processes? It is possible, that a law, or contract forces the company to proof the flow of a process or procedure.

The field of one of the most important factors is *juristic concerns*. Because this concerns may have a huge impact to a company, in case there is a problem, this really must be considered extremly carfully. Although this kind of concerns don't apply for every company, those companies that are affected by it, have to take notice of them. A company, that sells a software product and furthermore guarantees, that his piece of software works without any problems within defined parameters, is an example of such a company, that is a affected by juristic concerns. But the reasons why this company gives such a guarantee are different. It's the same for the kind of guarantees a company gives.

Imagine, there is a law, that requires a company to log all transactions a custumer performs. This can be required in order to be able to prove the amount of usage, the company provides to the custumer. The company is not allowed to bill a customer for services he/she has used without being able to prove exaclty when and how long the customer used these services. One famous example for such a regulation are telecommunication services. Connecting this issue with the point described above, *compulsory verification* fits this issue. But when taking a telecommunications service as an example for *juristic concerns*, in such a case *privacy* is also an important factor. Decomposing a new software architecture, is affected if verfiication is mandatory. A subsystem is required, that handles everything in order to store this information. When having a look at all the concerns described above, it might be necessary to cope with some special requirements regarding how this information must be stored. So the way how to store this information, as well as the place or the type of storage influences a software architecture, because these specifications must be fit.

Discussion

All the factors above describe things that influence a software architecture by organisational issues. This may be be caused by the structure of the company, custumers or the field of application. Furthermore available resources influence a software architecture. The same is with policies, strategies and procedures.

3.2.3 Technical Factors

As well as the organisational factors (See section 3.2.2) , the technical factors are also important. A technical factor may change a system architecture fundamentally. This is caused by technical

reasons where it is possible, that some special attributes of a software architecture change on the base level. Technical reasons may be a limited number of resources that can used at maximum. Another possibility is the fact, that special technical demands result in restrictions when developing a new software architecture. E.g. requirements based on technical issues, demand the software architecture to comply with some properties [21]. A number of technical factores are chosen in the section below. Of course, not every possible factor is written below, because this would exceed the range, that is necessary to handle decomposition criteria. These factors are described and explained. The following list describes this number of items, that describes technical factors:

Hardware

- Processor
- Main Memory
- Network
- Storage System

All the available hardware as well as features, this hardware provides. In case of a processor, the number of processors and its speed is an important factor. But each processor supports a number of instructions and instruction sets. This influences the development of new software. In case of main memory, its latency, the available amount of memory or the bandwith of the memory controller is crucial. When having a look on the network connection, the first point is the available bandwidth, but also the latency or the reliability of the connections is relevant. In case there is an existing storage system, the interfaces of this storage system, the type of this storage system and its charactaristics.

Software

- Operating System
- Existing Program Libraries
- Database System
- Middleware
- Other Server Software

Concerning the already used software components of an existing system, as listed above , their charactaristics are now described more precisely. Each operating sytem provides a number of features and charactaristics. Furthermore a operating system is able to provide a number of services and interfaces. The way, how these things can be used, as well as the the quality of service has an impact on the software architecture. Each operating system provides different services, as well as user interfaces to these serivces. In addition, program libraries are available.

These libraries, provide a number of functions and interfaces. It is possible to compare this with an operating system. If there is a program library, this library must be called a specific way and then delivers a result. The same is the database system. An instance of a such a database system is a relational database. It provides an interface between how to query data, and how to connect to this database. The interface between how this database is connected and its services, is crucial for a new software architecture. An example for this is an query using the Structured Query Language to this database system via Open Database Connectivity. Another software component is the middleware used in a existing software system. There is a number of different products used for middleware. Each middleware provides services that can be used by the new software architecture. It is important to know how to connect to this middleware, as well as what kind of service can be used. For example there is a software component that must be used in order to connect to the middlware. This software component may have a special way to cope with the middleware. Of course, there can be other server software also. As well as all the other software components described above, this influences the process when developing a new software architecture.

Data Structures

- Present Data Structures
- Changeability of current Data Structures

Compared to software, this factor is handled quite easy. Normally there are existing data structures. These data structures must be used in the new software architecture. So the new software must fit the existing data structures. There is a possible conflict, when it is required to modify an existing data structure. Maybe it is not possible to modify the data structure directly. Otherwise it might be inevitable to modify other software that accesses to this data structure. This possibly causes further problems.

Graphical User Interface

- Special Requirements
- Used technology
- Conventions concerning current User Interface

In case, there is an existing graphical user interface, this user interface might be extended. So conventions of this user interface should be respected by the new software. Another point is the technology, that is used in this present graphical user interface. Maybe some use cases might be not possible with the existing technology, or a component of this user interface must be modified in order to fit the requirements to the new software. Moreover it is possible, that there are special requirements to this graphical user interface. E.g. a 3D surface is required for some purposed. So decomposing a software architecture with such a requirement leads to another software architecture than without it.

Interface

- Existing Interfaces
- Conventions when calling Interfaces

In every software architecture, there exist interfaces. These existing interfaces have an effect on the new software archtecture. It must be adepted in order to operate with these interfaces. It is also necessary to make the software architecture work with the way how these interfaces can be accessed. This means the protocol, that is required in order to access the intefaces. Maybe there are conventions and rules in calling such an interface. This must be considered when decomposing a software architecture.

Discussion

The list above shows a number of factors, that are essential to every software project. These factors described above influence a software architecture a great deal. This is caused by the importance of these factors. Every system has some technical limitations or conventions that have to be known and to take notice of. These factors are so important, cause they describe the current IT system, for example. Existing technical resources and the way how these resources can be used in the new software architecture is very important. Maybe a new software must be used on an existing hardware system, therefore technical limitations as memory, bandwith, the operating system, etc ... lead to a software architecture that is characterised by these resources. The way, how these resources can be extended, the fact, what kind of new resources can be used and integrated into the consisting system limits the new software architecture.

When looking a the following situation: An existing hardware system is based on a virtual machine. This virtual machine, has an amount of main memory, a number of processors and some storage memory. Let's say, this resources are very limited, because the host system of the virtual machine doesn't support more resources, because it is at its maximum. In case there is no more free processor that can be used, the software must be so effiecient in using processor power so that this one processor is enough. Another possibility is limited main memory. So the program has to be as efficient in using the main memory, that the maximum size of available main memory it is no exceeded. Leaving this example system, imagine, that there is an existing data structure that is required to be used in order to store information in it. So when developing a new software architecture, it is necessary, to used this data structure e.g. on a database system. Another possibility is a file system that has XML files and requires the data to be stored in. So the new software architecture must use this existing structure in order to cope with the existing system.

3.2.4 Non-functional Requirements

When looking at the requirements a softare architecture has to fulfill, there can be destinguished functional, as well as non-functional requirements [10]. Non-functional requirements are compared to functional requirements more soften than functional requirements. At first an overview of non-functional requirements is given. ISO 9126 handles software quality charactaristics, but

it mainly handles non-functional requirements. The following shows all the non-functional requirements as defined in ISO 9126 [22]:

- Functionality
- Reliabilty
- Usability
- Efficiency
- Maintainability
- Portability

All these non-functional requirements lead to decomposition criteria. The reason, why these requirements lead to decomposition criteria is based on the following: Caused by the non-functional requirements, the software architecture is influenced vitally. As every requirement directly affects the structure of a software architecture, it also affects every aspect, that leads to decomposing a system. Thus each non-functional requirement has such an enormous effect, that each of them contribues, how a system is decomposed [23].

Taking one of these non-functional requirements, it is possible to give a good example, how these non-functional requirements influence a software architecture. Let's take the non-functional requirement *reliability*, for example. In case, reliability is demanded by software system, this influences the software architecture the following way: The software architecture must be designed, in order to guarantee, that the minimum required level of reliability is provided. E.g. such a level can be written down in a contract. In this case, it is possible to define an service level agreement that contains all necessary information in order to be able to describe the required level of reliability. Furthermore such a contract shall contain a number of metrics, that are able to charactarise the quality of the reliability requested. When reliability is requested, maybe it is necessary, to modify the architecture of an existing system or it is inescapable to change a planned new system. A *normal* software system uses standard methods in order to fulfill its requested service. A system, that has to be more reliable, than the *normal* level, requires a software architecture that must be extended compared to a standard architechure. Imagine there is a single-point-of-failure that is represented by one server, that runs the software solution. In case, the performance of this server is sufficient in order to handle the whole system load, this server represents such a single-point-of-failure. If this server fails, this implies, that the whole software system fails. But in case, there is a backup server, that is able to replace the main server, this backup server is able to guarantee, that the whole software system doesn't fail. But in order to be able to use such a backup strategy, the software architecture must be capable to allow such a solution. This leads to a software architecture, that must be enhanced, compared to a standard software architecture. E.g. replicating data between both servers is necessary. Another point is a mechanism, that is able to recognise whether the main server is no longer working. In this case, the backup server must handle all further actions.

When creating a software architecture, that is capable to provide reliability, the example above is only one aspect that influences a software architecture. Many other things can be required, in

order to guarantee that a system is reliable. When having a look at all the other non-functional requirements, similar scenarios can be found. E.g. when demanding *Efficiency* there are many ways, that affect a software architecture. The same thing is present, when taking one of the other non-functional requirements.

Discussion

When having a look at all the non-functional requirements, they influence directly or indirectly a system's decomposition. When having a look a the list above, there is to say, that here exist many more non-functional requirements. But these one are the most important. In case there are special requirements this definately influences the software architecture. Furthermore non-functional requirements are very powerful. Simply saying that a requirement must be fit may change a software architecture fundamentally.

3.2.5 Functional Requirements

All the functional requirements result in decompostion criteria as well. Normally in no project it is possible to fit every requirement, because it is either not possible or it increases cost and the amount of time for this project dramatically when trying to fit as many requirements as possible. There are functional requirements, that are more considerable, than others. This sections describes only few groups that functional requirements may belong to. This is caused by the huge number of groups that are available. There is a focus on the groups, that are in my opition the most important functional groups. When decomposing a software architecture the functional requirements lead to changes, because in case there are special modules or features. This is in order to enable the software architecture to cope with this requirements. Furthermore a requirement can cause serious problems when combining with other requirements. Detailed information about such situations are described in section 3.4. Section 3.4 gives an overview about problems, that occur, when different requirements compete against each other.

- Persistance
- Business Logic
- Integration
- Distribution
- Communication
- Localisation
- Workflow Management
- Security
- Logging
- Exception and Error Handling

- Transactions
- Flexible Configuration
- State and Session Handling
- Validation
- Model Driven Development

The list above describes a number of various criteria groups, that occur within developing a software architecture. It is possible to assign a functional requirement to one of these groups. These criteria groups are defined more precisely in the following parts. The criteria that is described in the following is *Persistance*.

Persistance

Persistance describes everything that handles storing information. Every software has to store data. This can be a simple file or e.g. a database. Persistance describes how this data is stored. In this case there is a system, that has a multi-tier architecture. One possibility of such a multi-tier is a three-tier architecture. For example, in a client-server architecture, normally there is a three-tier architecture. The user interface represents one layer, as well as the business logic represents one layer. The third layer in this case is the data tier. Persistance fully fits this layer. It is a good idea to make a persistance layer in present software architectures. But using a persistance layer can cause some ugly effects that interferre with some parts within the persistance layer.

Characteristics

- Often used with an Object-Relational-Mapper for data bases systems
- Reducing complexity when storing data

Motivation

- Data within an application has to be stored persistantly
- The stored data are often very valuable
- Data that exceed a systems memory must be stored

Challenges When modelling data structures, the combination with a object-relational mapper may cause some trouble. Normally, there must be a decision whether to prefer performane, redundancy or simplicity. This is caused by various ways how to map an object to a relational database. Another point can be side effects caused by data base triggers. In such a case, the data base may execute instructions that interferre with the persistance layer. This must be considered, when designing a persistance layer. Furthermore the requirements to a persistance layer may require specials conditions concerning other hardware and/or software.

Discussion

In the section above there is a number of functional requirements given. Of course this list is not complete. A functional requirement is very specific, so there is no possibility to say, which effect this requirement has to the software architecture. The example above, describes one group of functional requirements. Every group that is described in the list above leads to specific changes to the software architecture. Taking everything into account, there is to say, that functional requirements normally influence a software architecture in a very specific way.

3.3 Information Sources

When developing a new software architecture, there is a need for getting information, what needs a software architecture exactly has to fulfill. But often getting this information can be very difficult. In order to get this information there is a number of ways. The following list points out a number of sources in order to gather information that is relevant for creating a software architecture [21]:

- Stakeholders
- Requirements
- Specification
- Risk Management
- Resources
- Interfaces

In the following, the sections 3.3.1 - 3.3.6 describe these ways where to get useful information.

3.3.1 Stakeholders

A stakeholder is interest in or has a gain upon a successful completion of a project, and it may have influence to the project completion in a positive or negative way. So let's have a look at all the possible stakeholders involved into a software project:

- Software Developer
- Responsible IT-Staff
- Client
- Law
- Employee
- Boss

- IT-Regulations

Looking at this list of stakeholders it is evidently, that many various people have influence to a software architecture. As described in the top part 3.2.5 and 3.2.4 every decomposition criteria is influenced by the requirements to the software architecture. As every stakeholder impacts the requirements, each stakeholder also impacts all the decomposition criteria. Thus it is possible to say, that every stakeholder is helpful, in order to detect further decomposition criteria. A common approach is to interview a stakeholder in order to get its expectations about the software project.

Let's have a look at an example: A stakeholder, in this case the responsible person for the central IT services of the company is involved in introducing a new software. This software shall comply with some specified targets. In this case, the manager of the central IT services wants the new software to have some specific features. Another possibility is, that the responsible person wants that, when rolling out the software, some kind of functions must be guaranteed. So the new software needs to cope with this needs. The IT manager knows what kind of functions have to be guaranteed, as well as which other functions are important when developing a new software or when deploying this new software. The same problem is caused by all the other stakeholders. Every stakeholder has special requirements, so it is imporant to acquire these requirements.

3.3.2 Requirements

Of course the central information source are the requirements of a software project. These requirements vitally influence a software architecture and therefore the decomposition criteria. The following list represents a list of fields, that contain valuable information

- Goals
- Use Cases

Goals describes the analysis of stakeholders or scenarios. These may contain information that are necessary for the software system. nother possibility to gather information are *Use Cases*. E.g. use cases provide an overview how an user accesses information or submits new data into an software system. This may provide information how to decompose a software architecture.

3.3.3 Specification

The specification is also a potential source of information. As the specification contains many information about the exact requested behaviour of the software, there is no way to ignore the specification in order to detect decomposition criteria. A specification contains a huge number of functional requirements. Thus, a specification provides at first functional requirements, that are vital to decomposing a software architecture. It is possible, to assign a functional requirement to one of the groups, that are described in 3.2.5. As there are a number of various types of specifications, each of them may be useful to identify decomposition criteria.

3.3.4 Risk Management

Furthermore it is very important to consider the risk management of a software project. The risk management describes all the things that might go wrong. It estimates the probability that

something could go wrong as well as the impact of the failure. In case of developing a new software architecture, the risk management considers a number of risks that can occur. The following list gives a few of them [21]:

- Tight Time Table
- Technical Infrastructure
- Bad Requirements
- Restricted Knowledge
- High amount of Changes

Having a look at the risk management that concerns software development, a number of risk can be identified. This leads to a different view that affects the software architecture.

But risk management also takes place within a company. During the whole cycle, a software products "lives", it is possible, that a number of risks appear. These risks maybe are caused by the company in general. The corporation-wide risk management is able to handle completely different situations. So not only the risk management that is required when developing some new software is interesing, but also the corporation-wide risk management.

In developing safety critical system and products, there is a risk management in order to reduce the risks of using a product. For example ISO 14791 for medical products [24].

3.3.5 Resources

Moreover all the different resources that are involved into the software project are necessary to recognise whether the software architecture is influenced by these resources. As described in 3.2.3 resources often limit a software architecture. In order to find out what limitations exist, there is a number of sources that can help to identify such resources. Having a look at the current technical descriptions may list a number of such limitations. Another way is asking managers of former projects, or having a look at the current IT infrastructure. Another way, to gather information about potential limitations caused by technical reasons, is to have a look at other technical systems, that already exist. Either within the company or outside. Another possibility is the quality management within the company. Maybe there are experiences with some kind of technologies that affect a resource, or other factors concerning technical factores, that are already known by the quality management.

3.3.6 Interfaces

Another information sources are all the interfaces that are used in the current system or other systems that the new one has to interact with. Information about interfaces are written down in the specification, or other documents. In case there is an open interface, a standardisation organisation may have written down the properties of an interface. An example for such a case are all the RFCs, as well as ISO. When having a look at an interface, it is possible to recognise effects that an interface may have on the software architecture that is developed. E.g. a web service is

an interface, that can be used to identify how a software architecture can be decomposed. Even the fact, that there exists a web service may have influence to the software architecture.

3.4 Impact on each other

When having a look at all the factors, that influence a software architecture (See 3.2.2 - 3.2.5 on pages 26 - 34), there can be determined, that there is a huge number of factors. When decomposing a software architecture, some of theses factors may clash on each other. Many of these factors work together without any problems indeed. In addition they may be based on each other or even support each other. Another possibility is, that they don't care each other. E.g. neither factor a influences factor b, nor b influences a. In case two factors clash on each other, it is necessary to decide which one of those factors is more important than the other one (See 3.5). A more exact description of this situation is given in 3.4.2. Furthermore the opposite case is described below in 3.4.1.

3.4.1 Providing each other

At first there can be determined criteria, that can be used together without any problems. In case these criteria lead to the situation, that they profit from each other, they is a really good fact. An example for two criteria that fit to this description is *Portability* and *Maintainability*. Those two are both non-functional requirements. When considering portability as well as maintainability, it is obvious, that they support each other. Maintainability requires an architecture, that enables chaning a software architecture more easy. Thus everything that fits to this requirements is covered by maintainability. Furthermore it is necessary to modify a software project in order to be able to make the new software system fit to the new requirements. So it is an advantage if the software architecture is googe maintainable. The same is for other criteria that have the same characteristics.

3.4.2 Clashing each other

This case describes two factors, that cannot exist at the same time. Otherwise two factors cannot provide the same level of quality at the same time, because they work contrary. For this situation it is really simple, to get an example. When having a look at non-functional requirements, it is easy to identify two factors. In this case choose *Performance* as well as *Portability*. Imagine there is a software system that must be optimised. So it is possible to optimise the software system using hardware specific code. The usage of hardware specific code enhances the performance of a software system. The disadvantage of using such code is the fact, that this makes it more complicated to transfer the software system onto a new hardware platform. so either a high performance or protability is possible.

3.5 Making Decisions

Of course it is interesting what kind of factors are responsible in order to make a decision concerning a software architecture. When having a look at 3.4, it is not possible to fit every requirement. So it is necessary, to decide for one thing out of many possibilities. Not focusing on few things, stands for increased complexity as well as increased time and cost for developing the new software architecture. There is an enormous amount of different factors, that influence a software architecture (See 3.2.2 - 3.2.5 on pages 26 - 34). But not each of them requires to make a descision that affects a software architecture. Some of them influence a software architecture, but they don't require a decision. For example, having a limited size of main memory in a computer system leads to adapted software for this situation. There is no way to decide whether to fulfill this requirement or not. Furthermore there are many factors, that behave the same, as this example.

But there are factors, that require to make a decision, of course. When having a look at 3.4, there are non-functional requirements that perfectly fit to such a situation. E.g. taking *performance* and *fexibility*. As it is not possible, it is required to decide which one of those two non-functional requirements is more important. Based on the information sources, as described in 3.3 on page 36 it is possible to gather this required information. Dependend on the further plans for a software architecture, it might be a good idea to consider which feature is more important in front of the development. It is also a good idea to consider section 3.6. This section contains information how to prioritise different criteria.

Furthermore it is necessary, that the responsible person for making such a decision is able to handle this problem. But based on the available information, as well as the factor, that there are requirements, it is necessary to have a look at each single existing project, in order to be able to make a decision. All the information sources as described in 3.3 on page 36 lead to a decision in order to solve the situation. These information sources gather all the factors that have an impact to the software architecture, and so they provide the responsible person in order to make the correct decision. As in 3.2 is described what kind of factors influence a software architecture, every aspect is considered.

3.6 Making Priorities

It is not possible to satisfy every requirement the architecture should have. As this is not possible, it is necessary to give some requirements a higher priority than other ones. Often it is feasible to acquire one requirement as good as possible, without influencing other requirements. Thus, it is not necessary, to completely remove a criteria. Regarding to different criteria, it is possible to decide whether a criteria is more important than an other one. This is based on the field a criteria belongs to as well as on the kind of criteria.

E.g. there is a limitation in view of the resources, when trying to develop a system that needs to be highly available. When developing a software architecture it is necessary to know the importance of each criteria, because this influences making decisions (See 3.5). So when there is only a limited amount of hardware resources, the system has to run with, it maybe is not possible to comply with the requirements regarding the availability. Limited resources in this

case may be the number of systems, the amount of processing power, or the number of different storage systems. Otherwise it is not possible to run the system with all the available resources, because they are limited. Another example for limited resources is the amount of money, that is available to run the hardware platform. Thus it is not possible to run such a system without prioritising various criteria. So it is helpful to be able to make a decision, because priorities are defined for each criteria. In case there are no priorities for each criteria, they must be assigned when preferring one critia instead of another.

3.7 Conclusion

Taking everything into account, it is possible to say, that everything that has to do with a software project, directly or indirectly influences a software architecture. So when decomposing a software architecture, everyone of this aspects has an effect on the software architecture. During the whole development cycle of a software architecture they affect a software architecture. When gathering information for developing the software architecture, when prioritising different criteria and when deciding what kind of criteria is more important, every factor influences this situations. And thus the software architecture that is developed. These criteria cover everything from organisational issues (See 3.2.2), over technical issues (See 3.2.3) and non-functional requirements (See 3.2.4) to functional requirements (See 3.2.5). All these criteria influence the development of a software architecture. These criteria may support each other or they they do the opposite. As it is not possible to fulfill every requirement it is necessary to decide which are the most important one and then develop the software architecture by considering only these decomposition criteria.

4 Component-based Software Development

Kamil Fabisiewicz
fabisiew@in.tum.de

Abstract. Component-based development, also called componentware, is the development of software systems by using components as the essential building blocks. The most common understanding of a component is that of an encapsulated unit of software with well-documented and therefore well-understood interfaces, connecting components to each other. This paper describes some key aspects of componentware. Starting with an historical view and the basic ideas behind components, their essential properties are identified. Furthermore, some practical issues like componentware process models, component design and composition, as well as component infrastructures are discussed. The paper concludes with an overview on the current state of componentware, as well as some open research issues.

4.1 Introduction

The idea of component-based software development can be traced back to the year 1968. At that time for the first time costs for software exceeded hardware costs. More and more software projects failed, the term "software crisis" was created. This issue was discussed at a conference convened by the NATO in Garmisch-Partenkirchen [25]. The application of a systematic, disciplined and quantifiable approach to the development of software was requested. A new discipline called "software engineering" originated.

The cause of the software crisis was, and still is, the increasing complexity of software, as Dijkstra stated in 1972:

> "The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem" [26].

Previously used techniques in software development have not been sufficient at that time in order to catch the ever increasing complexity of software to control [27]:

- Low level programming languages offered no suitable constructs in order to map functionality at an understandable level of abstraction.

- There was a lack of methodologies, processes and standards in implementation of software projects. Both on development and on project management level.
- There were no powerful tools to support the developer in implementing and designing software.

McIlroy's solution to the crisis was an approach called component-based software development, he presented in a paper published at the NATO conference mentioned above [28]. His idea was to divide software systems into components in order to deal with the complexity. Furthermore he predicted a market of mass-produced components allowing software developers to buy needed functionality in form of units which can be composed up to whole systems.

Meanwhile, there are more precise definitions of component-based software development (also called componentware) available. A simple but useful comes from Bergner et al.:

> "Componentware is concerned with the development of software systems by using components as their essential building blocks" [29].

Since those early days software development has made enormous progress. Advanced and standardized technologies, methodologies and processes simplify the writing and designing of software today. So why do we need components? Software development is still difficult: It is widely agreed that there is no "silver bullet" – that is, no single approach which will prevent project overruns and failures in all cases [30]. In general, software projects which are large, complicated, poorly-specified, and involve unfamiliar aspects, are still particularly vulnerable to large, unanticipated problems. There are various reasons for this.

During the recent years, more and more different application domains such as mobile computing, automotive and e-Business emerged. In parallel, the importance and complexity of software in all domains increased: from the automation of business processes to safety-critical functions in embedded systems.

Object-oriented programming has become a key technology in software development increasing reuse, encapsulation and independent development. However, even though it is doing quite well in a number of ways, it also suffers from several drawbacks [27]:

- Objects only compose and cooperate if written in the same language.
- Objects cannot be deployed independently.
- Objects are tightly coupled, as they execute in the same process and data space.
- Concurrency is not well-integrated at all with the ideas of abstraction in classes.

Components on the other hand, promise to eliminate all these disadvantages bringing an wide-ranging degree of benefits for the software itself and the organization developing it [27]:

- Increase reuse: Preventing developers from reinventing the wheel. Usage of commercial-off-the shelf-components allows to concentrate on the essentials.
- Increase quality: Frequently used and tested components are relatively error free.

- Outsourcing: Subproblems packaged into components can be solved by domain experts.
- Maintenance, extensibility and configuration: Clear system structures, easy creation of variants and product families.
- More efficient development process: Faster prototyping, increased productivity from existing components.

All of these advantages of components will lead to cost and time reduction in software projects, while increasing the quality of the developed software. This in an important economic issue, helping us to reduce the problems of the software crisis.

4.2 Software Components

This chapter covers the fundamental non-technical underpinnings of componentware. Starting with a term definition, the key properties of components are discussed, especially reuse as the main concept of componentware. A differentiation from other - often confused - terms is also made. The chapter closes with the description of a componentware suited process model, and a brief overview of Catalysis and UML Components.

4.2.1 Definitions and Characteristics

After 40 years of research there is still no consensus about the question, what exactly a component is. Everyone talks about components but in many cases people talk about and address different issues. The goal of this section is to establish some degree of order and intuition as a basis for further discussions. For this, we fall back on the most frequently cited definition from Szyperski:

> "A software component is a **unit of composition** with **contractually specified interfaces** and **explicit context dependencies** only. A software component can be **deployed independently** and is subject to **composition by third parties** [31]".

Based on this definition, in the following the key features of components will be discussed in more detail:

- **deployed independently:** A component has to be well separated from its environment and other components. It needs to be sufficiently self-contained encapsulating its constituent features. Also, as it is a unit of deployment (.dll e.g.), a component will never be deployed partially. Note: Other authors like Siedersleben [32] consider the property of components being units of deployment as optional.
- **contractually specified interfaces:** A component exports (implements) one or more interfaces, which are contractually guaranteed. Furthermore it imports required interfaces. Components hide their implementation and can thus be replaced by other components implementing the same interfaces.

- **explicit context dependencies:** The definition of a component requires specification of what the deployment environment will need to provide so that the component can function (context dependencies). This includes the component model that defines the rules of composition and the component platform that defines the rules of deployment, installation, and activation of components.
- **composition by third parties:** Components can be composed into other components or a functioning system. A third party is one that cannot be expected to have access to the construction details of all the components involved.

All these properties serve the purpose of reuse. As a key feature of components, reuse has to be examined more closely [31].

Whitebox versus blackbox reuse

Blackbox reuse refers to the concept of reusing implementations without relying on anything but their interfaces and specifications. For example application programming interfaces (APIs) reveal nothing about the underlying implementation. In contrast, whitebox reuse refers to using a software fragment, through its interfaces, while relying on the understanding gained from studying the actual implementation.

Whitebox reuse however is problematic because of routines that can be replaced by a new release. Such a replacement will probably break some of the reusing clients, as these depend on implementation details that may have changed in the new release. Therefore whitebox reuse of components should generally be avoided.

Component "weight"

In order to maximize the reuse of a component, redundant implementations of secondary services within the component have to be avoided. It is desirable to "outsource" everything but the prime functionality that the component offers itself. This approach however has one substantial disadvantage – the explosion of context dependencies. In a world of different environments with different configurations and version mixes, with each added context dependency, it becomes less likely that a component will find clients that can satisfy the environmental requirements. On the other hand, a component could come with all required software bundled in, but that would clearly defeat the purpose of using components in the first place.

This optimization problem resulting from trading leanness against robustness as shown in figure 4.1 can be summarized in the following statement:

Maximizing reuse minimizes use.

This problem can be shifted toward leaner components by improving the degree of normalization and standardization of runtime environments. The widely supported a particular aspect is, the less risky it becomes to make it a specified requirement for a component. Context dependencies are harmless where their support is ubiquitous. This leads to the need of wide spread and standardized component runtime environments, which will be discussed in a subsequent section.

4.2.2 Differentiation from other terms

The term "component" often serves as an substitute for the terms "class", "module" and "object". Furthermore, component-based development is often associated with object-oriented pro-

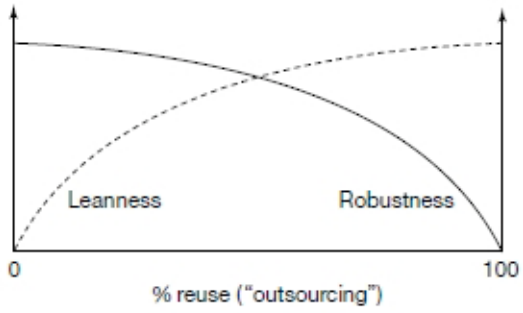


Fig. 4.1: Opposing forcefields of robustness (limited context dependence) and leanness (limited "fat"), as controlled by the degree of reuse within a component [31].

gramming. Although components are related to these concepts, in order to prevent confusion, a distinction has to be made. In the following, these terms are briefly explained, and related to each other [31].

- **Object-oriented programming:** Although object-oriented programming supports some of the key concept of components like reuse and encapsulation, there is no need for a component to contain classes at all. Instead, a component could contain traditional procedures, or it may be realized in its entirety using a functional programming or any other approach.
- **Class:** A component may contain multiple classes, but a class is necessarily confined to being part of a single component. Classes can depend on other classes using inheritance, components can depend on other components.
- **Object:** Objects are a units of instantiation, with a unique identity. In contrast, components are units of design with no state or identity. A component acts through objects at runtime.
- **Module:** Modules are design units, used to package multiple entities, such as classes to one unit. However, modules do not have a concept of instantiation, whereas classes do. A module can be compiled into one unit (e.g assembly). Note: Java doesn't have a separate module concept as Java-packages are translated into multiple class-files. A component may consists of one or more modules.

4.2.3 Process models

For a common understanding, the following definition of the term process model is used in this paper:

> "A process model supports system development by clearly defining individual development tasks, roles and results as well as the relationships between them" [29].

Componentware deals with reuse of existing software. As such, it does not fit very well with many of the existing methodologies tailored to developing software from scratch. Instead, development should be organized according to a process model adapted to componentware.

There are several reasons for this [29]:

- Current methods do not sufficiently take into account issues of reuse and customization. This leads to a very low reuse rate in practice. To reach a higher rate of productivity, the introduction of new results elaborated by individuals or groups in new roles is needed. This leads, for example, to the separation of the roles component developer and component assembler, and to new development results like a component repository, a market study, or an evaluation document for interesting commercial components.

- Furthermore, the rigidity of traditional prescriptive process models is widely felt as a strong drawback, and there is common agreement about the need to adapt the process to the actual needs during a project. A flexible process model should be more modular and adaptable to the current state of the project, much in analogy to the essential properties of components and componentware systems themselves. With componentware, this issue is very important, as changes on the component market, for example, may require changing the development process.

- Traditional top-down approaches start with the initial customer requirements and refine them continually until the level of detail is sufficient for implementation. Pure top-down development usually leads to systems that are brittle with respect to changing requirements because the system architecture and the involved components are specifically adjusted to the initial set of requirements. This is in sharp contrast to the idea of building a system from truly reusable components.

  A bottom-up approach, on the other hand, starts with existing, reusable components. They are iteratively combined and composed to higher-level components until, finally, a top-level component emerges which fulfills the customer's requirements. Obviously pure bottom-up approaches are impractical in most cases because the requirements are not taken into account early enough.

  Hence, neither a pure top-down nor a pure bottom-up approach is adequate for the development of large applications. To overcome the deficiencies of both extremes, the componentware approach has to combine top-down and bottom-up development as equivalent parts complementing each other.

According to Bergner et. al. [29], component-based development does not introduce a totally new paradigm, but rather builds on existing, well-proven techniques like object-orientation. Hence, the methodology for componentware should not be totally new, but adapt, improve, and combine well-known object-oriented methodologies.

Based on all this observations, a componentware process model using process patterns (each pattern describes the context wherein it can be applied, the problem it solves with prerequisites, constraints, and implications) is presented in [29]. It focuses on tasks, their deliverables, and the dependencies between them. Allowing to use this methodology with several process models, like the spiral model.

Figure 4.2 illustrates the different tasks of this componentware development process. Each task consists of several subtasks, which is requiring and/or producing development results. The

produced documents and other development artifacts serve as interfaces of the main tasks, analogous to "real" interfaces of software components. The connections between the tasks, namely, the consistency conditions and the flow of structured development information, are visualized by thick, grey arrows in figure 4.2. Subtasks are closer coupled than the main tasks and are usually developed together.

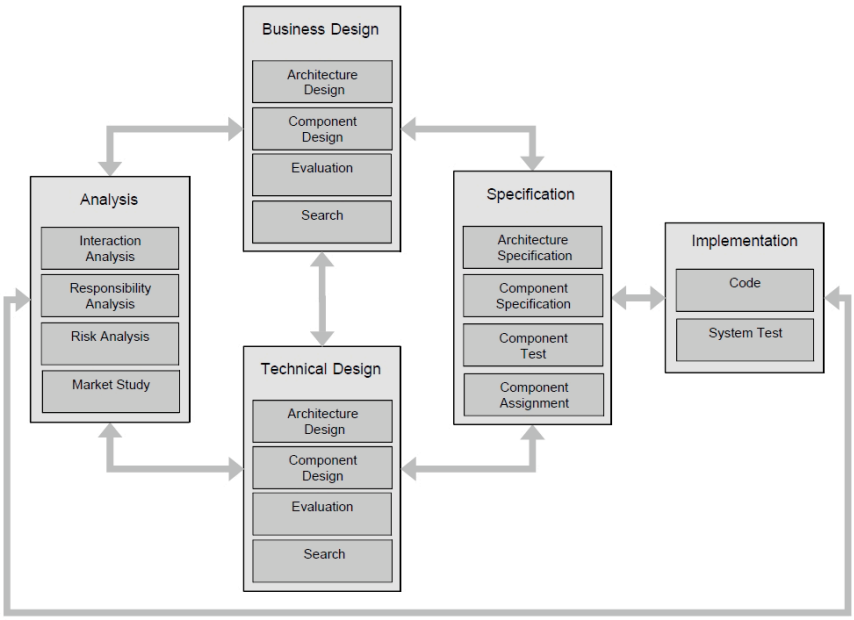


Fig. 4.2: Component-Oriented Process Model [29]

In contrast to traditional process models, no particular order on the temporal relationship between the development tasks and their results is defined. Instead the process should be adapted to the current state of the project which is partly determined by the current state of the development documents. According to this state, a given development context, and a set of external conditions, process patterns provide guidelines about the next possible development steps. Details about the proposed result structure and the pattern-based approach can be found in [33]. In the following, the main tasks (resp. results) are discussed in more detail.

- **Analysis:** The main result of this task contains the specification of the customer requirements. In particular, the interaction between the system and its environment, expected functionality of the system with respect to the functional and non-functional user requirements, the benefits and risks associated with the development of the system under consideration and a market study with information about existing business-oriented solutions, systems, and components.
- **Business Design:** Business Design defines the overall business-oriented architecture of the resulting system and specifies the employed business components, providing detailed

specification of the business-relevant aspects, interactions, algorithms, and responsibilities of the system and its components. It also contains a preselection of potentially suitable business components and standard business architectures that are subject to a final selection within the specification main result.

- **Technical Design:** This task comprises the specification of technical components, like database components, for example, and their overall connection architecture which together are suited to fulfill the customers non-functional requirements. It deals with technical aspects of the system like persistence, distribution, and communication schemes.
- **Specification:** The Business Design and Technical Design are concerned with two fundamentally different views on the developed system. The Specification main result merges and refines both views, thereby resulting in complete and consistent Architecture and Component Specifications. It also contains test logs (with respect to the user requirements) of the components which were preselected in the Business and Technical Design. The Component Assignment sub-result specifies which components are to be developed in the current project and which components are ordered from external component suppliers or in-house profit centers.
- **Implemenation:** The results of the Implementation task are the code of the system under consideration and the system test. This subtasks may be performed concurrently, influencing each other.

Moreover, the following specialized roles were introduced:

- **Component Developer:** Components are supplied by specialized component developers or by in-house reuse centers as a part of large enterprises. The responsibilities of a Component Developer are to recognize the common requirements of many customers or users and to construct reusable components accordingly. If a customer requests a particular component, the Component Developer offers a tender and sells the component.
- **Component Assembler:** Usually, complicated components have to be adapted to match their intended usage. The Component Assembler adapts and customizes pre-built standard components and integrates them into the system under development.
- **System Analyst:** As in other methodologies, a System Analyst elicits the requirements of the customer. Concerning componentware, he also has to be aware of the characteristics and features of existing systems and business-relevant components.
- **System Architect:** The System Architect develops a construction plan and selects adequate components as well as suitable Component Developers and Component Assemblers. During the construction of the system, the System Architect supervises and reviews the technical aspects and monitors the consistency and quality of the results.
- **Project Coordinator:** A Project Coordinator as an individual is usually only part of very large projects. He supervises the whole development process, especially with respect to its schedule and costs. The Project Coordinator is responsible to the customer for meeting the deadline and the cost limit.

Although this approach is a good example of a modular and adaptable process model suited for componentware, it is however still under research and thus not widespread in practice. Two other, more common, component-based model processees are Catalysis [34] and UML Components [35], briefly described in the following:

Catalysis was one of the first component-based software engineering processes. It combines an iterative and incremental process with a component notion and UML concepts. Basic building blocks of Catalysis are objects and actions. Objects can represent everything and can be structured hierarchically. Actions are interactions that can occur between all types of objects. Similar to objects, actions can be structured hierarchically. There are two core principles of Catalysis: abstraction and refinement. The refinement relation enables the traceability of the model from a higher level of abstraction to a lower level. Another core principle is precision. A model has to be defined precisely, that means unambiguously at all level of abstraction. Catalysis does not propose a special process, but the use of suitable process patterns. This is to keep the approach applicable to different application domains. However, in Catalysis, the component notion is not clearly defined. It is a kind of mixture between object and package. While the basic principles of Catalysis are still useful, the approach is difficult to understand and apply.

UML Components is a process centric componentware approach with two main objectives: The definition of a simple process to identify and specify components and the definition of an application of the UML notation to support the process. The process is an adaptation of Rational Unified Process (RUP) [36] to component-based development. In detail, the RUP phases analysis and design are replaced by new phases specification, provisioning and assembly. The approach defines activities and artifacts for component identification, interaction and specification. UML components does not define a component notion itself, it rather refers to properties a component should incorporate.

It has to be said that usually process models incorporate a description technique, are more or less tool supported and define an informal mapping to existing component frameworks. Almost all of it incorporate UML as a means of design and architecture modeling. However, UML is rather object-oriented than component-based, which leads to ambiguities between the process component notion and the mapping to UML constructs. Furthermore almost all approaches lack a formal system model or an satisfactory mapping to existing component frameworks [27].

4.3 Components as building blocks of software architecture

This chapter consists of some technical aspects of component-based software development. First, the construction of components is covered, including the key characteristics of interfaces and a short view on software categories. In the second subsection, the composition of components and the resulting dependencies between them are discussed. Finally, a brief overview of the two component infrastructures COM and EJB is given.

4.3.1 Component construction

Component construction is about finding proper components and defining interfaces. Both topics are discussed in the following.

Interfaces

The concept of a component interface is central to componentware. It allows to use the functionality of a component only through clearly defined access points. In other words, interfaces connect components to each other (program interface) or components to the user (user interface). Where each component provides no, one or multiple user interfaces. A program interface can be implemented using various programming languages, like Java or C. It is important to distinguish between the term "(program) interface" and the interface construct, existing in many modern object oriented programming languages. A program interface, implemented in Java for example, may consist of:

- One or multiple Java-interfaces.
- All classes and types needed for using the Java-interface, like exception types or parameter types.
- Other, such as configuration files.

In the simplest case, a program interface written in Java, would consist of only one Java-Interface or Java-Class.

Program interfaces contain everything a programmer has to know and understand, in order to use a component. In detail, this consists of a signature part, describing the operations provided and used by a component (syntax), and a behavior part, describing a components dynamic behavior (semantics). While the syntax of an interface can be expressed by means of a programming language, there is still no generally accepted way of documentation for the semantics. However, there a several, mostly programming language specific, approaches like OCL, JML or Spec#. But what should be described at all? What not? Siedersleben [32] suggests four assumptions that should be made during interface specification, allowing a simple and practical specification:

- **Exceptions** should not be specified, as they are dependent on the implementation of the interface.
- One should specify only for a **single process**, ignoring concurrency.
- **ACID-Behavior** (atomicity, consistency, isolation, durability) is assumed for every operation.
- **Repeatable read** (using the same parameter, multiple calls of the operation return the same result) and **restricted repeatable read** (operation is repeatable, as long as the corresponding object was not changed; List.get() or List.size() e.g.) are assumed for every operation.

Furthermore, four elements are presented, an interface specification should contain:

1. **State model:** The behavior of an interface should be described by a state model.
2. **Invariants:** Invariants are conditions a component met at all times.

3. **Pre- and postconditions:** Unlike invariants, pre- and postconditions apply only in some situations. Every invariant implies as a pre- or postcondition.
4. **Test cases:** Test cases are sequences of calls of an interface, with a predefined result. Test cases are intended to exemplary represent even complex processes and relationships between different operations.

An description language, designed to fulfill all this requirements is QSL (Quasar Specification Language) [32]:

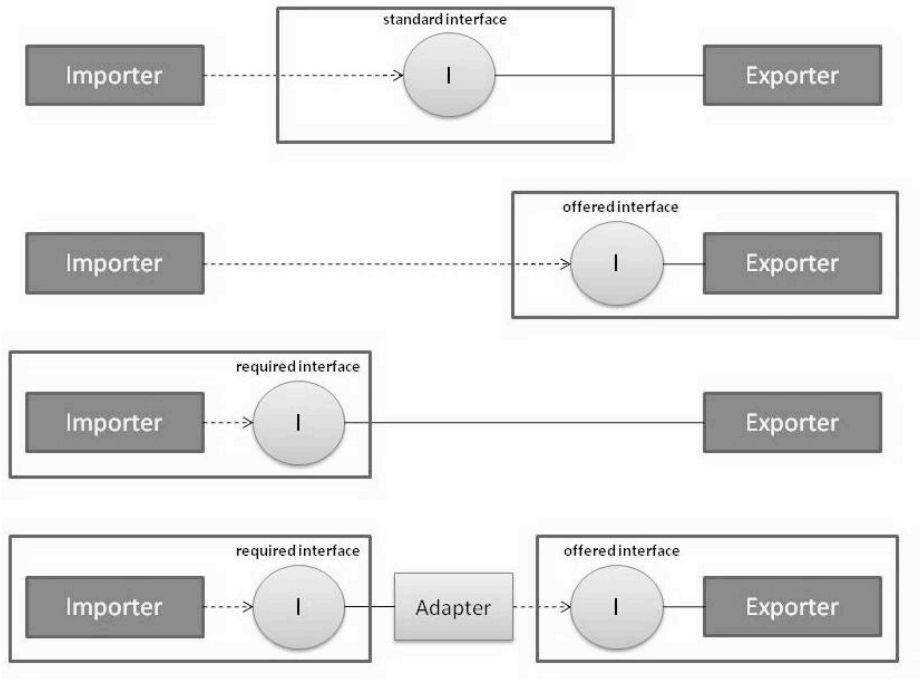


Fig. 4.3: Who defines interfaces? Four opportunities. [32]

Interfaces are separate constructs, they exist independently from implemented components. Therefore the question arises, who defines interfaces? As described before, a component consists of exported and imported interfaces. Taking this into account, Siedersleben [32] categorizes interfaces as following:

- **Standard interface:** Standard interfaces are well known and generally accepted by all programmers (Java-List e.g.). By using standard interfaces, an optimal decoupling of importer and exporter is achieved, as the exporter (who implements the interface) can be replaced easily. Moreover, every standard interface meets the following criteria:
  - It is readily available. For example, every programmer has it in his classpath.
  - It is enclosed. All types used by the interface are defined within it, or are themselves standard interfaces.

  - It is fully documented. There are tests for validation of implementations.
  - There are at least two implementations of it: one real and one dummy implementation for test purpose.

- **Offered interface:** Often the component which exports and defines an interface is already existing. Thus, the importer has to be implemented. In this case, the dependencies between the importer and exporter are determined by the offered interface. As the importer also has to reference all the types required by the interface, this may lead to undesireable dependencies.

- **Requested interface:** An requested interface is defined by the importer. It is suited to his needs and therefore very unlikely that this interface has been implemented already. As all requested interfaces doesn't have to be imported, an optimal decoupling between importer and exporter is accomplished.

- **Adapter:** An adapter is an specialized component, mapping imported and exported interfaces of two components and therefore decoupling them. In the simplest case, an adapter delegates method calls to the exporter. Often, however, it performs additional tasks like data transformation, customizing or exception handling.

All possibilities are outlined in Figure 4.3. Note: Every interface is either standard, offered or requested, regardless of whether this interface is imported or exported.

Component categories

Obviously, there is no algorithm, dividing a system into components. Many software architects rely on intuition when it comes to find an meaningful system structure. Siedersleben [32] purposes an approach based on the analysis of dependencies and the principle separation of concerns. He separates technical software from business software and defines a classification scheme for software in business information systems. Four categories of software according to their dependencies and their implemented concepts are defined [27]:

- **0-software** depends on nothing. Examples are class libraries such as STL in C++ or java.util. 0-software is ideally reusable, but it is of no use its own.

- **A-software** implements a business concept and depends on that concept only. Examples are classes such as customer, invoice or contract.

- **T-software** implements a technical concept and uses a lower level technical infrastructure. For example, a database access layer that uses JDBC is T-software.

- **AT-software** implements technical and business concepts at the same time, and depends therefore on both. Hence AT-software has to be modified all times, when business or technical requirements change.

Software-categories can be applied to componentware. 0-components, T-components, A-components and AT-components can be distinguished. This sets up a quality criterion and design heuristics for software architectures: AT-components should be avoided.

4.3.2 Component composition

This subsection covers the configuration of components, including some patterns of component composition, and the control of dependencies.

Configuration

Components decompose the overall system into manageable, understandable units, but how to combine components into an runnable whole? Designing and developing software against interfaces is an important property of componentware, but there has to be a place, putting the parts together: the composition manager.

The composition manager is itself a component, that is responsible for instantiation and binding of components. It exports an subset of interfaces, exported by its enclosing components and is itself composeable. There are several ways for the composition manager to obtain components:

- Calling a constructor.
- Indirect call to a constructor using a factory.
- Resolving and connecting to an existing component over an name service.

It has to be said, that every component has to offer an interface that is only used by the composition manager and therefore is not part of the programming interface. This configuration interface consist in the simplest case of only one method “bind”, providing the component with instances of its required components. That is how components are connected.

Siedersleben [32] points out that it is a good practice, configurating from back to front: starting with self contained components, which are independent from other components, ending with the main program. Furthermore, it is important, that the configuration happens only in a few places, allowing to keep an overview over all dependencies.

Composition manager can be hard coded, but as an stereotype and error-prone activity, this should be avoided. Instead configuration should be done by a generator which gains all dependency information from a configuration file, and based on this, instantiates and sets up all needed components automatically. This design pattern, called **dependency injection** is supported by many component frameworks like EJB 3.0 or Spring.

The composition manager typically composes its enclosing components hierarchically. Its like putting some components in a box, providing them with the needed import interfaces, whereas an arbitrary subset of the exported interfaces is made public as the composition managers export interface. A special case of this composition is the **facade** (figure 4.4), where different interfaces are combined into one export interface.

Multiple components, implementing the same interface, can be arranged according to the pipes&filter pattern (figure 4.5). This approach, called **symmetric composition**, provides a structure for components that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed from component to component, refining the result.

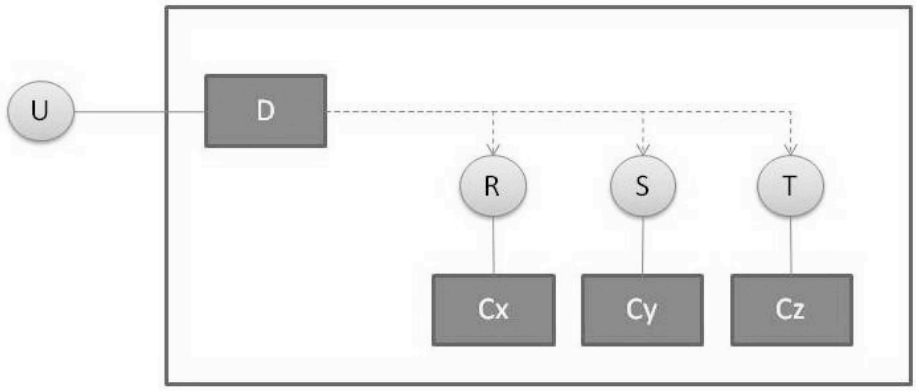


Fig. 4.4: Facade as a special case of composition. [32]

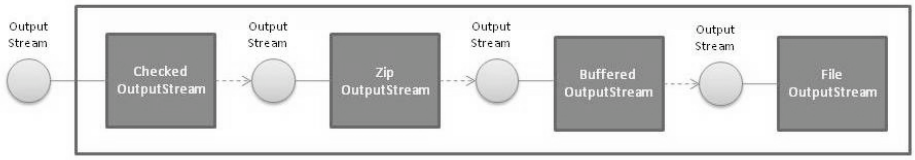


Fig. 4.5: OutputStream: An example for symmetric composition. [32]

Dependencies

According to Siedersleben [32], one of the most important tasks of a software engineer is the control of dependencies. The more dependencies are introduced into the system, the harder it is to change and to maintain. As a rule of thumb, is desirable to avoid cyclic compositions for several reasons:

- Components can be replaced more easily.
- Components can be omitted.
- Easier coordination of development teams.

Furthermore, there are two different approaches in the composition of a system of components [27]:

- A component knows the business concepts and/or the interfaces of the components it uses. The component imports these concepts as references in its documentation or as direct includes of the specified interfaces.
- Components are self-contained concepts and units of software. Apart from general stable business concepts and technical infrastructure, a component defines all concepts it uses on

its own. All interfaces the component provides and requires are a part of the component. The component defines the interfaces it expects from other components to provide, independent of the interfaces these components actually provide. The self-contained components are connected through extra channels (i.e. adapters), that provide a mapping between the interfaces and may define further contracts between the components.

The two types of composition differ in their introduced dependencies. The first and commonly used concept introduces dependencies between the components. If business requirements change, changes in interfaces and therefore in the providing and using components are likely. The second concept reduces dependencies, if interfaces change, only the adapters have to be changed. This approach can also be used, to decouple cyclic dependencies.

4.3.3 Component infrastructures

Technical component frameworks or component infrastructures focus explicitly on the support of component development and the specification of concrete component runtime environments. They include the component model that defines the rules of composition and the component platform that defines the rules of deployment, installation, and activation of components.

During the recent years three major component frameworks have emerged: COM[1], EJB[2] and CCM[3]. However, until now the CCM standard is not yet accepted in industry. Therefore only COM and EJB will be introduced in this section.

COM

History: The Common Object Model (COM), developed by Microsoft, is one of the oldest component frameworks still in use. COM, a descendant from the Object Linking and Embedding Technology OLE, was developed in 1995. COM was extended in 1996 to the Distributed Common Object Model, DCOM. DCOM enables the use of COM components in a distributed environment. Currently, COM and DCOM are combined with the Microsoft Transaction Service MTS to COM+ and provide the component framework for the .NET platform [27].

Basic idea: The basic idea behind COM is a language-neutral way of implementing objects that can be used in environments different from the one they were created in, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces component implementers to provide well-defined interfaces that are separate from the implementation.

Component notion: COM is a binary standard – it specifies nothing about how a particular programming language may be bound to it. The one fundamental entity that COM does define is an interface. On the binary level, an interface is represented as a pointer to an interface node. The only specified part of an interface node is another pointer held in the first field of the interface node. This second pointer is defined to point to a table of procedure variables (function pointers). Figure 4.6 shows a COM interface on the “binary” level. Every COM object has to

[1] `http://www.microsoft.com/com/`
[2] `http://java.sun.com/products/ejb/`
[3] `http://www.omg.org/technology/documents/formal/components.htm`

implement at least the so called “IUnknown” interface (which also identifies the COM object). This interface specifies three methods. The first method is QueryInterface, which takes the name of an interface, checks if the current COM object supports it, and, if so, returns the corresponding interface reference. The other two mandatory methods of any COM interface are called AddRef and Release. Together with some rules about when to call them, they serve to control an objects lifetime.

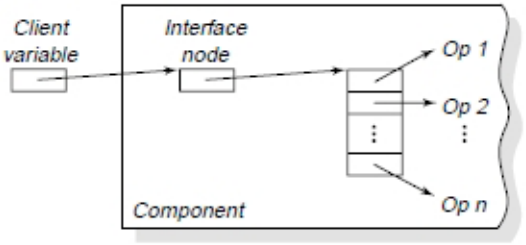


Fig. 4.6: Binary representation of a COM interface. [31]

COM offeres the following main functionality:

- **Life cycle management:** Creation of objects, reference count, garbage collection.
- **Platform independency:** Marshaling: Available on many other platforms (it has however never gained much support beyond the Microsoft Windows platforms).
- **Language independency:** It is a binary standard.
- **Reuse:** Supports composition (containment/aggregation).
- **Location transparency:** There is a central registry for COM objects. Remote procedure calls are supported.
- **Stable versioning:** An COM objects existing interface may not be changed, however new interfaces can be added.

There are also some **drawbacks** regarding COM [27]:

- Do not support design, architectural or process issues, not based on any formal model.
- Weak component notion: COM components can simply be everything with an interface.
- Interoperability and reusability of COM components is restricted to Microsoft platforms.
- Technical issues: Fairly complex implementation. There can be only one version of a certain component installed.

EJB

History: In 1998, Sun Microsystems released a specification for a new component framework, called Enterprise JavaBeans (EJB). EJB is an essential part of the Java Enterprise Edition (Java-EE). Originally defined by Sun Microsystems in corporation with other companies, today the specification is maintained by an open organization, the Java Community Process (JCP) as JSR 220 (EJB 3.0).

Basic idea: Enterprise JavaBeans (EJB) technology is an server-side component architecture, enabling the development of distributed, transactional and secure multi-tier applications, based largely on modular components running on an application server. As the main part of this application server, a so called EJB-Container, provides the runtime environment for business components. The EJB-Container offers technical, standard system services like persistency, allowing the developer to concentrate on business logic.

Component notion: In EJB, components are called enterprise beans. An enterprise bean comprises a bean class (plain old java object) with business logic, a home interface for the bean management and a remote interface to provide clients with access to the business logic. During the deployment process additional technical classes are added to enable the contracts between beans and their runtime environment (the EJB-container). Finally, the bean is packaged into an archive and deployed in a container.

EJB provides the following **main functionality:**

- **Life cycle management:** Creation of objects and Garbage collection.
- **Defines component types and component structure:** For example: Entity bean, session bean, message driven bean.
- **Platform independency:** Runs on the Java Virtual Machine (is an open standard).
- **Technical aspects:** Container managed persistence, transactions and security.
- **Reuse:** Declarative contracts between different components.
- **Location transparency:** Supports RMI.
- Defines **roles** in the development process and **supports deployment** by using deployment descriptors.

EJB contains the following **drawbacks:**

- Hardly support of design, architectural or process issues, not based on any formal model.
- Complex and heavy weight.
- Component notion is too restrictive, tends to represent an object more than a component.

EJB and COM are successful in industry, but both suffer non sufficient component notions. A component identified in system-design is larger than a class (EJB) or a remote interface (COM). The mapping of design-time components to technical component frameworks is still an issue of industrial research and development [27].

4.4 Conclusion

Although the composition of software systems from prefabricated business or technical components is not the solution to the software crisis so far, it offers a very attractive perspective, with the promise to improve software quality and to decrease development costs at the same time. Today, after 40 years of research, the development and usage of software components are well understood and applied in many projects. However, there is still a number of problems and open issues to be taken into account:

- There is still no consensus about the question, what exactly a component is. Szyperskis definition [31], discussed in this paper is a good starting point, describing the key features of components. There is however no explicit mapping of this notion to existing, technology and programming language specific component concepts. What is a component in Java, C or Python? It seems that every software architect has its own answer to this question. A common, technology related component notion is still missing. As components can appear on different levels of abstraction, probably more than only one notion is needed.

- The principles and requirements of a process model suited to componentware, used in the pattern based approach presented in this paper, are still under research. Even though there are some process models concentrating on component-based development like Catalysis or UML Components, they are rarely used in practice. Usually process models incorporate a description technique, tool support, and an informal mapping to existing component frameworks. However, since the advent of UML as a standard description technique, almost all process models incorporate UML as a means of design and architecture modeling. However, UML is rather object-oriented than component-based. This leads to ambiguities between the process component notion and the mapping to UML constructs.

- Today, there is a good technical understanding of how to build and composite components: Some approaches like dependency injection, composition patterns or component categories were presented in this paper. Components however, are not first class citizens in programming languages used today, which again leads to confusion about the question what a component actually is. Furthermore, it is still difficult to describe the semantics of an interface in a light weight and meaningful way. Existing description languages or concepts are largely ignored in practice.

- Component infrastructures like COM or EJB are the success stories of componentware. Their component notion and concepts however, are mostly incompatible. Furthermore, an integration into existing process models or design tools is needed. This is difficult to achieve, as this infrastructures are very unstable, evolving and changing very fast. Furthermore, they are not based on any formal model.

Summarized it can be said, that many areas of componentware are well understood and used in practice today. However, a consistent overall concept is still missing. There is no software-engineering algorithm or methodology that produces adequate components or architectures automatically and it is not possible to create proper components and a good architecture only by

using some tools and a more or less adaptable development process. Human intuition and experience as well as scientific-methodology are still required.

What about McIlroy's vision of the availability of mass produced components, mentioned in the introduction of this paper? It has only partially become true. Lots of reusable components enable us to develop software at a higher level of abstraction. From small scale programming language libraries such as java.util, to large scale standardized frameworks such as javax.swing. Furthermore, during the last years, a number of market places for commercially available components, like www.componentsource.com showed up. However, the use of ready-to-use components is not as widespread as it might be. This is mainly due to the lack of reliable standards and proprietary documentation, component structure, interface definitions, and behavior description. They are highly dependent on their providers.

Besides, only technical components such as GUI-controls or logging facilities are sold today, no substantial reuse of business components (e.g. some banking components) takes place. The idea of just assembling pre-defined components is currently not carried out and might never be [27].

5 Module Concepts in Programming Languages

Grigory Markin
markin@in.tum.de

Abstract. This section discusses various module concepts in programming languages. It outlines several design approaches of software systems and their impact on the modularization techniques. It also presents realization possibilities in modern programming languages using an example application.

5.1 Introduction

In 1972 D. Parnas presented the concept of *information hiding*, an approach of decomposing a system into modules [37] in order to improve the flexibility and comprehensibility of software systems. The main idea is to decouple the system into different parts so that they can be developed and changed independently. Before this work modularity was discussed by Richard Gauthier and Stephen Pont in their book titled Designing Systems Programs, but the criteria of decomposition were not considered. In the article "On the criteria to be used in decomposing systems into modules" by D. Parnas [37] such criteria were introduced. Later this concept was called *separation of concerns*.

Separation of concerns became the most influential concept in software engineering for decades and has had a huge influence on the development of programming languages and design concepts.

This section is organized as follows. In section 5.2, we introduce main notions and an example which we use throughout this chapter in order to demonstrate particular features of design concepts and their impact on the modularization techniques. Then, in section 5.3, we discuss object-oriented design and built-in modularization mechanism in programming languages Java and Python as well as an external framework OSGi. An example application is provided in order to demonstrate this approach. Further we present collaboration-based design in section 5.4 as well as an implementation of the example using mixin-layers in C++. In section 5.5, we introduce aspect-oriented design and use the AspectJ framework for our implementation. Finally, in section 5.6, multi-dimensional separation of concerns is considered as an evolutional step of separation of concerns.

5.2 Preliminaries

We introduce the notion of modularity and present an example application which serves as a demonstration of design concepts throughout this section.

5.2.1 Modularization

D. Parnas suggested *modularizations* as several partial system descriptions from different points of view. The *modularizations* include the design decisions which must be made *before* the work on independent modules can begin [37]. We can treat a *modularization* as a *design* approach that subdivides a system into smaller parts that can be examined independently. More precisely it can be defined in terms of *separation of concerns*, in its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose [38]. A *concern* can describe any point of interest or focus of software engineering. Examples of concerns are persistence, displaying feature, concurrency control, logging feature or configuration. Using this approach it is possible to achieve reduction of complexity, better comprehensibility, extensibility of software system, limitation of the impact of changes and easier component integration. Concerns can be enclosed in terms of a *module*. The structure of the module depends on the kinds of concerns which it encapsulates.

5.2.2 Example application

We will consider a graph traversal application, which was examined initially in [39] and later in [40], [41]. This application implements three different operations on an undirected graph using a depth-first traversal. *VertexNumbering* numbers all nodes in the graph in depth-first order, *CycleChecking* examines whether the graph has a cycle and *ConnectedRegions* classifies nodes into connected regions. The application consists of three distinct parts: *Graph*, *Vertex* and *Workspace*. The *Graph* and Vertex parts represent the data structure. The *Workspace* part encapsulates application routines which are specific to each operation. Initially this application was designed using collaboration-based design approach. We use an implementation from [41] in the corresponding section of this chapter. We also implement the example application using mixin-layers [42]. In each following section a specific implementation of this application using the corresponding design approach will be given. In some sections the application will be extended in order to demonstrate the design approach.

5.3 Object-Oriented Design

In this section a possible object-oriented architecture of the example application will be introduced. A specification of object-oriented programming is omitted due to the prevalence of this approach in the software development. Some specific properties, however, will be discussed.

5.3.1 Example application

As mentioned above the graph traversal application consists of three parts: *Graph*, Vertex, *Workspace*, in [41] each of them was implemented in a separate class. In a collaboration-based design context there are benefits to such a solution, however, in order to achieve sufficient simplicity of the architecture the functionality of the *Workspace* part was omitted. Fig. 5.1 shows the UML class diagram describing the architecture of the example application. Such an architecture can be easily implemented using some object-oriented language.

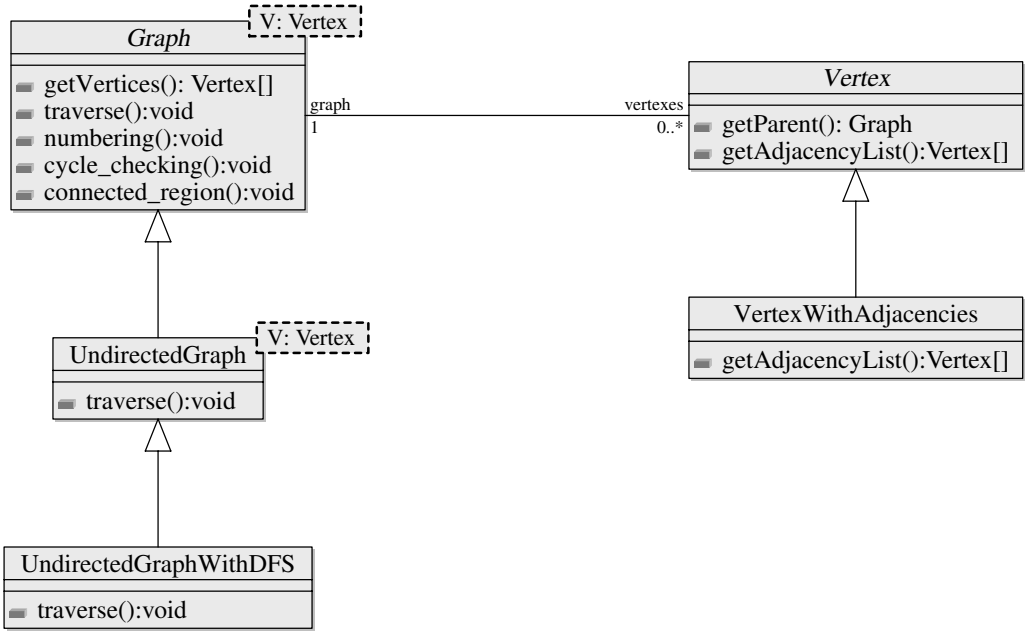


Fig. 5.1: Architecture of the example application using object-oriented design

In our solution the *Graph* class is an abstract class defining base methods such as *traverse, numbering, cycle_checking* and *connected_region*. The methods *numbering, cycle_checking, connected_region* might be implemented directly in the *Graph* class, because of the dependence on only one method, *traverse*. The latter, in contrast to that, has to be refined in each underlying class, because its implementation depends on the type of the graph and the algorithm used to traverse the graph. The same is valid for *Vertex* and *VertexWithAdjacencies* classes. The latter is a refinement of the former and implements the concrete structure of vertices. The graph can be viewed as a container of its vertices and the concrete class can be specified using generics or templates. Other methods of the *Graph* class are those which belong to the *Workspace* part of the application and represent application routines used by methods implementing graph operations.

5.3.2 Modularity in object-oriented design

In the previous section we presented an architecture of the graph application. The solution does not claim to be exhaustive and actually in a real graph library the class hierarchy would be

more precisely refined, for example, using appropriate patterns, but, in our case, it is enough to demonstrate the main aspects of the modularization.

Following the main principles of separation of concerns we identify Graph and Vertex data types as concerns along the data structure layer. Also the refinement of these types represented by inherited classes can be viewed as concerns. Obviously, in order to have an easily manageable part of the application which contains all routines belonging to it, all identified data types should be included in a module. Looking at the architecture, one can examine two ways of the refinement. The first one is the evolution of the data structure, for example, *UndirectedGraph* class concretizes the type of graph, and the second one is the refinement of graph operations, *UndirectedGraphWithDFS* class defines a concrete algorithm of the traversing. Such a class hierarchy can lead to some problems, such as the necessity to define a data type if the traversing algorithm has to be changed, for example we could need to define a new type like *UndirectedGraphWithBFS* implementing a breadth-first search algorithm, although the data structure itself would not be changed. It could be avoided if we would additionally identify graph operations as concerns expressing functions. In our solution we could then extract all methods implementing graph operations from already existing classes and define each of them in a separate class preserving the hierarchy. In our case, it means that all operations based on the traversing would be viewed as a refinement of the latter. Thus the graph operations could be directly applied to the appropriate graph realization, but these operations would need full access to the internals of the graph classes, what does not preserve the *information hiding*, one of the most important principles of the object-oriented design. Another problem is that implementation of operations or features can depend on methods included in multiple classes of the data structure, so the entities which does not belongs directly to the data structure can affect the latter.

Summarizing our observations we conclude that the data decomposition using object-oriented design greatly facilitates evolution of data structure details, because they are encapsulated within single or closely related classes, but it impedes addition or evolution of features.

5.3.3 Modularization in Java, Python and OSGi

Referring to specifications of Java and Python programming language we give main properties with respect to modularization. In Java imperative instructions are modularized into methods, which are then modularized into classes. Classes can be further combined into packages. In Python all constructs, including classes, variables, functions etc., are encapsulated in modules. Each module in Python defines a namespace. The role of a namespace is a mapping between names and objects. Java packages identify objects, they can be treated as a naming mechanism. In Python a file, which includes a definition of a module, is considered as a modularization unit. Java archive files, which include Java class files, are modularization units in Java. At runtime boundaries provided by JAR files fade away, so at runtime there is no way to find out, which module a class belongs to. In Python modules are objects which available at runtime and are used to look up objects within the scope defined by the namespace of the module.

OSGi specification

We give main features of the OSGI specification from the book by C.Walls [43]. OSGi specification defines a deployment model for Java-based modules. In OSGi a *bundle* is a unit of deployment. OSGi bundles are common Java archives, which additionally contains OSGi-specific metadata, including a definitive name, version, dependencies, and other deployment details. The main features of OSGi with respect to modularity are *content hiding*, each bundle is loaded into its own class space; *service registry*, which enables modules to publish services and to depend on services published by other bundles; *parallel bundle versions*, it is possible for several versions of a bundle to reside in the OSGi framework at the same time; *dynamic modularity*, a bundle may be installed, stopped, started, updated, or uninstalled independently; *strong naming*, unique identification of OSGi bundles by a name and version number.

5.4 Collaboration-Based-Design

In this section we present another design approach, collaboration-based design, which allow a different way than object-oriented design to organize the cooperation between data structure types and operations on them. We would like to start with main ideas of this approach and a realization technique. Then we present an implementation of the example application and finally we discuss modularization techniques. All implementations in this section will be given using C++.

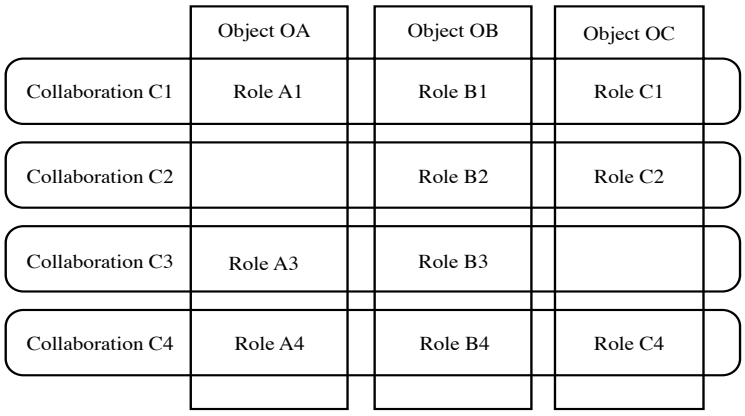


Fig. 5.2: A schematic illustration of collaboration-based design

5.4.1 Preliminaries

In object-oriented design concerns are viewed as data types, which encapsulates entities' details. As we already mentioned, data types are rarely self-sufficient due to the necessity to cooperate with other data types to complete a task. A possible way to represent interdependencies between data types are *collaborations*. Referring to the work by Y. Smaragdakis and D. Batory [41] a *collaboration* is a set of objects and a protocol (set of allowed behaviors) that determines how these objects interact. The part of an object enforcing the protocol, that a collaboration prescribes, is the object's *role* in the collaboration. An object may participate in several collaborations, thus may include several different roles. In that way the main goal of the *collaboration-based design* is to identify a set of collaborations of an application and a set of objects which are, in fact, collections of *roles* describing operations on data types. *Collaborations* are also collections of roles, describing interdependencies between objects. Fig. 5.2 shows the schematic illustration of this idea.

5.4.2 Mixin Layers

The implementation approach, which we discuss in this section, was suggested by Y. Smaragdakis and D. Batory in [42] and is an extension of the technique called *mixin classes* or *mixins*. The definition of this term can be also found in [42]. In object-oriented languages, a superclass can be defined without specifying its subclasses. But this relation is not symmetric, namely, a subclass can not be defined without specifying a superclass. The main idea of mixins is to represent a mechanism for specifying classes that will eventually inherit from a superclass. This superclass, however, is not specified at the site of the *mixin*'s definition [42]. This makes the relationship between superclass and subclass symmetric. In that way a *mixin* can be instantiated with distinct superclasses. This allows to extend a subclass with the additional behavior through specifying a superclass. Using C++ syntax a *mixin* can written as:

```
template <class Super>
class Mixin : public Super { ... /* mixin body */ };
```

This approach is quit simple and has its own advantages and disadvantages, but for the implementation of the collaboration-based design it is not enough efficient what was discussed in [42]. We does not go into details of this technique and go over to the *mixin layers*.

In the article [42] a solution of inefficiency by using pure *mixins* was suggested through implementing collaborations as *mixins that encapsulate other mixins*. They were called encapsulated mixin classes *inner mixins* and the mixin, that encapsulate them, the *outer mixin*. An *outer mixin* was called a *mixin layer* when the parameter (superclass) of the *outer mixin* encapsulates all parameters (superclasses) of *inner mixins*. It is illustrated schematically in fig. 5.3.

Using C++ parameterized inheritance and nested classes, *mixin layer* implementing a collaboration can be expressed as:

```
template <class CollabSuper>
class CollabThis : public CollabSuper{
  public:
    class FirstRole : public CollabSuper::FirstRole { ...};
```

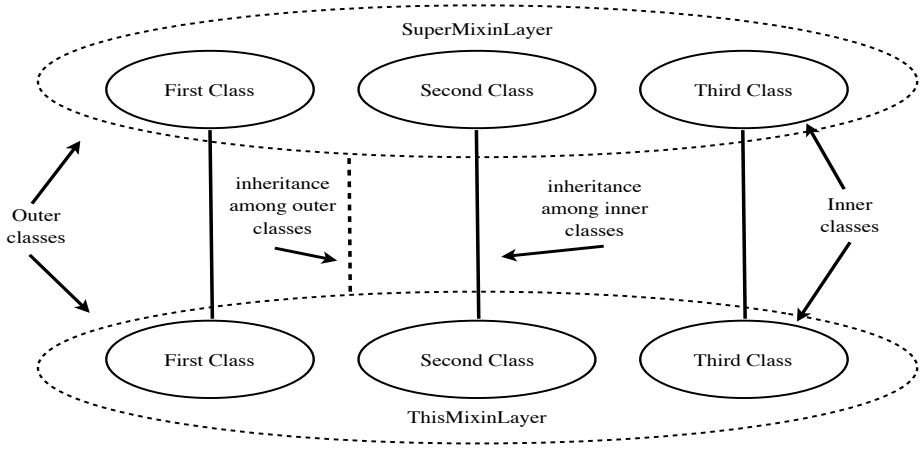


Fig. 5.3: Mixin layers schematically

```
    class SecondRole : public CollabSuper::SecondRole { ...};
    class ThirdRole : public CollabSuper::ThirdRole { ...};

    ... // more roles
};
```

In order to form concrete classes we have to compose *mixin layers*. We have, as presented in fig. 5.2, four different *mixin layers* implementing four collaborations, which we can express using C++ as:

```
typedef Collab4 <Collab3<Collab2<Collab1>>> Example;
```

Note that not all roles have to be defined in each layer. If any roles are not specified for any classes in a collaboration, such roles will be inherited from superclasses.

5.4.3 Example application

Before starting with the implementation of the example application we need to decompose the application into collaborations. Fig. 5.4 shows the schematic representation of the defined collaborations and the roles of the objects. The decomposition is really straightforward. One collaboration expresses the properties of the graph and another four collaborations encodes the graph operations. The order of the collaborations in some cases is important as each layer (collaboration) can be viewed as the refinement of the overlying layer or depends on it. Thus the *depth-first-traversal* depends on the structure of the graph, while other graph operations are considered as refinement of traversal algorithm. These layers can be also reordered among each

other, but they have to underlie the layer implementing the traversal algorithm. The intersection of a class and a collaboration represents the role of that class by the collaboration. The role is a part of an object that is relevant for a collaboration. For instance the role of the *Graph* object in the *UndirectedGraph* collaboration supports storing and retrieving of vertices and the role of this object in the *Depth-First-Traversal* collaboration implements an actual traversal algorithm.

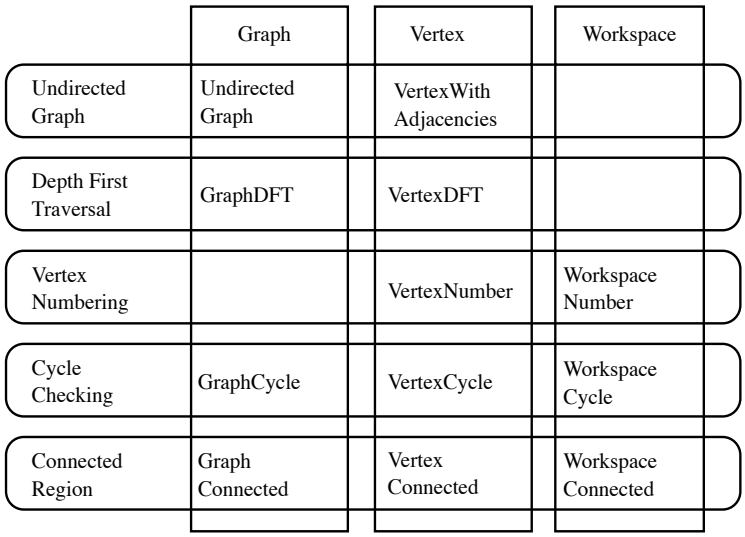


Fig. 5.4: Decomposition of the example application into collaborations

Implementation of the example application

Initially the implementation of this example was considered in the [41]. We use here this implementation with some simplifications. The name of the used approach and its more detailed definition was given in [42]. The terminology in our implementation is taken from the latter.

As in the initially implementation each collaboration is represented as a collaboration component. *Depth-First-Traversal* for example prescribes roles for object of two different classes: *Graph* and *Vertex*. Its implementation using C++:

```
template <class OverlyingCollab>
class DFT : public OverlyingCollab{
  public:
    class Graph : public OverlyingCollab::Graph {
```

```
        protected:
          virtual void OnVertexFounded(Vertex* v);
    };

    class Vertex : public OverlyingCollab::Vertex {...};
};
```

As you can see the roles of the *Graph* and *Vertex* classes in the collaboration are defined as nested classes and inherited from the corresponding nested classes of the superclass implementing the overlying collaboration. Note that the *Workspace* class is not included because objects of the *Workspace* class do not participate on this collaboration, and thus, has no role in it. The method *OnVertexFounded* is treated as a connection point for underlying layers and acts as an event. It triggers whenever a vertex is found. This method is virtual and has to be implemented in underlying layers, if needed. It is illustrated in the following listing.

```
template <class OverlyingCollab>
class VertexNumbering : public OverlyingCollab{
  public:
    class Graph: public OverlyingCollab::Workspace{

        protected:
           void virtual OnVertexFounded(Vertex* v){
                workspace->registerVertex(v);
        }
    };

    class Workspace : public OverlyingCollab::Workspace {

        public:
          void registerVertex(Vertex* v){
                v->setNumber();
          }
    };

    class Vertex : public OverlyingCollab::Vertex {
        public void setNumber(int i){...};
    };
};
```

In such a way all members of the collaboration are combined in a class representing the latter. Before we can compose our *mixin layers* in order to form concrete classes, we have to define a *mixin layer* implementing a collaboration, which does not depends on or refines another one, so that the *mixin layer* class could be a top class in the class hierarchy. In our example it can be the *mixin layer* implementing the *UndirectedGraph* collaboration.

```
class UndirectedGraph{
```

```
  public:
    class Graph
    {
    ...
    };

    class Vertex
    {
    ...
    };

    class Workspace;
};
```

Note, that the *UndirectedGraph* class, implementing corresponding collaboration, has no superclass and it includes definitions of *all* roles, although not all of them are presented in this collaboration. Such roles can be defined as empty classes. It's done, in order to allow inheritance of nested classes in the underlying *mixin layers*. Composing the *mixin layers* into a new type, the concrete classes can be used in the following way:

```
// mixin layers composition
typedef VertexNumbering <DFT<UndirectedGraph>> Example;

Example::Workspace* workspace = new Example::Workspace();
Example::Graph* graph = new Example::Graph(workspace);

graph->numberVertices();
```

Using new type declaration in C++, mixin layers can be modularized in different combinations. Modules can be easily extended by adding new collaborations. The latter can be removed or replaced, forming new configurations of modules.

5.4.4 Modularity in collaboration-based design

In the previous section we showed, how interdependencies between classes can be organized using collaboration-based design. This design approach decomposes an object-oriented application into a set of classes and a set of collaborations. Thus, it can be seen as an approach for organizing entities of an object-oriented architecture. On the one hand, each application class is a set of roles, each of them expresses a separate aspect of the class's behavior. On the other hand, a collaboration is also a set of roles describing a distinct aspect of the application. In section 5.3 we defined concerns along the data structure layer and along the functional layer. Following the same principle, we can define further concerns from the collaboration perspective.

The implementation approach using *mixin layers* allow us to express such concerns as classes, which can be then encapsulated in a module in a common way. Using collaboration-based de-

sign, parts of objects, which belongs to an aspect of an application, can be efficiently encapsulated including their interdependencies within a collaboration.

5.5 Aspect-Oriented Design

In this section we introduce an approach of modularization of *crosscutting concerns*, which can not be modularized using modularity mechanisms of object-oriented languages. We will also extend our example to a additional part, which we will implement using the AspectJ framework.

5.5.1 Motivation

Assume, we want to extend our example application to a logging part. This part have to be able to print a log message into the output before a graph operation starts to execute and after the execution. Using object-oriented design, it can be an quite difficult task, as we have to put some logging routines at the beginning and at the end of each method, implementing corresponding operation. Also using collaboration-based design, we have to define a collaboration, which encapsulates logging routines, define this collaboration as the top *mixin layer* and invoke methods of the logging layer in each method of the underlying *mixin layers*.

Adding such features into existed application is too complicated and in some cases impossible, but taking it into account during the designing of the architecture, is not the best solution due to the necessity of mixing different kinds of concerns, and thus, the impossibility of the modularization of them separately. Such concerns called *crosscutting* concerns. We discuss such kind of concerns in the following section and introduce an implementation framework.

5.5.2 Preliminaries

Fist we introduce the main idea of the aspect-oriented programming and then we discuss the AspectJ framework which we use for the implementation of the logging part of the example application.

Aspect-oriented programming

Aspect-oriented programming (AOP) [44] has been proposed as a technique for improving *separation of concerns* in software. AOP builds on previous technologies, including procedural programming and object-oriented programming [45]. The central idea of *aspect-oriented programming* is to allow the modularization of concerns of interest which can not be modularized using object-oriented languages. These concerns inherently *crosscuts* the natural modularity of the rest of the implementation. *Aspect-oriented programming* enables to program such concerns in a modular way. Well modularized *crosscutting concern* was called an *aspect* [45]. In the previous section was shown how an aspect (logging feature) crosscuts the class hierarchy of the example application.

AspectJ

An overview of the AspectJ framework was given in [45] [1]. We introduce main definitions and principles of programming *aspects* from this paper.

AspectJ is an extension to the Java programming language. It is a general-purpose language like Java, it has a more Java-like balance between declarative and imperative constructs. AspectJ is statically typed, and uses Java's static type system. In AspectJ programs classes are used for traditional class-like modularity structure and aspects are then used for concerns that crosscut the class structure.

There are two kinds of crosscutting implementation: *dynamic crosscutting* and *static crosscutting*. The first one allows to define additional implementation to run at certain well defined points in the program execution. The second one makes it possible to define new operations on existing types. In our implementation of the example application we will use the first one, as it best satisfies our requirements.

Dynamic crosscutting based on small set of specific constructs. *Join points* are well-defined points in the program execution; *pointcuts* are collections of *join points* and certain values at those points; *advices* are method-like constructs defining an additional behavior at *join points*; and finally *aspects* are units of modular crosscutting implementation, composed of *pointcuts*, *advices* and ordinary Java member declarations.

5.5.3 Example application

This section presents an implementation of the aspect expressing the logging part of the example application.

Firstly we have to choose the kind of join points, we are interested in. AspectJ provides several kinds, which was described in details in the article overviewing this framework [45]. We use *method execution* join points in our implementation as we are interested in logging each method invocation. Then we have to specify a pointcut, which is a set of join points. It can be expressed using AspectJ as follows:

```
pointcut traverse_op:
        executions(void Graph.traverse())

pointcut other_op:
        executions(void Graph.numbering()) ||
        executions(void Graph.cycle_checking()) ||
        executions(void Graph.connected_region());
```

After the specification of a pointcut, an advice has to be defined. An advice is a method-like mechanism used to declare the code, which has to be executed at each of the join points in a pointcut. AspectJ provides three kinds of advices: *before, after, around*, which specify when the code have to be invoked relative to a join point. As we want our code to be executed before the invocation of an method, the advice can be defined in the following way:

[1] This paper gives an overview corresponding to the AspectJ version 0.8, the currently available version is 1.6.8, but the main principles of the implementation remains applicable.

```
before(): traverse_op() {
        writeToLog("Traversing started");
}

before(): other_op(){
        writeToLog("Other operation started");
}
```

Finally we have to define an aspect, a modular unit of crosscutting implementation. Aspect declarations have a form similar to that of class declarations. We defined our aspect including pointcuts' and advices' declarations as follows:

```
aspect MoveTracking {
        static void writeToLog(String msg) {
                System.out.println(msg);
        }

        pointcut traverse_op:
                executions(void Graph.traverse())

        pointcut other_op:
                executions(void Graph.numbering()) ||
                executions(void Graph.cycle_checking()) ||
                executions(void Graph.connected_region());

        before(): traverse_op() {
                writeToLog("Traversing started");
        }

        before(): other_op(){
                writeToLog("Other operation started");
        }
}
```

This simple example illustrates the use of the main constructs of the AspectJ language. The more detailed description of the constructs including parameterization possibilities and further features of the AspectJ framework can be found in the article giving an overview of the framework [45] or in the official documentation of the project.

5.5.4 Modularity in aspect-oriented design

In this section we have presented further kind of concerns, *crosscutting concerns*, and an approach for modularization of them. Also we showed, by means of the example application, that without explicit modularization of such concerns, it leads to more complex code, which is much harder to maintain and develop. In some cases it is even impossible to add features, expressed by *crosscutting concerns*, into existing applications. The main idea of this approach, that, in

contrast to design techniques, which we have discussed in previous sections, its implementation mechanisms are not directly embedded into any programming language. Using extensions to programming languages, concerns along different abstraction levels can be modularized simultaneously. In most cases the *clean* modularization of concerns using some programming language, which supports one programming paradigm, can be only achieved along one abstraction level. In particular, the design-approaches, which we have discussed in previous sections, enable us to modularize concerns along only *one* abstraction level: from the perspective of collaborations or data types.

5.6 Multi-Dimensional Separation of Concerns

In this section we discuss a *multi-dimensional separation of concerns*, an another point of view on the classical separation of concern, with an implementation framework Hyper/J.

5.6.1 Motivation

In previous sections we have introduced several techniques of the decomposition of a software system into manageable parts. Each of them considers the system from different points of view and, thus, different kinds of concerns are relevant by the decomposition of a system. For example in object-oriented programming the prevalent kind of concerns is class; each concern along this abstraction level is a data type defined and encapsulated by a class. Features and aspects are also concerns, as well as concerns, which we does not discuss here, for example, configuration, viewpoint etc. Corresponding to [38] a kind of concern, like class or aspect, is a *dimension of concern*.

In section 5.5 it was shown that also several kinds of concerns are of the interest simultaneously, in particular, aspects and classes. Moreover, they may overlap and interact, as, for example, features and classes do. The set of relevant concerns varies over the time and is context-sensitive - different development activities or stages of the software life cycle can involve different kinds of concerns.

Modern languages and design approaches, however, suffer from a problem, which was termed the "tyranny of the dominant decomposition" [46]: most of them enable the separation and encapsulation of only one kind of concern at a time [38]. Even when some of the design techniques allow the separation along several dimensions, the kinds of concern (dimensions), however, are still limited, for example in aspect-oriented design.

In the article "Multi-dimensional separation of concerns and the hyperspace approach" by H. Ossher and P. Tarr [38] the term *multi-dimensional separation of concerns* was defined as separation of concerns involving: multiple, arbitrary dimensions of concern; separation along these dimensions *simultaneously*; the ability to handle new concerns and new dimensions of concern *dynamically*; overlapping and interacting concerns.

5.6.2 Preliminaries

We consider an approach to multi-dimensional separation of concerns developed by H. Ossher and P. Tarr and presented in [38]. We will consider most important principles, which are needed

to understand the main ideas, but we omit some specific details.

The Hyperspace Approach

H. Ossher and P. Tarr called their approach *hypespaces*. Hyperspaces enables the *identification* of any concerns of importance, *encapsulation* of those concerns, identification and management of *relationships* among those concerns, and *integration* of them [38].

Concern Space of Units

Software consists of *artifacts*, which composed of descriptive material in a suitable language. A *unit* is a syntactic construct of such a language. For example, a unit might be a statement, a declaration a class or any other entity described using a given language. There are *primitive* units and *compound* units, which group units together. For example, method or variable might be treated as primitive units and class or package might be treated as compound units. A *concern space* bounds units in some body of software, such as a software system or component library. H. Ossher and P. Tarr divide the process of separation of concerns into three phases: *identification*, involves selecting concerns and populating them with units; *encapsulating* of concerns in order to manipulate them as first-order entities; *integration* of concerns. In the following each of the phases will be considered in details.

Identification of concerns

A *hyperspace* is a concern space specially structured by H. Ossher and P. Tarr to support their approach to multi-dimensional separation of concerns. In a hyperspace all units are organized in a multi-dimensional matrix. Each axis represents a dimension of concern; each point on the axis represents a concern in that dimension and the coordinates of a unit indicates which concerns it affects. In such a way it can be easily specified which dimensions of interest exist; which concerns belongs to that dimensions and which concerns are affected by which units. From this structure also follows that an unit can affect only one concern along a dimension.

Encapsulation of concerns

In order to compose units into modules an additional mechanism is needed. This mechanism was called *hyper-slices*: sets of units that are *declaratively complete*. That means that *everything*, which that hyper-slice refers to, has to be declared within that hyper-slice. The hyper-slice need not to include the full definition. It is necessary in order to eliminate coupling between hyper-slices.

Relationships among concerns and integration of concerns

This phase of the separation of concerns we does not discuss in details here. We only give a definition of the *hyper-module*, which was defined as a set of *hyper-slices* and a set of *integration relationships*, which specifies how the hype-slices relates to one another, and how they should be

integrated. Various types of relationships were considered in the [38] illustrating on an example. We will refer to some of them implementing our example.

5.6.3 An example

For the implementation of the example we will use a tool supporting hyper-slices which was considered in [38]. It supports Java as a language for defining units. A detailed description of this tool can be also found in [38].

Developers create hyper-spaces initially by specifying Java class files that contain the code units which then populate the hyper-space. Hyper/J automatically creates one dimension, the *Class File* dimension, and it creates one concern in that dimension for each class file it loads. These concerns contain units, which are classes, interfaces, methods and member variables, in the corresponding class file.

Firstly we have to encapsulate features of the example application as first-class concerns. It can be done by creating a new dimension, in our case it is *Feature* dimension, and describing how existing units in the hyperspace address concerns in that dimension. In order to do it we have to specify *concern mappings* in the following way:

```
package com.tum.example.GraphLib: Feature.DataStructure
operation traverse: Feature.Traverse
operation numbering: Feature.Numbering
operation cycle_checking: Feature.CycleChecking
```

The first mapping indicates that, by default, all of the units contained within Java package *com.tum.example.GraphLib* address the concern DataStructure in the feature dimension. Next three mappings indicate that any methods named traverse, numbering or cycle_checking addresses the concerns Traverse, Numbering and CycleChecking, respectively. Thus the concern matrix contains two dimensions, Class File and Feature. Next we have to define a hyper-module, which specifies the hyper-slices and relationships among them. It is done as follows:

```
hypermodule GraphLib_with_CycleChecking_and_Traversing
    hyperslices: Feature.DataStructure, Feature.Traverse,
                 Feature.CycleChecking
    relationships: mergeByName
```

In the hyper-module concerns are related by "mergeByName" integration relation. This relation means that units in the different concerns correspond to each other if they have the same names. Corresponding entities have to be combined so that they include all their details. For example, all members in corresponding classes will be included in the composed class.

Finally, Hyper/J can generate Java class files and composed pseudo-source files for any composed hyper-slice using the hyper-module specification. The class files can be then executed in a common way, pseudo-files can be used for debugging.

5.6.4 Modularity

The structure of hyper-spaces allow developers to concentrate only on those part of the software system which is relevant at particular stage of the development process or software life

cycle. Developers only need to examine a hyper-plane containing those dimensions and concerns which are of the interest. There is also a possibility to define new concerns and dimensions on demand. Thus the necessity of identifying of *all* concerns, which might be important at *some* point of the software life cycle, at the initial design phase can be avoided. The practice shows that in most cases all concerns and dimensions can not be identified during the initial designing due to the continual changes of requirements. Hype-spaces specifies explicit how concerns affect one another and how they relates to each other. Hyper-spaces also makes it possible to achieve the limited impact of changes. They offer an efficient way to extend, customize and extract hyper-slices, without or with little changes of another hyper-slices through its declaratively completeness. Adding new units to hype-space is also quite straightforward, forcing the developer to define how new units affect already existed.

There are a lot of advantages of the hyper-space approach. We does not give them all here referring to [38], but we mention the key feature of this approach in comparison to other, it's the support of on-demand re-modularization: the ability to extract hyper-slices to encapsulate concerns that were not separated in the original software artifact [38].

The hyper-space approach is very interesting technique which could allow to solve a problem of the "tyranny of the dominant decomposition" [46] in the software engineering and weak the development process. But in the current state it enables to solve only limited set of tasks. Although developers can benefit from using Hyper/J framework, it is still not applicable in the development of complex software systems. There are a lot of open issues, which was discussed in detail in [38].

5.7 Conclusion

We discussed object-oriented, aspect-oriented and collaboration-based design approaches as well as modularization possibilities using all of them. These design techniques comply with the separation of concerns concept. In addition, we illustrated by means of an example application how these approaches can be applied. We also considered multi-dimensional separation of concerns, which can be viewed as an extension of the separation of concerns concept. The application possibilities of this concept were also demonstrated by means of the example application.

Data decomposition using object-oriented design aids to express the evolution of data structure details, as they are encapsulated within a single class or a number of closely related classes. These classes can be then combined into separate modules expressing different aspects of an application. However, object-oriented design makes the addition and evolution of features quite difficult, restricting the possibilities of its separate modularization.

Data types are rarely self-sufficient because they typically have to cooperate with other data types in order to complete a task. Collaboration-based design approach aids to organize the cooperation between data types. It allows to organize those parts of data types, including their interdependencies, which are necessary for completion of a task, within a collaboration. Collaborations can be then combined into modules. The implementation method which we considered is, however, only applicable using programming languages which allow to express collaborations. This, unfortunately, limits the use of this approach.

Aspect-oriented programming was considered as a design approach improving the separation

of concerns concept. This approach allows to modularize concerns of interest, which can not be modularized using object-oriented programming. These concerns inherently crosscut the natural modularity of the rest of the implementation. There are programming languages, which directly enable the implementation of the aspect-oriented design. For languages without embedded support, there are extensions. One such extension we used for the implementation of the example application.

Often several kinds of concerns, which also may overlap an interact, are of interest simultaneously. While separation of concerns implies the decomposition of the system along one dimension, multi-dimensional separation of concerns, viewed as an extension, involves system decomposition along multiple dimensions simultaneously, handling new concerns dynamically as well as handling overlapping and interacting concerns. The Hyper/J framework, which we used for the implementation of the example application, realizes this concept. However the approach is still in an early stage and can not be applied in the development of complex systems due to a large number of open issues, such as identification of the structure of concerns' interactions within and across software artifacts; ways of specification of encapsulated concerns and their integration; mechanisms of achievement of goals of multi-dimensional separation of concerns and their scalability; development of tools for support of the concept; improvement of software development process in order to take advantage of using this concept.

We showed that some tasks can be solved more efficiently using one design approach than another, while the use of one design approach may promote some tasks while impeding others. Thus, in most cases several tasks can not be solved efficiently at the same time, only using design approaches complying with the concept of separation of concerns. Multi-dimensional separation of concerns might be viewed as an improvement, eliminating such limitations. However, many open issues have to be solved in order to enable wide-spread use of the concept. Similarly, the simultaneous and co-dependent development of programming languages and modularization techniques inhibits progress in the latter, due to the limitations imposed by programming paradigms. Thus we believe that modularization techniques should be treated and developed separately.

6 Service-Oriented Architectures

Tankred Hase
hase@in.tum.de

Abstract. Service-oriented architectures are becoming more and more attractive for large enterprises as historically grown monolithic IT infrastructures become less flexible over time. In this chapter this problem is discussed and some common terms are introduced. Decomposition criteria are given as well as a set of best practices for SOA, which have been gathered from the internet community. Finally a formal approach is introduced through the Service-Oriented Modeling Framework.

6.1 Introduction and Motivation

Service-oriented architectures have become quite popular for large enterprise environments, as they enforce best practices for large-scale software architectures. Because it has not always been that way, the evolution towards SOA is described.

6.1.1 Challenges behind Business-IT Architectures

As small organizations grow into large enterprises, the IT-infrastructures grow with them. This often results in a broad range of hardware and software technologies, that have to interact with one another. When systems are designed, an architecture is created. Yet this architecture usually degrades over time, as new functionality is added to the systems. This can have several reasons:

- The architect who designed the original system, is no longer available. This means that there is no one qualified to refactor the architecture, when changes have to be made.
- The system has been created with a technology or programming language in mind, that no one understands anymore. Therefor no one feels confident in changing or replacing the current system.
- Not enough recourses have been allocated to the project, resulting in a "quick and dirty" approach.

There are many others reasons why system architectures change over time. Usually it does not matter what the reason is. The result is often a complex monolithic software system, which becomes increasingly difficult to maintain. In the following, some of these problems are discussed. [47]

Legacy Systems

Technology changes dramatically over a brief period of time. A few decades ago, it was still quite common to program using assembly language. As time went on the importance of abstraction grew, as people began to understand the importance of using platform independent languages and technologies. This led to the use of technologies like Java and Corba, which were designed to increase portability and thereby ease maintainability

The problem is, that the investments made to the legacy systems cannot be ignored. With the rate of advancement in the software industry, it is simply not possible to rewrite all legacy software each time a new technology emerges. Yet maintanance becomes increasingly difficult as time goes on, since the people who know how to maintain the legacy software may well be retired.

Redundancy

When designing software architectures many engineers decide to redo work that has already been done before them. This can have many reasons, one of them being, that the architect simply did not know of the first system. The result can lead to many problems in maintenance as changes may have to be made to multiple systems, to insure that all of them stay in sync.

Complexity

One of the biggest problems in IT-architectures lies in the complexity of the system. As time goes on and requirements change, functionality is added to a system. From a component-oriented view, this means that the number of dependencies between the components grows constantly. This results in systems that are too complex to understand and maintain.

In the following, different solutions to the mentioned problems are discussed. All of these have the common goal of trying to reduce complexity and ensure that the IT-system does what it is supposed to do. Which is to support the business processes of the organizations that use them.

6.1.2 Enterprise Application Integration (EAI)

One commonly used strategy to reduce dependencies in an IT-infrastructure is called Enterprise Application Integration (EAI). Here a central Business Bus is used with technologies such as the Common Object Request Broker Architecture (CORBA) in order to facilitate a central communications hub, as displayed in figure 6.1.

This has the main advantage of reducing dependencies between software components, but still creates a dependency to the bus technology being used. If the architecture strategy was to change in the future, all interfaces to the bus would have to change as well.

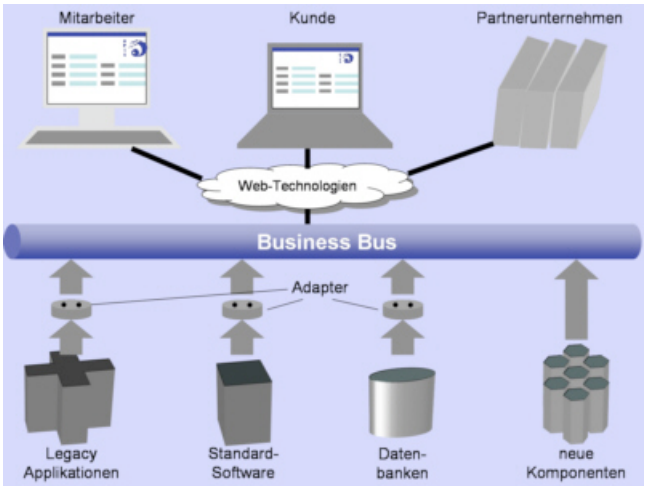


Fig. 6.1: Schema of EAI [48]

6.1.3 Business IT Alignment

Not so much a technology paradigm, but more a philosophy of the way a business IT architecture should be constructed, is called Business IT Alignment. This simply states that when designing any IT architecture, one should first consider business processes that have established themselves in the organization. It should not be required for the business practices to have to adapt to the IT, but more that the IT should adapt to the business practices. This sounds completely logical, but can easily be underestimated when concentrating mainly on the software engineering aspects of a system.

6.2 A Definition of Service-Oriented Architecture (SOA)

One way to explain what a SOA is, is through the use of a metaphor. The design of a single software application is to constructing a building, as designing a SOA is to planing a whole city. This comparison makes sense, because during the construction of the building, a certain set of rules and constraints exist. The same thing goes for writing software for a large company. Also when workers build a house, they don't reinvent things such as plumbing and electricity for every single house, but use the infrastructure that is already available for the current city. This philosophy of integration is not that common in software when one thinks about it, as software isn't always designed with the rest of the IT infrastructure in mind. SOA tries to rectify this complaint. [49]

Finding a commonly accepted definition for SOA is quite difficult, as the understanding of what a SOA is has changed dramatically over the years. In the past, building a SOA meant that the software architecture was decomposed into loosely coupled services using web-technologies. Today, there is much more to it than a simple programming paradigm. One very important aspect to keep in mind when talking about SOA today, is that when an organization decides to build a SOA, it does not implement a certain standard or specification, but it commits itself to a set of principles and good practices. Moreover, a SOA does not require there to be IT-Services at all. [49]

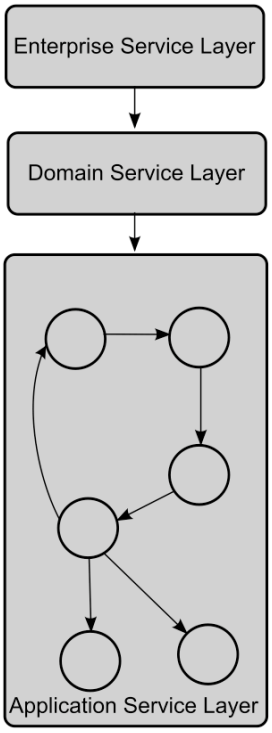


Fig. 6.2: Decomposion through domains [50]

6.2.1 Decomposition Criteria for SOA

A SOA creates an abstraction layer on top of business services, IT services and other technical services. A service can even be represented by a group of people who perform a set of manual tasks, such as filling out an application form or setting a stamp with a signature at the bottom. This is important to keep in mind when designing a SOA, as it prevents the architecture model from being reduced to an abstraction level, which focuses only on technical aspects.

Enterprise Services represent a set of services used throughout an organization. Domain Services are attributed to the domains that exist within an organization, such as billing or production. Finally, the application service layer represents the lowest level of abstraction. These services implement the technical detail required by high level services. This relationship is depicted in Figure 6.2. Decomposition using domains is only one possibility. Like any other good architecture, a SOA should use decomposition through separation of concerns and information hiding. This type of abstraction is shown in figure 6.3.

When designing the technical aspects of a SOA, one of the top priorities is to achieve flexibil-

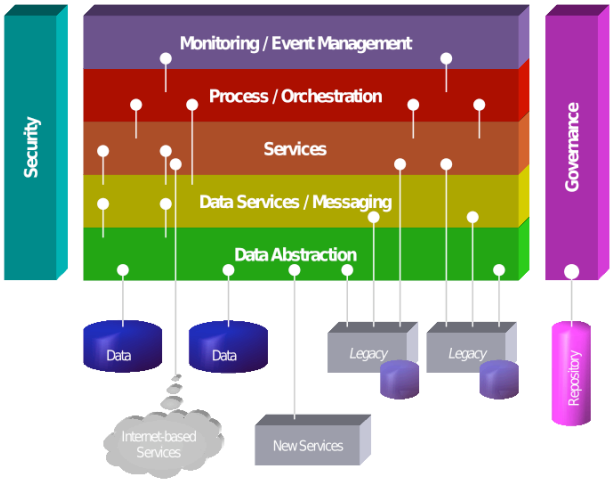


Fig. 6.3: Decomposition through separation of concerns [50]

ity. This is done through a standardized description of platform-independent services, which are loosely coupled from one another. The services should have no calls to each other embedded in them and should serve a single purpose, such as filling out a form or querying a database. This allows the IT-infrastructure to be adapted easily, when business strategies change, as services simply have to be reconnected in the required fashion.

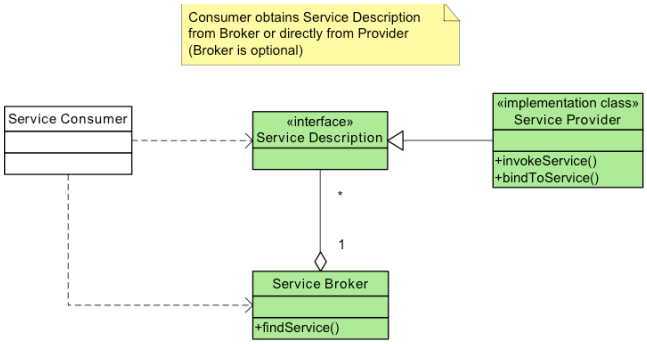


Fig. 6.4: SOA architectural style [51]

An architectural style for SOA is depicted in figure 6.4. This shows how a service can be implemented. There has to be a formal service description which specifies the service functionality in a platform independent manner. This is usually done by using xml. When invoking the service, the service consumer has to then use the signature provided by the description. It is possible to have a service broker or registry, which provides a list of all the available services, but is not required. Finally, the service can then be implemented using any programming language. This means, that when the service implementation changes (for instance by changing the

programming environment), the service consumer does not notice it.

6.2.2 Best Practices for SOA

Defining a set of generic best practices for SOA is not as simple as for the architecture of a single application. This is because the complexity that comes with designing a SOA is far greater. Nonetheless, a set of best practices have been gathered by the internet community and documented in the SOA Manifesto:

1. "Respect the social and power structure of the organization.
2. Recognize that SOA ultimately demands change on many levels.
3. The scope of SOA adoption can vary. Keep efforts manageable and within meaningful boundaries.
4. Products and standards alone will neither give you SOA nor apply the service-oriented paradigm for you.
5. SOA can be realized through a variety of technologies and standards.
6. Establish a uniform set of enterprise standards and policies based on industry, de facto, and community standards.
7. Pursue uniformity on the outside while allowing diversity on the inside.
8. Identify services through collaboration with business and technology stakeholders.
9. Maximize service usage by considering the current and future scope of utilization.
10. Verify that services satisfy business requirements and goals.
11. Evolve services and their organization in response to real use.
12. Separate the different aspects of a system that change at different rates.
13. Reduce implicit dependencies and publish all external dependencies to increase robustness and reduce the impact of change.
14. At every level of abstraction, organize each service around a cohesive and manageable unit of functionality." [1]

[1] http://www.soa-manifesto.org/

These best practices can ultimately be boiled down to the following set of priorities. It is important to keep in mind this simply represents guidelines, which have been erected through experience. They are certainly not set in stone and can be altered depending on the situation.

- "**Business value** over technical strategy
- **Strategic goals** over project-specific benefits
- **Intrinsic interoperability** over custom integration
- **Shared services** over specific-purpose implementations
- **Flexibility** over optimization
- **Evolutionary refinement** over pursuit of initial perfection" [2]

Having summarized what a SOA actually is and having gathered some best practices, some formal methods for designing a SOA will be discussed in the following section. This will introduce a development process as well as a formal modeling language for SOA.

[2] http://www.soa-manifesto.org/

6.3 Service-Oriented Modeling (SOM)

As if trying to establish a standardized definition for SOA is not enough, finding a notation that can be used and understood by all project stakeholders is even harder. Although service-oriented modeling can done done using traditional modeling languages such as UML, there isn't any specific notation for services. The Service-Oriented Modeling Framework (SOMF) introduced by Micheal Bell tries to establish a standard modeling method for SOA. This attempt is formulated in the Service-Oriented Modeling Framework (SOMF). [52]

6.3.1 Service-Oriented Modeling Framework (SOMF)

The Service-Oriented Modeling Framework is a collection of best practices for SOA. Instead of simply naming priorities, like the ones which were introduced in section 6.2.2, SOMF defines a distinct development process for SOA projects. Also a modeling language or notation is introduced, which can be used to document the architecture throughout development. An overview of the these practices is shown in figure 6.5.

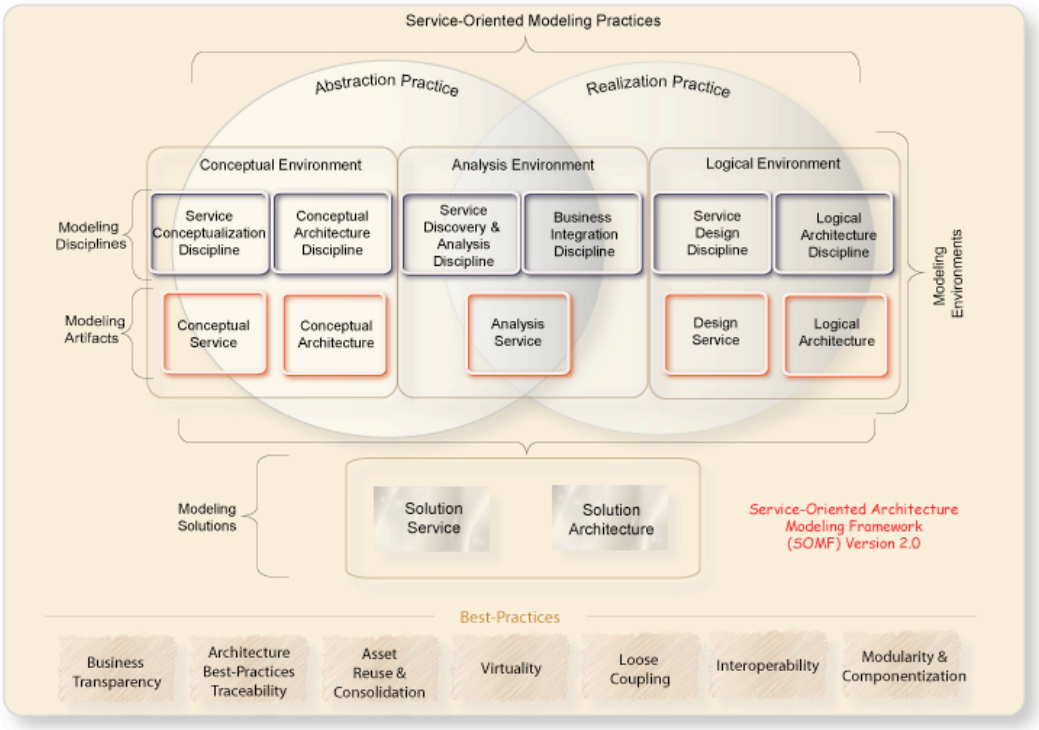


Fig. 6.5: The service-oriented modeling framework [53]

6.3.2 SOM Notation

Just like any modeling language, the SOM introduces a common notation for components (figure 6.6) and relationships (figure 6.7). These can be connected similarly to a class-diagram in uml.

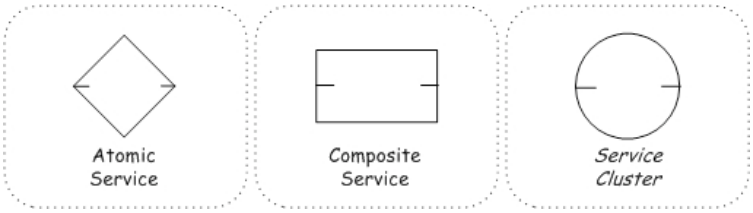


Fig. 6.6: SOM assets [53]

The SOM assets include the following:

- **Atomic Service**: This component acts as an autonomous service which is not subject to decomposition. It represents business or technical requirements, which are needed to perform simple tasks. An example for this could be a service providing the address-book of a company.
- **Composite Service**: This component represents a more complex service which can be broken down to other composite or atomic services. These services provide rich functionality. An example could be a service providing email functionality, which relies on the address-book service.
- **Service Cluster**: This component represents a collection of distributed and related services, which have been gathered because they serve the needs of a certain business problem. An example for this could be a set of services, which are used for a certain domain, such as billing. This service cluster in turn relies on the email service and the address-book service, as well as other services used for financial-specific purposes.

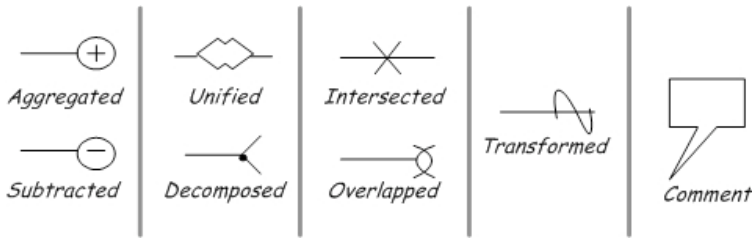


Fig. 6.7: SOM relationships [53]

The relationships can finally be used to create connections between the service components. An example for this is depicted in figure 6.8. This example shows the dependencies between two atomic and two composite services.

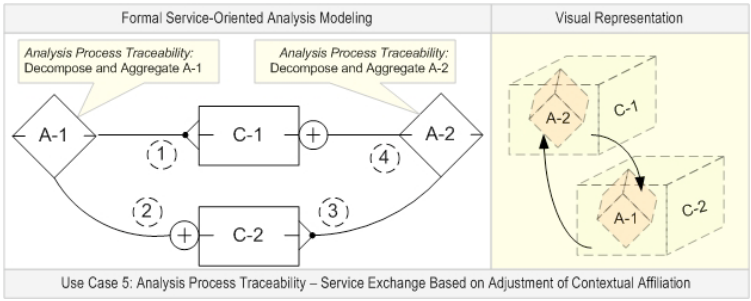


Fig. 6.8: An example of the SOM language [53]

6.3.3 SOM Development Life Cycle

Due to its complexity, the consolidation of large IT-environments requires much experience. This is why the development process defined by the SOMF is very detailed and specific. It gives novice developers a solid foundation on which they can build.

Figure 6.9 displays the iterative nature of this development process. Whats important here is that results created through modeling are continuously integrated and verified. This ensures that the SOA model aligns with the business needs, while abstracting out technical details. Each phase will be explained in the following.

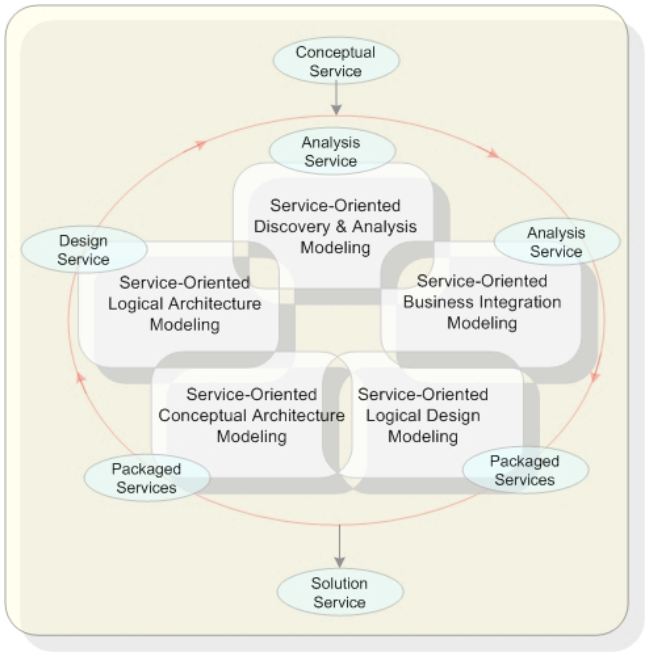


Fig. 6.9: SOM life cycle activities [53]

Service-Oriented Conceptualization

The intent of a SOA is captured during this phase. Conceptual services are defined which tend to the needs of strategic business requirements. A conceptual service does not hold any information about software implementation, as it merely represents an abstract content defined by the business analyst.

Service-Oriented Discovery and Analysis

Concrete services are identified, which have to be integrated into the SOA. Conceptual services defined in the previous phase are mapped to technology solutions. This is where the gross of the modeling takes place, using the above mentioned notation.

Service-Oriented Business Integration

This phase represents an ongoing activity to ensure the close tie between business and technology. Services defined during analysis are continuously integrated throughout development. This shows if the chosen technological direction is really aligned with the business strategy.

Service-Oriented Design

A game of "connect the dots" which results in a model of message and information exchange between services and consumers. Logical relationships, interfaces, and structural aspects of services and their consumers are established.

Service-Oriented Conceptual Architecture

Strategic architecture choices are made, which ensure that the resulting architecture is flexible enough for future needs. The resulting architecture is not tied to any software implementation but rather represents a skeletal system into which software components can be inserted. This lays a foundation for an implementation plan while staying implementation independent.

Service-Oriented Logical Architecture

A platform independent architecture is produced, which addresses non-functional requirements such as reusability and interoperability. This architecture model introduces detailed diagrams of software components, middlewares, and platforms being used. During the design phase of logical architecture loose coupling should be enforced. [52]

These phases represent an iterative approach, similar to the spiral model and enforce continuous integration. Although they represent quite a strict sequence for development, they also incorporate best practices gained through experience.

6.4 Summary

In this chapter the SOA paradigm was introduced. This does not represent a set of technologies such as web-services, but provides a set of best practices for large enterprise architectures. SOA should be used to decouple monolithic IT-infrastructures, which have grown historically without any consistent plan.

Also service-oriented modeling was introduced. This paradigm is an attempt to formalize best practices using a custom modeling language. SOM also offers a development life-cycle, which tries to ensure the agile nature of SOA. This development process borrows from iterative approaches like continuous integration, and the spiral model. Although it represents a solid foundation on which novice architects can build on, it is also very theoretical and perhaps even too strict to represent an agile development process.

Development in the business IT domain has gone a long way. From monolithic architectures which were not flexible enough to EAI architectures, which consolidated components through a central bus technology. Then from EAI to SOA, which enforces the implementation of loosely coupled services and promises flexible enterprise architectures, which are able to embrace future change. But what comes next? Will SOA really stand the test of time or is it just a hype, created by consulting firms in order to sell their products. Only time will tell.

7 Analyzing Software Architectures

Marius Capota
capotam@in.tum.de

Abstract. The software architecture is crucial for the success of a software-intensive system. It represents the foundation every implemented system is based on. For that reason, a solid design of the software architecture has immense impacts for the future of the system. The correct design of the software architecture includes a testing and evaluating phase that has to prove the system will be conform to the required functionality and quality. The analysis process can occur on the architectural level after the software architecture is presented.

7.1 Introduction

We can use an analogy between the development process in software engineering and the construction of a house. First, the architect creates a design of the house. It has to prove that the house will be stable and also it will fulfill the qualitative requirements asked by the client. In the design phase, after the architecture of the house is available, it can be discussed and analyzed. If mistakes or mismatches in the design are discovered, then it can be changed and improved in an early phase, before the actual construction begins.

A similar process we can find in the development of large software projects. Because such projects are very costly and complex, involving lots of developers, their development often follows principles of the software engineering. One fundamental recognition is that the software architecture has to be created at the beginning of the project as one of the first artifacts. The software architecture is the basis the later refinements of the design and the implementation will be depend on. That is why the software architecture has to be robust to the requirements. In an evaluation process it can be tested to what extent the software architecture is adequate. From a premature architecture we will often get after a redesign a more stable one. This architectural process saves a lot of effort and money that will need to be invested lately to remediate original weakness points in the architecture.

In this paper, I present efforts that are made by different academic institutions and developers to assist the software architects in order to analyze the software architecture of their software-intensive systems.

The first hurdle to be cleared, if the software architect wants to estimate the own created architecture is to find the appropriate evaluation method from the large number of available methods. A classification of different techniques shows that the software architect has to be assisted in choosing the “right” method. A solution to the problem is presented in section 7.4.

The central parts of the paper will present a scenario-based analyzing method (ATAM) and an example of a quantitative evaluation of two software architectures. The description is detailed

to get a concrete picture, how such methods work on real projects. These examples constitute an introduction to the important and interesting domain of testing and evaluation of software architectures.

The last section will give a summary and my own thoughts regarding future work.

7.2 Why do we need to evaluate a software architecture?

The developers of many different software architecture methods that skipped the academic stadium and were applied in different real projects affirm that its outcome was always a benefit for the stakeholders and for the development process of the project.

The advantages of a software evaluation can be described by two domains:

- functional and quality aspects
- economic aspects

7.2.1 Functional and quality aspects

Functional and quality aspects motivating the evaluation of a software architecture go hand in hand with the response to the important question whether the presented software architecture fulfills *functional* and *nonfunctional, qualitative* requirements of the already developed system or the system being in the development process.

Functional aspects consider the ability of the software architecture to fulfill what the system has to do from the customer's viewpoint, i. e. what functionality it should provide. The functional view is important for the customer because that is the primary reason why he or she gives into commission the development of a system and the developed system has to meet his or her criteria.

On that high abstraction level the architectural level represents, the software architecture can only presume that the functionality of the system will be properly implemented after the design phase. After the testing phase of the software development process, the implemented system should provide the required functionality. That is why the design of the software architecture of the system has the premise that it intrinsically provides the functionality asked by the customer. Because of that, it is more important to evaluate the consistency of the architecture regarding the qualitative requirements like performance, security, reliability, changeability, maintainability, etc.

One major result of the evaluation with the participation of different stakeholders is that risks in the architecture can be discovered. The main focus on analysis is the conformance of the software architecture regarding quality attributes. Affirmations about the overall quality of the software architecture (and thus of the overall system) can be concluded: how good or bad it contributes to the fulfillment of the qualitative requirements.

Concurrent architectures, architecture models with different design focus can also be evaluated and compared. The final results of their evaluation should present advantages and disadvantages of them. The comparison of the software architecture concepts makes the decision to implement the more suitable architecture more easily.

7.2.2 Economic aspects

The economic aspects concerning a software architecture evaluation (SAE) consider factors that are relevant for the business of the customer.

The analysis of the software architecture in a premature stadium of the development could reveal serious quality problems or the impossibility to accomplish the requirements due to immature technologies or non existent architectural or implementation methods. That may be foreseen in a an early phase of the development process. If that is the case, then the project can be aborted and resources become available for other more productive investments for the customer.

Because of the permanent and rapid development in the Information Technology (IT) many software systems that were developed in the past and have been properly fulfilled their duty for a long time are at present "reliques of old times", difficult to maintain and to extend their functionality. An evaluation of their software architecture with the emphasis on the economic aspects could reveal that the further development based on the old architecture is related with very high costs (see the Cost-Benefit Analysis Method [54] and [55]).

Should the existing software architecture be refined and extended or is it better to start from scratch with a new one and proceed with its implementation? Seeing from the business viewpoint, the decision to overcome the old architecture and replace it with a new one is perhaps more plausible. Such crucial decisions must have a solid basis. The software architecture evaluation techniques specialized on this perspective are available to support the software architect and his team.

7.3 A classification of software architecture evaluation methods

The following figure 7.1 shows the software architecture evaluation in the context of the software architecture development process as described by Starke [21]. The process starts with the requirements on the system. It is also known as *Requirements Engineering*. Based on the elicitation and analysis of the requirements, the architecture team conducted by the software architect develops the software architectures which have to consider and apply to many factors and aspects.

If the software architecture has been constructed, then it can be presented to the stakeholders in the form of diagrams and documents. These outputs are considered artifacts of the software development process. If the decision is taken to analyze the software architecture, then an evaluation meeting or more are managed. The evaluation sessions can be more informal like a review or they can go into details as we will see in section 7.5 of that paper where I present an analyzing method.

In the evaluation meeting questions could arise about the disregard of some requirements on the system that are not clearly supported by the presented software architecture. Another concern could be that some of the requirements were not properly specified and thus correctly implemented. This mismatch should take the design and evaluation process to the beginning and repeat the first step. The cycle should be iterated till a new improved software architecture will evolve that considers all demands and diverse requirements. Design problems regarding the

software architecture are clarified in an early phase (if the architecture has not been implemented yet) after they were discovered in the evaluation step.

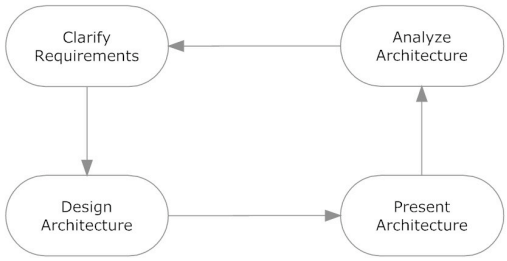


Fig. 7.1: Process of the software architecture development [21]

In the last two decades plenty of methods for analyzing software architectures have been proposed. Some of them are applied in real middle or large projects and prove to be consistent. A classification for the different types of the methods emerges.

- **The quantitative software architecture analyzing methods** are oriented towards the measuring of special characteristics of a class, a component, a prototype or of the completely implemented system. Software artifacts as source code, dependencies between methods or components, or their complexity are mapped on numbers. Unfortunately, such mathematical models are especially for non experts difficult to understand. Expert knowledge is required to create the models and then to compare and interpret the results. It is still discussed on an academic basis what metrics are relevant on the architectural level [56].
- **The qualitative software architecture analyzing methods** try to find out if the architecture satisfies to the nonfunctional requirements of the system. *Questionnaires* and *checklists* are often based on practical knowledge that was accumulated on past projects. The software architect and his team make use of this experience in order to avoid usual mistakes in the design of the architecture. *Scenario-based analyzing methods* are based on possible interactions with the system. These are modeled and presented by use cases. In section 7.5 I give an example from that subclass.

7.4 A methodology to choose the "right" evaluation method

Recently, a multitude of new methods and techniques for analyzing software architectures have been proposed by academic institutes that should support the software architect and the involved stakeholders in evaluating the architecture. If the software architect takes the "wise" decision to evaluate the software architecture of his or her project, he or she must ask the question which evaluation method is the best one for his or her own purposes. That section gives an overview for many evaluation techniques. There are some academic works that address that dilemma for the software architect as presented by Ionita [57], Babar [58] or Eicker [59]. Each of them created his own framework that should help to characterize and compare diverse analyzing methods.

In my opinion, I find the work by Eicker to be more helpful especially for its step-by-step guidance and its understandability for non experts. A guidance based on his work is given in the following.

The report of Eicker [59] examines systematically a variety of evaluation methods. Its results can be seen as an instrument for resolving the dilemma presented in the introduction of that section. Furthermore, it gives software architects or other interested people an overview over eighteen different analyzing methods.

One observation is that the evaluation methods can be classified according to their relationships to each other.

Inheritance

The inheritance relationship points out a specialization between two methods. Most of the elements of the inherited methods are equal, or changed and extended only in little aspects. The inheritance relationship is showed in figure 7.2.

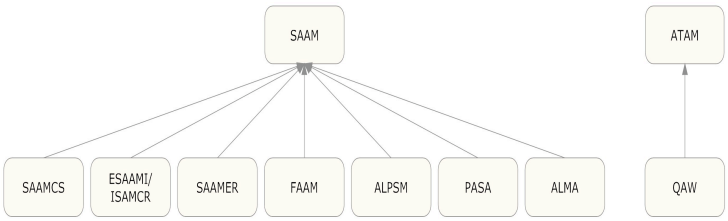


Fig. 7.2: Inheritance relationship between methods [59]

As we can see in the figure, the SAAM and the ATAM are inheriting methods.

Association

The other relationship that was discovered by Eicker is the association. An association relationship between two methods depicts that only part aspects of one method flow into the other one. The new method is composed of parts of the associating method and new models that are characteristic for the associated method.

The following picture 7.3 gives an overview about the association relationship between six methods. The annotation of the link between two methods denotes part aspects that are taken over by the associated method. The new specific concepts and models are annotated sideways to the methods.

Referring to the association relationship between the SAAM and the ATAM depicted by figure 7.3 we can see that the ATAM takes over only the characteristics of the *quality attribute scenario*. The ATAM is enhanced with additional specific features like the generation of an *utility tree* or the discovery of *sensitivity* and *tradeoff points* in the software architecture.

Finally, between the methods SAAM, SAE, AQA, ABAS, SAEM there is no relationship. They are considered to be independent evaluation methods.

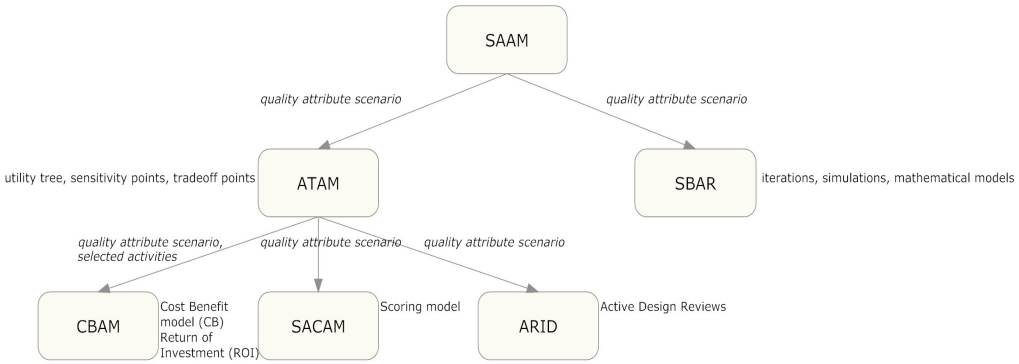


Fig. 7.3: Association relationship between methods [59]

Taxonomy

In order to compare the different evaluation methods a taxonomy was developed which contains the following criteria:

- **Quality attribute**: Does the method evaluate the architecture regarding one quality attribute or does it concern relationships between different quality attributes? Most analyzing methods evaluate a software architecture only for one particular attribute at the same time. In practice, often more attributes than one are correlated and interact concomitantly.
- **Requirements for the application**: Should the implementation of the architecture be available? Is initial training to learn the method necessary? Is the conduction of the evaluation method intense?
- **Maturity level**: Is the method mature? Was it proven in academic and industrial projects? Is there an accompanying example in the documentation of the method?
- **Scope of the method**: Should the architecture be investigated on capability, contained risks, sensitivity, tradeoff points?
- **Project stage**: When can the method be applied? In an early or late phase of the development?
- **Evaluation approach**: Which approaches are used? Scenarios, utility tree, mathematical models, simulations?
- **Participation of the Stakeholders**: Is the attendance of stakeholders in the evaluation necessary?
- **Experience of the evaluation team members**: How much experience of the team is essential for a successful evaluation?

First example

Based on that taxonomy, we can form now a decision tree that will help software architects to find the right method they are looking for in their evaluation. The entrance point into the decision tree represented by the highest node in the picture 7.4 is in that case the criterion *quality attribute* of the presented taxonomy above.

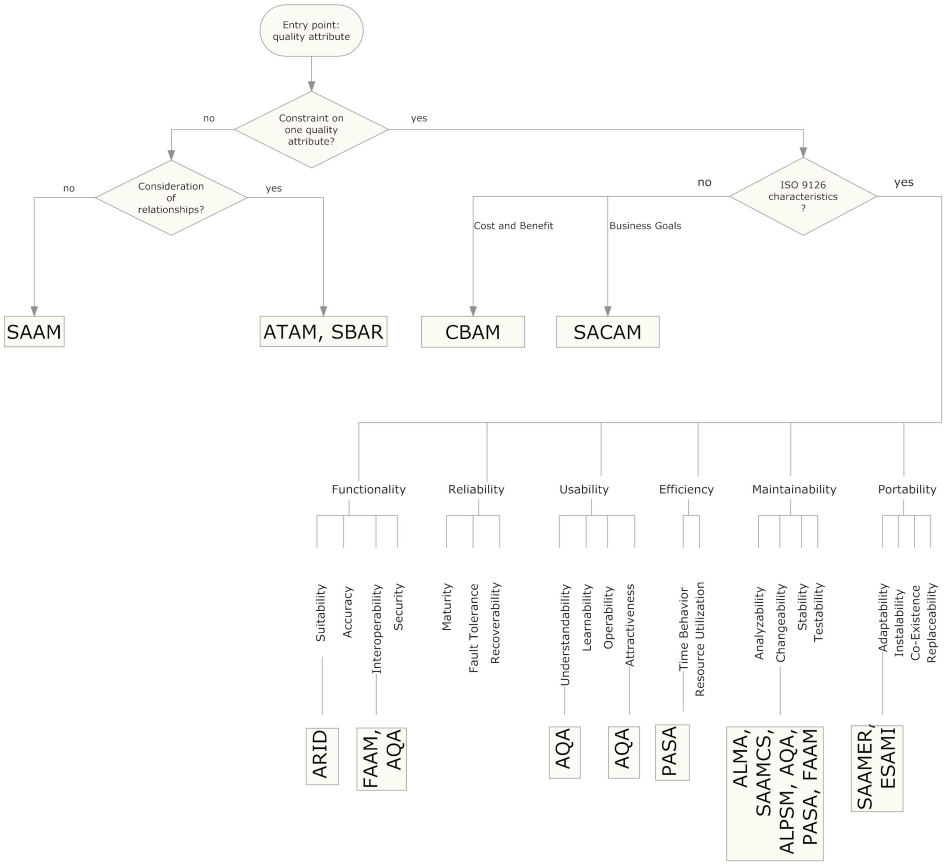


Fig. 7.4: Decision tree based on entry point: *quality attribute* [59]

The first ramification in the tree is depicted by the decision node where we determine if we want to evaluate the software architecture respective to one quality attribute or to more than one at the same time. In the first case, we traverse the tree to the right, in the second case to the left. If we traverse the tree to the right ramification, then we come across the decision node *ISO 9126 characteristics* [22] and we have to answer the question whether we want to deal with a quality attribute that is defined by that ISO norm, or not. The ISO 9126 is a standard quality model that permits the comparison of the quality attributes. In the figure 7.4 we can identify individual refinements of the major quality attributes that are covered and can be evaluated by an existing evaluation method. As we can see, most analyzing methods exist for testing the quality attribute "Changeability". If we are interested in efficiency aspects around our architecture, then

the *Performance Analysis Software Architecture* method (PASA) is the correct choice [60]. If further evaluation methods should be created in future, then they can be added as leaves to that decision tree analogously.

Second example

As a second example how we can help the software architect using the presented taxonomy and methodology is to change the decision tree by choosing the *project phase* as entry point. If the software architect must or wants to analyze the architecture in a late phase of the development, i.e the architecture is mostly implemented, quantitative methods can be applied too. Examples for late evaluation methods are the *Attribute Quality Assessment* (AQA), the *Software Architecture Method* (SAE) or the *Scenario-Based Architecture Reengineering* (SBAR). The following picture 7.5 gives an overview about the decision process.

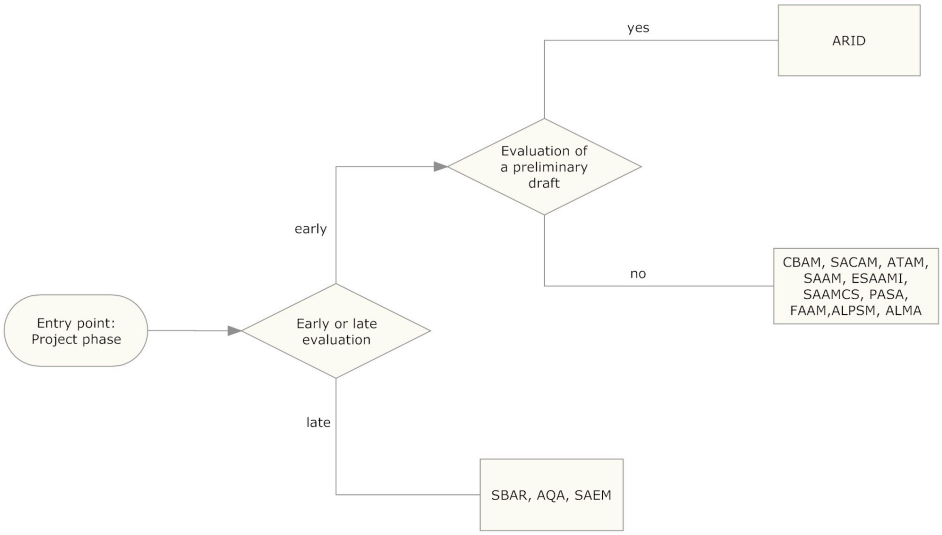


Fig. 7.5: Decision tree based on entry point: *project phase* [59]

Further examples based on the presented set of rules can be exercised. Nevertheless, other criteria can be added in concordance with own's wishes and scope.

7.5 A scenario-based software architecture evaluation method: ATAM

After we have seen the classification of the evaluation methods, I want to present in that section an example from the catalog of scenario based evaluation methods in more detail.

In the following, I present one important scenario based analyzing method, namely the *Architecture Tradeoff Analyzing Method* (ATAM), that is considered de facto standard for evaluating

software architectures. It was tested and improved by its developers in academic and industrial projects. The ATAM is an advancement of the SAAM (Software Architecture Analyzing Method) and was developed at the Software Engineering Institute (SEI) by Kazman, Klein and Clements in year 2000. The ATAM is a mechanism for identifying risks in the software architecture in an early or late phase of the development. It permits a comprehensive evaluation of the software architecture by concurrently analyzing more than one quality attribute at the same time. It supports the reasoning about architecture decisions that led to the presented software architecture.

First, as every evaluation, the use of the ATAM in the software development process is correlated with costs. If the software development process is based on the *Waterfall-model*, than the software architecture has to be created in an early phase. The system and software design (including the object design) and the implementation phase can follow only after that. For that reason, we can evaluate the software architecture quite at the beginning. If a system is implemented following the *Extreme Programming* (XP) principles, then there it may not be a constructed software architecture at all. If we do not have the software architecture of the system but its complete implementation, then we can go backwards in the software development process and do re-engineering. It means that we have to create class and component diagrams from the source code. Additionally, the dynamic view of the system is designed. That undertaking may be very difficult and the conceived software architecture may not correspond to the actual system.

Secondly, the participants on the evaluation must learn and understand the method itself and prepare the requisite documents in order to hold the meeting successfully. Such documents are documentations and presentation slides of the requirements and of the software architecture itself. The benefit is often that the documentation papers are better structured and formulated in an language easier to understand by non experts. The requirements of the system are clearly formulated. The participants can follow what architectural decisions led to the presented software architecture, analyze and test to what extend they affect the fulfillment of the requirements regarding the quality attributes. In many cases, a mature architecture emerges after an intensive evaluation process making visible if it complies to the scenarios the stakeholders have. Finally, the architectural decisions and the software architecture with its advantages and risks are understood by all stakeholders.

As descried by the development team around Kazman, the execution of the evaluation based on ATAM consists of nine steps. For a complete description of the method with two major applications please refer to the standard book "Evaluating Software Architectures" [61] or "Documenting Software Architectures" [55].
The picture 7.6 shows the nine steps which are described in detail in the follwing.

Step 1: Present the ATAM

The evaluation process begins with an introductory phase with the evaluation team presenting an overview about the ATAM, namely: its steps, its techniques as the generation of an utility tree, the elicitation and analysis of the architecture, brainstorming of scenarios and the outputs of the method. Emphasis of this step lies also in the team building.

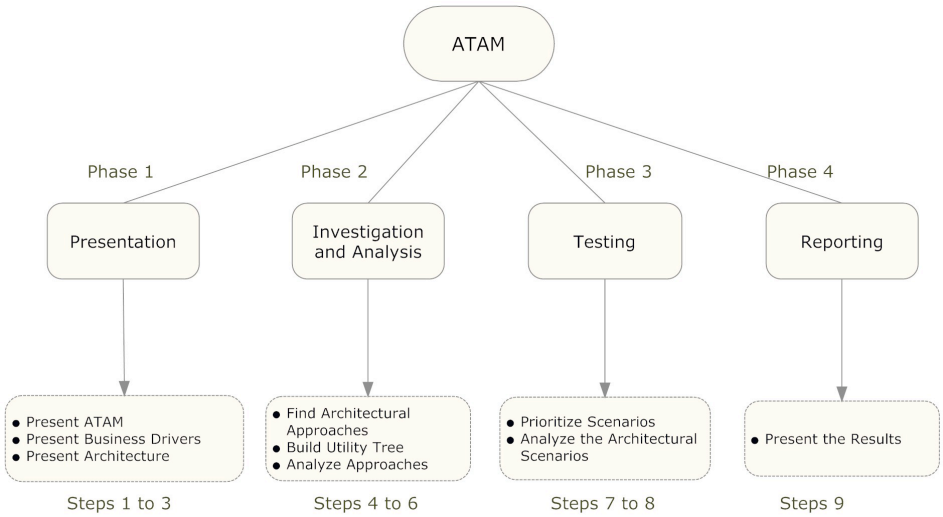


Fig. 7.6: Overview of the ATAM

Step 2: Present the Business Drivers

In the second preparation step the client presents the business drivers of the system as the business context, the most important functional requirements and what nonfunctional attributes the system has to fulfill. The client explains in an informal way why he mandates the development of the system, what its purpose is and what he wants to reach with it in his business area.

Step 3: Present the Architecture

In the following step, the software architect and his team present an overview about the architecture using diagrams. Its presentation can follow the recommendation for documenting software architectures as presented in the *4+1 Views* by Kruchten [62] or *3+1 Views* by Clements [55] (The views are: the module view, the component and connector view, and the allocation view). Clements has examined how high level design, as the architecture level is, can be modeled and presented by UML. He uses class and component diagrams to model the static view of the system, component diagrams with detailed interfaces and roles to model connectors and sequence diagrams to model the dynamic view of the system. Finally, deployment diagrams are used to model the allocation of the software components to physical entities of the system. The main advantages of UML are its standardization and tool support.
Additional information presented by the software architect concerns technical restrictions as system operating, hard/middleware support, other systems it has to interact with. Regarding the construction of the architecture, the architecture team explains architectural approaches and architectural styles that address the required quality properties. The evaluation team searches for risks in the architecture already.

Step 4: Identify Architectural Approaches

After the presentation phase, the next steps constitute the investigation and analysis of the architecture. The evaluation team searches for points in the architecture that are crucial for the accomplishment of the quality attributes. Predominant architectural decisions are identified. Examples for such extensive decisions are: The software architecture is designed as a *Client-Server* architecture, *Three Layer* or *Publish-Subscribe* architecture. Further important questions are: Does the architecture use redundant hardware to make the system more available due to possible hardware defects? Or: Is the system distributed?

Step 5: Generate the Quality Attribute Tree

In this step, different participants like members from the evaluation team, the project leader and the client work actively in generating scenarios and an utility tree. Each stakeholder has the chance to acclaim its essential requirements and functions the system should support. There are two approaches:

- **Top-down** approach means that the utility tree is created based on previously specified quality attributes and scenarios are linked to them in the end. For example, the stakeholders know that performance, maintainability are very important characteristics the system has to offer. They are added firstly to the utility tree as quality attributes. At last, scenarios describing the attributes more concretely are added as leaves to the utility tree.
- **Bottom-up** approach considers the elicitation of many scenarios at the beginning. After their generation, they are analyzed and classified to quality attributes on a finer level based on their quality requirements they address. At last, the specific quality attributes are connected to the top level attributes.

The picture 7.7 depicts an utility tree from the book *Evaluating Software Architectures: Methods and Case Studies* [61].

For both approaches, the quality attributes are structured hierarchically. At the beginning we introduce the *utility* node followed by the top level attribute nodes. Next, finer characteristics that are parts of the top level attributes are connected with the scenarios that represent the leaves of the tree. The parentheses give the weighting of the scenarios. Their importance is weighted by two dimensions. The first dimension indicates how important the scenario is for the success of the system (Business). The second dimension specifies how difficult its implementation is in the opinion of the software architect. Measures are *low*, *medium* and *high* (L, M, H).

The developers of the ATAM do not give concrete definitions about special quality attributes. The decision what quality attributes are important for owns's software architecture, and their personal interpretation are leaved to the evaluators. That is why evaluators may consider to refer to the standard ISO 9126 as described in section 7.4 and shown in figure 7.4.

Scenarios describe in an informal way desired interactions with the system. Three kinds of scenarios are used:

- *Use case scenarios* describe standard interactions of the users with the completed running system.
  For example: "A remote user accesses the data base with tool support."

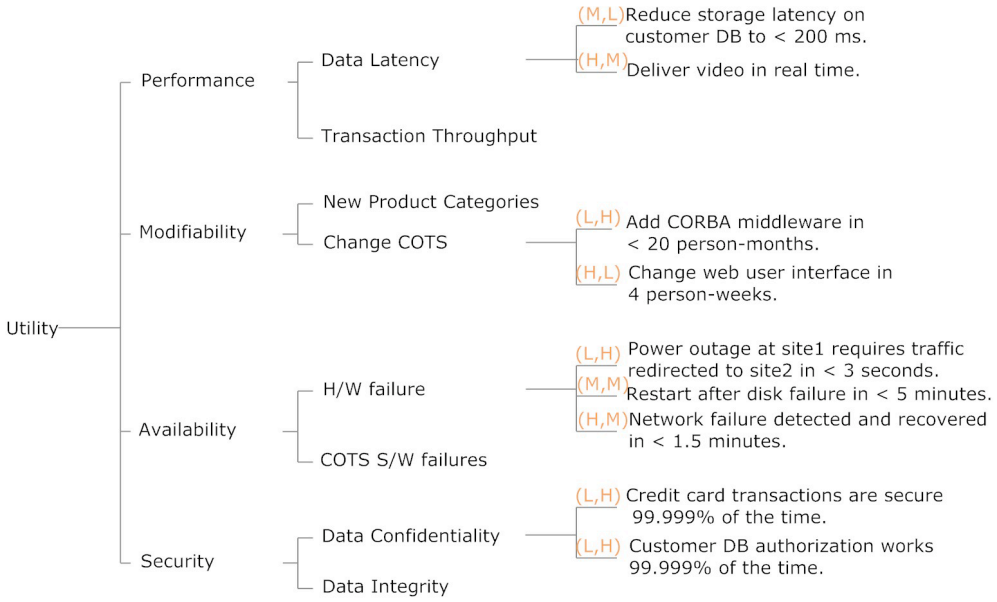


Fig. 7.7: An example of a quality attribute tree [61]

- *Growth scenarios* are anticipated future changes to the system. They can refer to any of the attributes (performance, modifiability, etc.).
  For example: "Migrate the system to the newest version of the operating system."
- *Exploratory scenarios* are used to see how the software architecture and the system respectively react under stress conditions. The goal is to see its limitations.
  For example: "Increase the number of simultaneously connected users to the online email system by a factor of ten."

Step 6: Analyze the Architectural Approaches

In that step, the evaluation team examines the architectural approaches from the perspective of the anteriorly specified quality attributes in order to discover risks in the architecture. Architectural approaches are identified that help to fulfill most important nonfunctional requirements of the system reflected by its software architecture. Specific quality question as described in *Attribute-based Architectural Style* (ABAS) [63] are asked to the architecture team. Risks, non risks, sensitivity and tradeoff points are identified and documented.

- A **risk** is an architectural decision that may influence the realization of a quality attribute in a negative way.
  For example: "The decision to use a backup system for the data base can influence the performance of the system negatively if the backup system is very slow."
- A **non risk** is an architectural decision which does not affect the success of the system but it is relevant to future possible changes on the requirements regarding the quality

attributes.
For example: "The decision to use a backup system for the data base is a non risk if the loss in performance is not high and the performance of the system is not very important."

- A **sensitivity point** is the property of a component that is essential for the success of the system.
For Example: "The encryption strength (Security) is sensitive to the bit size of the key."

- A **tradeoff point** is a property of the system that is related to more than one quality attribute or sensitivity point and influences them. They interact vice versa most of the time antagonistically.
For example: "If we use a backup system for the data base, the availability of the system is higher but the overall performance will suffer."

Step 7: Brainstorm and Prioritize Scenarios

This step is a repetitive step equivalent to the fifth one in that the circle of the participants is enlarged and new scenarios are generated. Often, a large number of scenarios are elicited, that is why they are prioritized by the stakeholders. Each participant gets a number of near thirty percent of the amount of scenarios as votes and distributes them based on their importance in his or her opinion.

Step 8: Analyze the Architectural Approaches

Most important scenarios, that were generated in step 7, are highlighted and analyzed by passing them into the software architecture to reflect to what extend they are complied. New risks, non risks, sensitivity or tradeoff points in the architecture may get visible to the audience.

Step 9: Present the Results

In the last step, the results of the evaluation and the personal experience accumulated by the analysis process are presented. The documentation of the requirements has improved. The software architecture was presented in an informative way at the beginning and later in a more formal presentation (with diagrams especially by using *UML* or *architectural description languages* (ADL)). Architectural decisions and styles are underlined and recorded. The elicitation of scenarios, the creation of an utility tree and the discover of risks, non risks, sensitivity and tradeoff points complete the evaluation.

The insights into the software architecture with its pros and cons gained by its evaluation make the stakeholders consciously about its capabilities and effectiveness. The developers of the ATAM affirm that very often the different stakeholders get a positive experience by communicating their expectations with other participants involved in the project after the evaluation meetings.

7.6 An example of the use of metrics for evaluating software architectures: EMS

This section describes how software metrics can be applied for the evaluation of a software architecture. The example presented in the following describes an *Experience Management System* (EMS). Its original implementation contains over 10 000 lines of code (LOC). Users asked for new functionality that the system could not offer in its first implementation. Because of that, the system had to be extended. Trying to extend the functional spectrum, the developers came into serious difficulties. They determined that the implementation of the new functionality was very difficult because of the extensive changes in the source code to be made. Further disadvantages of the previous system development are the absence of a documentation of the software architecture the implementation is based on and the poor documentation of the source code.
Because of the ripple effects caused by the initial unstructured software architecture and manifested in the source code meaning high effort to implement the new requirements, it was decided to design the system from the scratch. The way to go was the creation of a new software architecture as recommended and practiced in modern Software Engineering (see the Waterfall model).

The metrics should show now if the first software architecture would have to be extensively changed and if the second software architecture presents an improvement in design.

The analysis is based on late Software Architecture Evaluation (SAE) and the Goal-Question-Metric (GQM) [64]:

1. Selection of a perspective for the evaluation
   In our example the system had to be extended. For that reason, it is necessary to examine the software architecture in respect to the quality attribute, *maintainability* (*changeability*).

2. Definition and selection of metrics and guidelines
   A couple of metrics has to be chosen from a catalog (like Goal-Question-Metric) if particular metrics were already defined and correlated to questions regarding their influence on the software architecture. For the architectural abstraction level, *coupling metrics* are state of the art.

3. Acquiring metric values
   Tool support can be used to get the values for further processing.

4. Analysis and evaluation of the software architecture
   The results (metric values) have to be interpreted to gain meaningful information about the tested software architecture.

The main difference between the two software architectures is that the original implementation has library based modules. Each module communicates with all others without a restriction. The new software architecture is better structured. It is described by using components that clearly define their provided functionality specified by interfaces. A further essential feature of the newly designed architecture is the usage of the *Mediator* design pattern, by which the control flow is realized and determined. The other components are connected only with the *mediator* component and communicate completely over it. A disadvantage of the mediator pattern is that its implementation could get very complex.

The figure 7.8 shows both software architectures side by side for a better comparison.

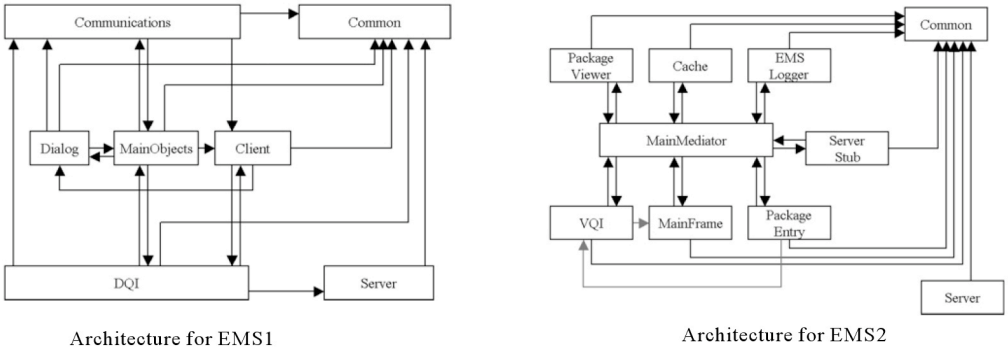


Fig. 7.8: Initial (EMS1) and planned (EMS2) architecture [65]

Coupling metrics are suitable to detect to what extend components, respectively modules, are coupled and as a result dependent of each other.
We consider the following coupling metrics for the investigation of the dependencies between modules and module classes:

- *Coupling between modules CBM*
- *Coupling between module classes* $CBMC_{all}$: Coupling to library classes are considered
- *Coupling between module classes* $CBMC_{nolib}$: Coupling to library classes are not considered
- *Coupling intra-module classes CIM*

The following picture 7.9 shows the metric values gained by measuring the connections between modules or classes as described by the coupling metrics.

For a better visual comparison and analysis of the data the metric values are presented in histograms as depicted by figure 7.10. The advantage of the histograms is the clear presentation of the coupling metric values of the two software architectures side by side.

In our experiment, we can calculate the median of the different metric values and verify the following expectations:

- *The coupling between modules is better for the new architecture*
  I.e. the median for CBM_{his}(EMS2) is less than the median for CBM_{his}(EMS1). This is indeed the case. While CBM_{his}(EMS1) is 5, CBM_{his}(EMS2) is 2.5.
- *The coupling between module classes is better for the new architecture*
  I. e. the median for $CBMC_{his}$(EMS2) is less than the median for $CBMC_{his}$(EMS1). If we take a look at the histogram $CBMC_{nonlib}$ we can see that eight of ten modules of EMS2 have a value less than ten by contrast to $CBMC_{his}$(EMS1), whose four of seven modules

Module	CBM	CIM	$CBMC_{all}$	$CBMC_{nolib}$
Client	**5**	1	10	9
Common	**6**	1	31	N/A
Communications	**5**	1.7	30	**22**
Dialog	4	1.3	18	**16**
DQI	**5**	4.5	39	**26**
Main Objects	**5**	1.6	41	**39**
Server	2	2	7	2

Metrics for EMS1

Module	CBM	CIM	$CBMC_{all}$	$CBMC_{nolib}$
Cache	2	0	9	3
Common	**8**	2	85	
EMS Logger	2	1	4	3
Main Frame	*3*	3.9	14	7
Main Mediator	**8**	1	37	**25**
Package Entry	*3*	1	13	6
Package Viewer	2	4.3	27	4
Server	1	2.25	12	0
Server Stub	2	0	4	3
VQI	*4*	2.9	23	7

Metrics for EMS2

Fig. 7.9: Table with metric values for EMS1 and EMS2 [65]

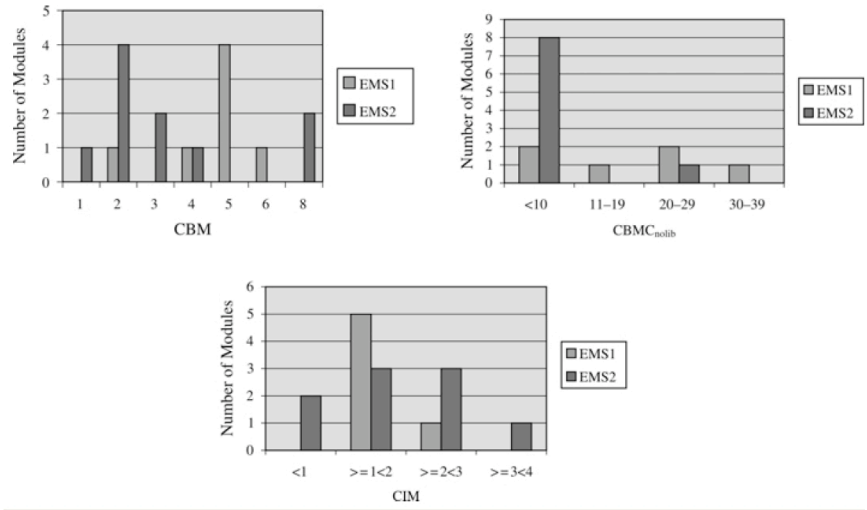


Fig. 7.10: Table with metric values for EMS1 and EMS2 [65]

have a value higher than ten. The communication of EMS1 is dispersed in comparison to EMS2 where the communication is localized.

- *The coupling intra modules is nearly identical for both architectures*
  I. e. CIM_{his}(EMS1) has the same value as CIM_{his}(EMS2). This is indeed true. The median for CIM_{his}(EMS1) is 1.6 and for CIM_{his}(EMS2) is 1.5. The request on the design is that the reduction of the coupling between modules is not transfered into the modules (classes). The classes inside a module should preserve their coupling value. That can only be achieved with a well defined design of the functionality on the class and component level. This design principle is known as *high cohesion* in Software Engineering. On the other hand the impact of high cohesion can also conduct to large and complex classes. The right balance between *loose coupling* and *high cohesion* is a challenge for every software architect and designer and should be found carefully.

With these results that confirmed the expectations, it is proven that the new software architecture outperforms the original one. Coupling between modules indicating references on the static view were measured. This approach of evaluation shows that the new architecture reduces the couplings and offers components with a well defined functionality. A further improvement is the use of the *mediator* architectural style. The communication flow is controlled by the mediator component. For that reasons, it can be assumed that the new architecture is easier to be maintained (ISO 9126) if future requirements on additional functionality has to be implemented (changeability).
As we can see in figure 7.4 more evaluating methods for the architecture exits that can be applied to test it regarding the quality attribute (maintainability - changeability).

7.7 Summary and outlook

This paper showed arguments for the necessity of a testing phase of the software architecture itself. Once the decision has fallen positively to examine the foundation of the software system, the software architect has to find an analysis method. Section 7.3 gave a classification of different techniques followed by a guidance that should help the software architect in finding a suitable one as described in section 7.4. The Architecture Tradeoff Analysis Method (ATAM) was presented in section 7.5. That scenario-based evaluation method is considered to be mature and finds appliance in industrial software projects. Its steps were described in more detail to get a clue about testing software architectures. The last section 7.6 showed an example how a software architecture can be analyzed by metrics well known from the object oriented software engineering. Coupling metrics play a central role in comparing two different designs, as we have seen in the example.

A comparison of different evaluation methods testing a particular software architecture of a real project regarding the same or different quality attributes was missing in my literature research.

A further question that was not clear while I wrote this paper, is which metrics are important on the architectural level. Empirically-based studies that confirm their effectiveness on this abstract high level are missing. Shereshevsky examines in his paper [56] on a theoretical basis coupling

metrics and their meaning on the architectural level. More concrete results are welcome in this research area. For the lack of proven guidance, the professional software architect must have experience on the programming level too, so that he or she can estimate the impact of coupling, or other metrics measuring implementation artifacts, on the architecture.

There is no algorithm for the conduction of a software architecture evaluation. Its success is often based on the experience of the evaluation team and the motivation of the participants. Nevertheless, participants report of a gain for the project, if we remember the nine steps of the ATAM, where lots of information concerning requirements, architectural description and analysis is processed.

An interesting approach for evaluating software architecture is described in the work by Penzenstadler and Cruz [66] where *architecturally significant requirements* (ASR) are correlated with *architecturally significant decisions* (ASD) using desin patterns. If we go from ASR to ASD we create an architecture. This is the design step of the software architecture. If we go the other way from ASD to ASR we evaluate our software architecture. For both processes, the creating and the testing one, architectural patterns play a significant role. That method reminds me of the *Attribute-Based Architectural Styles* [63] where attributes describing nonfunctional requirements are related to architectural styles in order to make the design or the evaluation (or both) of a software architecture easier for all participants.

8 Refactoring of Architecture

Stefan Deser
deser@in.tum.de

Abstract. This text examines the theory of refactoring of architecture and points out the current state of the art. It will be shown that the main challenges are: how to identify possible refactorings and assess their effects, how to deal with behaviour preservation and how to keep all software artifacts in sync after having applied a refactoring. Methods to deal with these challenges are explained in further detail: Patterns can be used as heuristics for the problem of identifiying possible refactorings. Furthermore software architecture analysis methods and metrics can be employed for more accurate assessments. For the task of behaviour preservation many approaches exist. The most important ones are graph transformations, first order logic and (as a very pragmatic approach) testing.

In the quest of maintaining consistency MDA will be shown a promising approach. Nevertheless, the refactorings of software architectures are influenced by many factors and hence it is difficult at the moment to invest on a future-proof technology. MDA, again, seems to be a good choice: It is capable of keeping artifacts in sync, it is developed by the OMG and it eases providing tool support. But still further research is needed, mainly due to missing UML features. Refactoring as a process in software development is also strongly influenced by and has impacts on management processes. Hence this should often be the starting point regarding decisions to architectural refactorings.

8.1 Introduction

Software is constantly subject to change and never finished. Its permanent development increases its complexity and bears resemblances to organic growth. Therefore the terms *software evolution* and *software life cycle* are widely used to refer to this phenomenon. Over the time requirements become clearer, new requirements evolve; the software is getting modified by having new functionality added or removed. This often results in the internal structure of the software becoming more complex and moving away from its original design. As in natural evolution, where the 'construction plans' for creatures undergo selection and mutation in order to adapt their fitness under altered conditions, software code has to be restructered from time to time, in order to make the software more maintainable, readable and extendable. The defining feature of this activity, which is called *refactoring*[1], is that it must preserve a programs external behavior while improving its internal structure [68].

[1]The term *refactoring* was originally introduced by William Opdyke in his PhD dissertation in 1992 [67].

Refactoring is normally done on the lowest level in the abstraction hierarchy of software development documents, i.e. on the source code. Nevertheless it is obvious, that refactoring on higher levels of abstraction, e.g. in a formal model given in the *Unified Modeling Language* (*UML*) or more generally on any architecture description, may imply many advantages, as Ivkovic and Kontogiannis state: "Since software architecture artifacts represent the highest level of implementation abstraction, and constitute the first step in mapping requirements to design, architecture refactorings can be considered as the first step in the quest of maintaining system quality during evolution" [69]. Changes on the architectural level "may lead to more significant, and positive improvements in the structure of a software system."

This text is structured as follows: First defintions and the main challenges are introduced. After exposing methods to deal with these problems a discussion section follows where remaining problems, pros and cons of the methods as well as further considerations will be discussed.

8.1.1 Definitions

We start with the defintions of the terms *Software Architecture* and *Refactoring* in order to narrow possible interpretations of this terms to a very common basis.

DEFINITION 1 *The software architecture of a program is its structure or structures of the components it is build of and the relationship among them.* □

It is very hard to give a precise though abstract and generic definition. Therefore additional information is hereby provided in order to illuminate its meaning: A computing system can be divided into two parts: A public part, which consists of its observable *behaviour* and a private part, which is its *internal structure*, leading to a visible behaviour. Software architecture is concerned with the private part of this division. Furthermore the definition points out, that *systems can and do comprise more than one structure* and that no one structure can irrefutably claim to be *the* architecture [10] and that *every computing system with software has a software architecture*, indepently from being explicitly described or not.

There are many definitions of refactoring as well, probably the most famous one is given in [70]. The following definition acknowledges them and tries to summarize their ideas.

DEFINITION 2 *Refactoring is the process of changing an existing software development artifact in such a way that its (internal) structure is getting improved while its (external) behaviour is preserved.* □

This definition addresses three important points:

1. The term *software development artifact* refers not only to source code, but to any document in any abstraction layer, therefore including architectural software descriptions.

2. The improvement lies in the internal structure regarding *qualitiy characteristics* such as *readability*, *maintainability* and *modifiability*[2].

[2] A quality characteristic or attribute is a nonfunctional property of a component or a system. A software quality is defined e.g. in IEEE 1061. Six categories of characteristics exist: functionality, reliability, usability, efficiency, maintainability, and portability. Further definitions can also be found in ISO 9126.

3. Although internal modifications have taken place, there is no difference in the functional behavior of the software. The external behaviour of the system must be preserved.

The general process of refactoring includes several steps [71] [69]. After having decided on the layer in the abstraction hierarchy whose artifacts should get refactored it has to be determined which refactorings should be applied. It then has to be guaranteed that the refactorings preserve external behavior, which can be shown before and/or after they have been applied. Furthermore the effects of the refactorings on the system quality attributes have to be assessed. Finally the consistency between the refactored elements and other software artifacts – e.g. source code and architecture documentation – must be kept.

8.2 Challenges

In the following we introduce the most important challenges being firstly to determine the abstraction level, on which the refactor should be applied, secondly how to identify possible refactorings, thirdly how to prove behaviour preservation and finally how to maintain consistency of all software artifacts.

8.2.1 Determine the Abstraction Level

The first challenge is to determine the appropriate level of abstraction to apply the refactoring. Refactoring in a 'traditional' meaning refers to the source code level. Nevertheless refactorings can be applied to more abstract software artifacts as well, e.g., to design models or requirement documents[3]. More recent approaches recognize that the high abstraction of software architecture artifacts enables a more powerful instrument for the quest of maintaining system quality during evolution. So the question to solve is: Which level of abstraction is the best to deal with refactorings?

Software architectures are capable of describing software systems at a high level of abstraction, ignoring code details. This ability is typically ensured by specifying them in a formal way using an *Architecture Description Language* (*ADL*). In the quest of evolving software architectures by refactoring this is a basic prerequisite. The software engineering community has developed several ADLs, such as *Acme*, *Wright* (both developed by Carnegie Mellon University) or *Darwin* (developed by Imperial College London). Unfortunately, these ADLs have not found widespread adoption in industry. UML, on the other hand, is a de facto industry standard for software modeling, and also includes a way to specify software architectures [72]. The question of the abstraction level hence also implies the question of which formal architecture representation, i.e. ADL, to use.

One further question arises by considering the fact that the refactoring process is embedded in a certain software development process: How does a certain *software development methodology* affect the refactoring process?

[3] Note that even if 'classical' refacoring is focused on the code, refactoring has a large impact on the design of a system [70]

8.2.2 Identification of Possible Refactorings

Even with given *how* to do refactorings (in the means of which techniques to apply) there relies the issue of deciding *when* and *where* a refactoring is recommandable or even required. For refactoring source code Fowler and Beck introduced the vague notion of "bad smells" [70], a set of heuristics to identify possible refactorings. Amongst others they list *Duplicated Code*, *Switch Statements* in object-oriented code, and *Speculative Generality*[4].

What heuristics or metrics can be used to identify these "bad smells" on the architectural layer? And how can be decided which quality attributes should be prefered in the case of competing possible refactorings?

8.2.3 Behaviour Preservation

Preserving behaviour is one of the defining attributes of refactoring, as stated in its defintion. There mainly exist two (not exclusive) ways of showing that an applied refactoring actually preserved behaviour. The first one is *formal verification* the second one is *testing* (in the means of pragmatically trying out the system behaviour by feeding the program with different input and observing the outcome). As the first one is executed *before*, the second one *after* the actual refactoring had taken place, one could in an epistemological sense call them *a priori* resp. *a posteriori* examination of behaviour preservation. (The original definition of behaviour preservation as suggested by Opdyke [67] states that, for the same set of input values, the resulting set of output values should be the same before and after the refactoring.)

The issues brought up by the challenge of preserving behaviour include: Is proofing behaviour preservation possible at all? If so, how can it be done best? What points will be hard to face?

8.2.4 Maintaining Consistency of Software Artifacts

During the evolution of a software, the changes made in the source code, have to be mapped and reflected against the architectural descriptions and vice versa. In the software development process documents normally exist on different abstraction layers, as e.g. the source code and the encompassing design documentation. These documents have to be kept in sync with each other. Changes in the system – on whatsoever level – should propagate through the whole document hierarchy. If a refactoring on any of these software artifacts has been done, there should be mechanisms to maintain their consistency. How can this be achieved (best) in refactorings of software architectures?

Contemporary IDEs provide support for refactorings to the source code, but not to design documents, e.g. UML models. Furthermore the automation of the refactoring process on a higher level is restricted by the complexity of a programming language: The more complex a language, the more difficult to automate the refactoring process [71]. The difficulty for a tool or formal method is to manage the balance between being sufficiently abstract to be applicable to different progamming languages, but also providing the necessary hooks to deal with language-specific behaviour. What is required to provide tool support and what approaches are made in current research?

[4] Note the similarity to the *Swiss Army Knife AntiPattern*, which will be introduced lateron.

8.3 Methodology

The general process of refactoring includes several steps [71] [69] – as stated in the introduction. The challenges of this process have already been introduced. In this section methods are shown which help solving (or at least simplifying) them. *Patterns* as heuristics and *architecture analysis methods* with corresponding *metrics* face the problem of identifying possible refactorings and deciding which to apply. Furthermore methods for proofing *behaviour preservation* will be pointed out.

8.3.1 Identification of Possible Refactorings

In the task of identifying possible refactorings as well as in assessing the quality of a given architecture *heuristics* can be used, which mainly serve as a rule of thumb and provide very generic proposals. Futhermore there are *analysis methods* for software architectures, which can provide further assistance.

Patterns as Heuristics

Gamma et al. [73] describe general reusable solutions to commonly occuring problems regarding software design, which they call *design patterns*. A design pattern consists of a description of the context, where the given pattern can be applied, a generic problem description, and a rough solution proposal, which is not a finished detailed design capable of being directly transformed into running code, but rather a experienced-driven heuristic. Design patterns are often used in the context of object-oriented design (but not limited to them), where they focus on relationships and interactions between classes or concrete instances of classes and make use of delegation and inheritance.

Although design patterns are a conventional abstraction technique and capable of providing targets for refactorings, they are mainly suitable for lower levels than the architectural. Therefore – following the concept of design patterns – Buschmann et al. [74] introduced (eight) *architectural patterns*[5], each of them describing a particular recurring design problem that arises in specific design contexts as well as a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate. A division into four categories is made: *From Mud to Structure*, *Distributed Systems*, *Interactive Systems*, and *Adaptable Systems*. Famous examples are *Layers*, *Broker*, *Model View Controller* (*MVC)* and *Microkernel*, respectively. Architectural patterns as the highest-level patterns are intended to provide the ”skeleton of an overall system architecture” [74]. To illustrate the motivation and idea of architectural patterns, the MVC pattern is explained in further detail:

- **Context**: Interactive applications with a flexible user interface.
- **Problem**: User interfaces are especially prone to change requests, therefore its modifiability has to be taken into consideration before or while building the system. Support

[5] Actually [74] not only introduces *architectural patterns*, but also refers to *design patterns* and even very low level language-dependent patterns, which the authors call *idioms*.

for several user interface paradigms (e.g. clicking icons and buttons, using a text-only interface with a keyboard) should be easily incorporated.

- **Solution**: The MVC pattern divides an interactive application into three components. The *model* contains the core functionality and data. *Views* display information to the users, whereas *controllers* handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model. The separation of the model from view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all other views dependent on this data should reflect the changes. The views in turn retrieve new data from the model and update the displayed information.

In addition to this generic descriptions usually also a illustrating graphical representation is given and the dynamics as well as variants of the pattern are discussed. Furthermore implementation proposals can be given in a concrete language. Note that the number of classes needed may depend on the features provided by the language and also that architectural patterns can be tightly interwoven with design patterns: The MVC pattern is proposed to use four classes in C++, whereas three in Smalltalk [74]. Moreover the classes for view and controller basically implement the observing part of the *Observer design pattern*.

Design patterns are commonly used but due to their high abstraction level they are often inherently ambiguous and easy to misunderstand, such that – sometimes – their consequences outweigh their advantages: A design pattern can become an *AntiPattern* when it causes more problems than it solves. Nevertheless AntiPatterns are also are very useful 'negative' heuristics in the means of how not to do it. Furthermore the authors of the book having first dealed with AntiPatterns [75] state that "there are many more AntiPatterns in practice than design patterns."

They divide their set of AntiPatterns into three primary categories: Whereas the key goal of *Software Development AntiPatterns* is to describe useful forms of software refactoring, *Software Architecture AntiPatterns* focus on common problems and mistakes in the creation, implementation, and management of architecture. The *Management AntiPatterns* finally try to identify some of the key scenarios in which human communication and interaction are destructive to software development processes. As the management of software processes is beyond the scope of the operations a software architect normally has to fulfil, in the following some of the most important Software- and Software Architecture AntiPatterns are given[6].

- **Software Development AntiPatterns**

  The *Blob* denotes a class with a large number of attributes, operations, or both. (As a rule of thumb a class with 60 ore more attributes and operations.) Often caused by procedural-style design one object holds a lion's share of the responsibilities, while most other objects only hold data or execute simple processes.

  The *Lava Flow* AntiPattern is commonly found in systems that originated as research but ended up in production. It describes the state of a system, where a lot of new code is built

[6]The style of speech used in the description of these AntiPatterns is often very non-technical and metaphoric due to the high abstraction for this very generic approach.

around so called 'dead code' (not used code). Dead code is not deleted due to lack of knowledge or comfort and leads to more complex and less readable source code.

Poltergeists are less important classes with limited responsibilities and therefore their effective life cycle is quiet brief. Poltergeists clutter software designs, creating unnecessary abstractions[7].

Cut-and-Paste Programming as another common AntiPattern leads to significant maintenance problems. (Remember that duplicated code is a "bad smell" (see 8.2.2 on p. 112.)

- **Software Architecture AntiPatterns**

  A *Vendor Lock-In* happens, when a software project becomes completely dependent upon a certain vendor. This is especially problematic on upgrading software, where interoperability problems can occur and continuous maintenance is required to keep the system running. In addition, the vendor can easily delay the development of new needed product features and compromise the development of software depending on this features. The solution proposed is basically to introduce indirection, that means an *isolation layer* that abstracts the underlying infrastructure or product-dependent software interfaces (similar to the *Adapter design pattern*).

  Architecture by Implication is another AntiPattern that is characterized by the lack of architecture specifications for a system under development. The problem of not having good architecture documenation leads to adding new pieces of software unsystematically. A solution entails an an organized approach to systems architecture definition, and relies on multiple views of the system.

  Design by Committee denotes the problem of a software design being the product of a committee process. Because of having too many features it is infeasible for any group of developers to realize the specifications in the restricions regarding time and budget. Moreover the specification defects (excessive complexity, ambiguities etc.) make exhaustive testing virtually impossible. The solution proposed is essentially to reform the meeting process and designate a project architect and his or her responsibilities.

  Swiss Army Knife is an excessively complex class interface, where the designer attempts to provide for all possible uses of the class. This is problematic because it ignores the force of managing complexity in the means of other progammers to understand it. As a guideline a class interface should be as small as possible (but as extensive as needed).

Architecture Analysis Methods and Metrics

Basically the assessment of the effect of a refactoring is made by a comparison between the architecture before and after the refactoring. Therefore and for the identification of possible refactorings software architecture analysis methods (in addition to patterns) are needed. Unfortunately there is no one standard method and various (often just slightly) different methods

[7] The suggested solution to this AntiPattern is called *Ghostbusters* and simply removes Poltergeists from the class hierarchy altogether.

exist, each emphasizing another aspect, though these methods are in part complementary. The very good overview given in [76] lists the following important methods[8]:

- SAAM (Scenario-based Architecture Analysis Method) and its extensions:
  - SAAMCS (SAAM Founded on Complex Scenarios)
  - ESAAMI (Extending SAAM by Integration in the Domain)
  - SAAMER (Software Architecture Analysis Method for Evolution and Reusability)
  - ASAAM (Aspectual SAAM [77].)
- ATAM (Architecture Trade-off Analysis Method)
- SBAR (Scenario-based Architecture Reengineering)
- ALPSM (Architecture Level Prediction of Software Maintenance)
- SAEM (Software Architecture Evaluation Model)

Software architecture analysis methods aim to assess the quality of a system independent from its concrete implementation – mostly *before* it has been built. They focus on quality attributes such as e.g., complexity or maintainability. There are basically two classes of evaluation techniques available at the architecture level: *questioning* and *measuring*.

- *Questioning techniques* generate qualitative questions to be asked of an architecture and they can be applied for any given quality. This class includes scenarios, questionnaires, and checklists, with scenarios being most widely spreaded. Analysis methods focusing on scenarios take as input the architecture design and measure the impact of predefined scenarios in order to identify the potential risks and the sensitive points of the architecture. This helps to predict the quality of the system before it is built.
- *Measuring techniques* suggest quantitative measurements to be made on an architecture. They are used to answer specific questions and they address specific software qualities and, therefore, they are not as broadly applicable as questioning techniques. This class includes metrics, simulations, prototypes and experiences. Mens and Tourwé state that the use of object-oriented metrics, especially in combination with software visualization, seems well suited to identify possible refactorings in the source code [71]. Moreover the measurement of quantifiable properties of soft-goals (i.e. quality attributes) is necessary to prioritize the refactorings to apply.

  For example the six metrics Chidamber and Kemerer have developed in [78] could be used. They make proposals how to measure e.g. the *Depth of Inheritance Tree* (*DIT*), the *Number of Children* (*NOC*) and how to estimate coupling and cohesion as in *Coupling between object classes* (*CBO*) or *Lack of Cohesion in Methods* (*LCOM*). Nevertheless,

[8] The various methods will not be explained in further detail at this point; an own chapter deals with the task of analyzing software architectures.

metrics should not only be applicable to source code but rather to formal software description models, such as UML. Van Kempen et. al. use a tool called *Saat*, which is able to calculate metrics about UML models [68]. These metrics can then be used in analysing the model for potential flaws or AntiPatterns[9]. For example the Blob-AntiPattern should be easier locatable with this tool.

In terms of quantitative and qualitative aspects, both classes of techniques (questioning and measuring) are needed for evaluating architectures. Scenarios are necessary but not sufficient to predict and control quality attributes and they have to be supplemented with other evaluation techniqes and, particularly, quantitative interpretations.

If the soft-goals a software should fulfil are determined, the problem still exists how to prioritize them. For example the two quality attributes (or soft-goals) of security and efficiency are counterparts; (normally) one cannot improve both at the same time. An interesting approach to deal with this interdependencies is the so called *soft-goal interdependence graph* (*SIG*), which is used e.g. in the approach of Ivkovic and Kontogiannis [69].

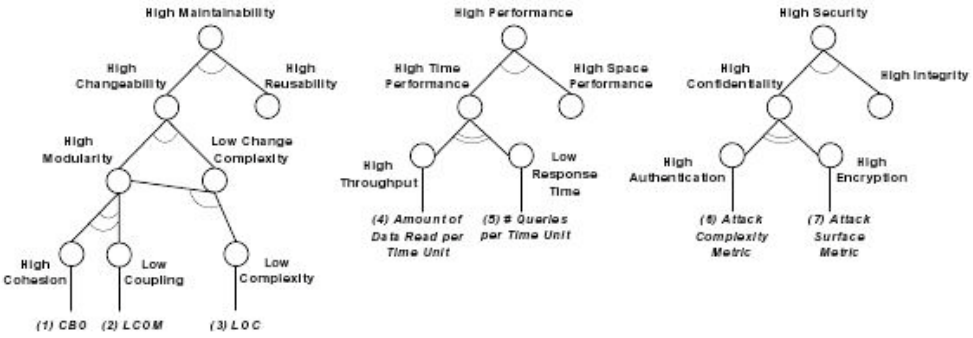


Fig. 8.1: An example of a soft-goal interdependence graph

8.3.2 Behaviour Preservation

Proofing the preservation of behaviour for refactored software can be done in several ways in principle. Basically the methods can be differentiated along either *formally proofing* or *testing*.

Formalisms for Behaviour Preservation

One fundamental approach is to formally prove that refactorings preserve the full program semantics, which can be done before or independently of the refactoring itself. Regarding refactoring in general basically several methods can be considered, as for example *type checking* (which mainly refers to the source code), *graph transformation*, using *first order logic* or *assertions* (invariants, pre- and postconditions) to name the most important ones.

[9] Typically high values for (dynamic) coupling and method call metrics indicate potential future problems regarding maintainability [68].

- **Graph transformation**

  Programs (or other kinds of software artifacts) can be expressed as *graphs*. Refactorings then correspond to *graph production rules*, the application of a refactoring corresponds to a *graph transformation*, refactoring pre- and postconditions can be expressed as *application pre- and postconditions* [71]. Hence the theory of graph transformations has been used to provide more formal support for software refactoring. Mens and Tourwé provide a list of additional literature, where e.g. graph rewriting formalism is used to prove the refactorings preserve certain kinds of relationships that can be inferred statically from the source code. Bottoni et al. [79] describe refactorings as graph transformation schemes in order to maintain consistency between a program and its design when any of them changes due to refactoring.

- **Mathematical Model**

  Using a precise mathemical model Philipps and Rumpe [80] propose a "promising approach", as Mens and Tourwé state [71]. In this approach refactoring rules are based directly on the graphical representation of a system architecture and a precise mathematical model is employed that – in their own terms – is "simple, yet powerful". They basically divide a system architecture into a set of *components* and their *connections* following the suggestive box-and-arrow approach that is common in informal notations. Arrows are interpreted as data flow channels and boxes as components that process data flows. The motivation for the authors to introduce their calculus is that they consider it crucial for the applicability of a formal method. It should be incorporated easily into CASE-tools, which then allows for a graphical manipulation of architectures or dataflow networks, respectively.

  Another approach by van Kempen et. al. focusing on UML uses a *CSP*[10]-based formalism, to describe the refactoring, and show that the refactoring indeed preserves behaviour [68]. Every class in the UML model gets its own statechart, defining its behaviour, which gets mapped to CSP notation. Finally the CSP-mappings of the behaviour before and after the refactoring are compared in order to compute whether the behaviour will be preserved or not.

- **Invariants, pre- and postconditions**

  The use of preconditions and invariants has been suggested repeatedly in research literature as a way to address the problem of behaviour preservation when restructuring or refactoring software artifacts. For example van Gorp et al. propose a UML extension to express the pre- and postconditions of source code refactorings using *OCL*[11]. This approach tries to enable refactoring independent from the used (object-oriented) programming language by proposing a minimal set of UML metamodel extensions.

[10] *Communicating Sequential Processes* (*CSP*) is a formal language that aims at describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi.

[11] *OCL Object Constrained Language* is part of the UML.

Testing

Testing plays a very important role in refactoring. Fowler [70] even describes it as its first step: "If you want to refactor, the essential precondition is having solid tests. Even if you are fortunate enough to have a tool that can automate the refactorings, you still need tests." This crucial role of testing is indeed due to the *agile development methodology* Fowler prefers, namely *XP* (*Extreme Programing*), but nonetheless a rigorous testing discipline is a very pragmatic way to deal with behaviour preservation.

The principle of testing is that certain results are expected and compared to the actual results executing a test. Of course the granularity of testing can widely differ: There are tests for small progamming units, for the integration of this units or for the whole built system. Furthermore the testing procedure is highly dependent on the employed software development model. Nevertheless, it is obvious that testing allows the comparison between an expected and an actual behaviour and is therefore well-suited to empirically prove or disprove behaviour preservation.

8.3.3 Maintaining Consistency

The quest of maintaining software artifacts of different abstraction levels in order to keep them in sync (highly) depends on the employed software development methodology. Whereas in some of them a synchronisation is very hard, one recent software development methodology, being capable of this feature in principle, is *Model-Driven Engineering* (*MDE*). MDE focuses on creating models (abstractions) more close to some particular domain concepts rather than computing (or algorithmic) concepts. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. As the models approach completion, they enable the development of software and systems. The best known MDE initiative is *Model-Driven Architecture* (*MDA*)[12][81] [82].

An important medium in MDA are *model transformations*, which take as input a model conforming to a given metamodel and produce as output another model conforming to a given metamodel. The basic proceeding is to create a *Platform Independent Model* (*PIM)* initially, a model with a high level of abstraction that is independent of any implementation technology. The PIM is then transformed into one or more *Platform Specific Models* (*PSM*). A PSM is tailored to specify the system in terms of the implementation constructs that are available in one specific implementation technology. The final step in the development is the transformation of each PSM to code. Further details on the MDA development life cylce can be found in [82]. However the crucial benefit of the MDA-approach is that it not only refers to forward engineering, being the transformation from abstract to more conrete levels, but also to reverse engineering in the sense of extracting high level information out of low level documents – the source code.

In MDA the UML is used as an ADL; unfortunately, seen as such, UML tends to be much more ambiguous and informal than the 'traditional' ADLs [72]. Nevertheless there is a "wide

[12] MDA is an *Object Management Group* (*OMG*) initiative and a registered trademark of OMG. (The OMG consortium itself was founded in 1989 and sets its focus on the development of standards for manufacturer-independent and distributed Object Oriented Programming. It has more than 800 members, e.g. IBM, Apple and Microsoft. The OMGs website is located on http://www.omg.org/.)

acceptance (in both industry and academia) to consider and use UML 2 as an ADL" [72] and it is a relatively new area of research [68]. Many of the approaches, which employ UML for architectural refactorings ([68] [83] [69] [84]) hence propose an extension to the UML standard using its built-in extension mechanisms (namely profiles and corresponding stereotypes). E.g. the motivation of Ivkovic and Kontogiannis is to standardize and formalize quality-driven refactoring using UML. They make use of *UML profiles*[13] to represent different architecture modeling views in a standardized format and also suggest to employ *model transformations* and *semantic annotations* [69]. Riva et al. proposes a UML-based software maintenance process, which is guided by architectural descriptions and models [84]. These descriptions are given as variants of UML profiles.

Refactoring seems to fit well into a model-driven (re)engineering process, because the idea of MDA incorporates forward- as well as reengineering from models to source code and vice versa and hence supports the process of keeping consistency of software artifacts of different abstraction levels.

8.4 Discussion

In theory the benefits of the approach to refactor on the architectural layer is that – if performed correctly and accurately – the changes may lead to more significant, and positive improvements in the structure (architecture) of a software system compared to the relatively expensive refactorings only on the source code level. In practice there are many challenges to face (see section 8.2) which impacts and influences will be discussed in the following.

The discussion is driven by the fact, that refactoring – as software development in general – is highly influenced by many factors, such as the *development methodology*, the *application domain*, the *programming paradigm and language* and the *size of the software project*.

8.4.1 Determine the Abstraction Level

In the problem of determining the abstraction level of a refactoring it was silently assumed that there always are artifacts on different abstraction levels – but this actually also depends on the software development method the refactoring is embedded in. Whereas some development methods start with very high abstracted artifacts as a (formal) architecture description, other ones as *agile* software development methods (e.g. *Extreme Programming*) work without an explicit and detailed architecture description. Hence it has to be thought about whether refactoring on the architectural level can be done and should be done at all. The methodology depends on the size of a project. In bigger software projects high abstraction levels and corresponding documents simplify the problems of dealing with complexity. In smaller projects the effort of producing such documents outweighs its advantages. The time consumed to abstract and formalize could

[13] A UML 2.0 profile allows to extend or adapt the metamodel of UML 2.0. It is a collection of stereotypes, which specify how an already existing metaclass, which is given by the metamodel of the UML, can be adapted to a specific (application) domain. In the approach of Ivkovic and Kontogiannis the conceptual architecture view is represented as a UML 2.0 profile with corresponding *stereotypes*.

already suffice for actually implementing the software. In the latter case tending to source code refactoring probably makes more sense than refactoring architecture descriptions.

A further issue to deal with is the high dependency on the particular application domain [71]. Consider, for example, web-based software in comparison with (parts of) operating systems. Even worse there are many different programming paradigms, e.g. imperative, functional, logic, object-oriented, or aspect-oriented and corresponding languages, e.g. C, Lisp, Prolog, Java or AspectJ, which all have influences on how and where to refactor. A overview of support for program restructuring and refactoring is given in [71][14].

8.4.2 Behaviour Preservation

One defining property of refactoring is the preservation of behaviour. The definition assumes, that it is clear, what exactly is meant by the term 'behaviour', but unfortunately, a precise definition of behaviour is rarely provided, or may be inefficient to be checked in practice [71]. Even if there is a clear conception of behaviour, in many application domains requiring the preservation of input-output behaviour is insufficient, since many other aspects of the behavior may be relevant as well [71]. For example developers have different prospects regarding real-time software, embedded software or safety-critical software. There it may not be sufficient that the same output sets are generated for the same sets of inputs before and after the refactoring but it may also be of high importance that certain timing aspects are considered as well [68].

This problem supports the division of showing behaviour preservation with two methods: formal proving and testing. On the hand formal proves have big advantages: Stated that the transformations of system behaviour into the notations of the choosed formal method have been done correctly the strength of the employed calculus can serve for a very high degree of certainty, i.e. one can literally proove that everything will work as expected. Futhermore the chances are high that the calculus can be incorporated into software, which then automates the proof. But on the other hand there are two main disadvantages: First the question arises, whether it is possible to tranform the system behaviour correctly into the calculus. Second the more complex the system behaviour is the more difficult is this transformation and the more time and memory has to be consumed. Treating a very large system the computing resources may not be enough to automatically prove the program.

Hence not all improvements of the internal structure can be proved (a priori) to not alter the external system behaviour. Nevertheless one can rely on testing as an a posteriori examination. But it has to be taken into consideration that testing is not capable of proving program correctness; it *only* can show the presence of failures, not their absence! If a test of a program did not reveal any failures this does *not* imply that there do not exist any failures! Perhaps they just were

[14] Generally the more complex a language is the harder to refactor. For example preprocessor directives in a C/C++ program are not part of the actual language syntax. Some languages even support many different programming paradigms (e.g. Python, C++), which has to be taken into consideration if a consistent procedure should be applied. Furthermore Mens and Tourwé state that programs that are not written in an object-oriented language are more difficult to restructure because data flow and control flow are tightly interwoven. Therefore restructurings are typically limited to the level of a function or a block of code. On the other hand they argue that the very nature of object-oriented principles makes some seemingly straightforward restructurings surprisingly hard to implement. This is due to the inheritance mechanism, dynamic binding, interfaces, subtyping, overriding and polymorphism [71].

not found. The question, whether one relies solely on testing depends on the requirements of the software. In critical application domains (e.g. in avionics) failures are not only an unfortunate event probably leading to financial problems but can directly cause death to many persons. It would be irresponsible to only relying on testing in this case. In non-critical application domains this argument does not hold. As testing is a regular part of the software development process one could argue that no extraordinary activity (namely the formalisation and proving of system behaviour) is required.

8.4.3 Maintaining Consistency

Independent of the software development methodology there (normally) exist software artifacts of different abstraction levels, e.g. a software architecture description, a detailed design given in the UML and the source code. Whenever one artifact is getting modified the changes have to be *propagated* through the whole abstraction hierarchy in order to keep the consistency of the overall software documentation. This is not an exclusive problem of refactoring on the architectural layer but especially important in this case. Given an architecture and corresponding program the question arises how the source code of the program can be modified correctly according to a refactoring of the architecture *after* the refactoring had already taken place. At this point tool support is desirable; e.g. to establish certain references between the different software artifacts.

Fortunately there exist MDA tools that provide all the infrastructure needed to solve this consistency problem. Namely a configurable parser, model transformater and code generator. So developers can parse their existing program sources to UML and use the resulting design model to reason about possible refactorings. These refactorings can then be propagted to the source code again. A MDA tool supports them in preserving all occurrences of the metamodel entities and their surrounding text to regenerate the executable system. As a result, the extracted UML design has been improved without loosing its consistency with the source [83].

So van Gorp et al. state that it is both *feasible* and *desirable* to use the UML as a way to refactor designs independent of the underlying programming language. They think the coming generation of MDA tools will allow for a seamless integration between modeling and coding.

8.4.4 Missing Standardization and Tool Support

Philipps and Rumpe utter their hopes that their article about a precise mathematical model to deal with behaviour preservation will be of some influence to the software architecture community, because – as they state – the definition of architecture is still rather informal [80].

Looking at the many proposals for (semi-)formal (meta-)languages for describing architectures, for analyzing architectures and for proving behaviour preservation, there seems to be no overall standard for each of these aspects.

As a very interesting ADL the *Architecture Analysis & Design Language*[15] (*AADL*) was not developed in the academic environment, but is standardized by SAE, an organization for mobility engineering professionals in the aerospace, automotive, and commercial vehicle industries. AADL is used to model the software and hardware architecture of an embedded, real-time system, i.e. it is a *specific* language emphasazing the embedded domain. This underlines the already

[15] More information is available at its website http://www.aadl.info

mentioned point that there are strong influences by the context; in this case the application area (aerospace and automotive) and corresponding solution domain (embedded systems).

In the case of architecture analysis methods it was shown, that there exist many different ones too (see 8.3.1 on p. 115). They differ in their focus on specific quality goals and unfortunately only few have been documented to be widely validated in practice [76]. The same basically holds for the hard work of defining formal methods to prove behaviour preservation. There is a broad variety of approaches (e.g. using graph transformations or first order predicate logic) each diversified in its concrete realizations. This variety implies difficulties to develop generic standards on a common basis. There are too many dependencies on the context – especially regarding the application domain – which leads to a heterogeneous field of approaches and in conclusion to poor tool support. Nevertheless UML seems to be promising because it is generic (but extendable) and (what is of most importance) widely spread and (open) standardized.

Whereas refactoring on the source code level is well tool-supported, e.g. in the different IDEs, because the specification of the specific used language is very clear: It is relatively easy to implement e.g. the movement of a method, the renaming of variable etc. On the architectural level in contrast it is much more difficult, because there is no solid formal specification, where a tool hook could be established. Furthermore the standard UML metamodel is inadequate for maintaining the consistency between a design model and the corresponding program code [83]. But luckily it is extensible. Van Gorp et al. suggest such an extension to UML, which enables the UML to deal with refactorings in the means of keeping source code and model in sync. Refactoring pre- and postconditions should be specified in OCL. This enables the runtime verification of behaviour preservation of a refactoring engine: “Summarizing, by implementing the OCL refactoring contracts we proposed in this paper, UML CASE tool vendors can support automatic refactoring composition.” [83]

8.4.5 Impacts on Project Management

The software development process and hence refactoring (on whatever layer) is embedded in a project, which has to be managed. In this subsection we will deal with questions which cannot be answered within architectural but managerial considerations.

Erich Gamma states in the foreword of [70] that refactoring is the key to keeping code readable and modifiable and it therefore is essential for the quality of a software product. Refactoring is a necessary step in the quest of keeping software on a high quality level. However it is *risky* and becomes even riskier when practiced informally or ad hoc – it must be done systematically. Deciding on refactorings regarding the involved risk as well as organize them in the means of assigning responsibilities, keeping track of time and budget constraints is an important task of the project management and a necessary premise to a successful project implementation.

One of the main problems with refactoring is that its effects are often not immediately visible. The quality of a software in the means of extensibility, maintainability and of improving the implementation due to changes in the nonfunctional requirements of a system cannot be illustrated as easily as the addition of new functions, e.g. by the presence of a new element to the graphical user interface. Furthermore the project management is under pressure regarding time and budget, such that it *seems* reasonable not changing anything on a a working system. Therefore

the awareness for the meaning of refactoring, its long-term impact on the software-quality, has to be constantly made aware to the ones responsible for the software development process. It must be one task of the management to constantly emphasize the meaning of refactoring and being aware, that refactoring is an important *investment*: “Flexible, reusable code requires an investment in its initial development, otherwise long-term benefits will not be achieved.” [75]

That the process of refactoring and project management is closely interconnected is implicitely claimed in the already introduced heuristic of the AntiPatterns [75]. For example the *Vendor Lock-In* AntiPattern also impacts risk management. The *Architecture by Implication* AntiPattern evolves due to missing risk management. Also the authors are explicitly speaking of *Software Architecture* AntiPatterns, it is obvious that these patterns are interdependent with the management process. In the context of the *Wolf Ticket*[16] the authors even state that they “are working on an initiative in technology consumer politics” which is all but an issue of software architecture. In addition to the involvements of the management the authors call for developers “to be up to date on technology trends, both within the organization’s domain and in the software industry at large” and appeal “to adopt a commitment to open systems and architectures”. The management should ”actively invest in the professional development of software developers, as well as reward developers who take initiative in improving their own work.”

8.5 Conclusion

Refactoring of software architectures is influenced by very many factors: the software development methodology it is embedded in, the application domain, the solution domain (programming paradigm and language) and the various methods which were developed for its different challenges. Patterns as heuristics are often wrongly interpreted, there exist a lot of SAAMs as well as ADLs and methods for proving behaviour preservation.

Therefore it is very difficult at the moment to invest on the “right” technology. *Standardization* is hence of utter importance and as long as this broad variety of approaches for each method exist a big progress in this area of research seems not to be very likely. On the other hand, one has to admit that this area of research is relatively new and therefore not that well developed.

Future works should focus on the possibilities of employing MDA. It seems to be a promising approach, because – in theory – refactorings on the source code as well as on the design level (using UML) can be done without loosing the consistency of the affected software artifacts. Its main advantages are that it can keep software artifacts in sync, that UML as well as MDA are developed under the patronage of the OMG and that tool support is more easy to provide. UML on the other hand is not completely perfect and further extensions have to be discussed and adapted.

As seen in the discussion section refactoring is also influenced by and has impacts on management processes. Hence this is often the starting point regarding decisions to architectural refactorings: No refacoring is able to completely undo mismanagement and well managed refactorings can better realize their potential.

[16] This AntiPattern refers to a product that claims openness and conformance to standards that have no enforceable meaning.

9 Architectural Styles

Michael Weber
weberm@in.tum.de

Abstract. With the steadily increasing complexity of todays software systems, the architecture of such a system is a crucial factor and has a great impact upon the success or failure of its development. As software engineers have built a multitude of software systems, they were able to learn from the past projects how different software architectures performed in reality. Thus, an experienced software engineer will favor similar software architectures that have already proved its quality in past projects. Architectural styles are a possibility to document a generic architecture, its constraints, and the qualities induced by the resulting architecture. Hence, software engineers that are familiar with architectural styles may derive a software architecture by applying the styles in order to elicit desired qualities.

9.1 Motivation

Reuse of software and software components is an important aspect in the domain of software engineering. The reuse becomes even more important with the increasing complexity of todays software systems.

In order to build larger and more complex software systems, the focus during the development moved more and more to the software architecture. With software architecture as an emerging discipline [85], the aspect of reuse gained a growing importance in the software architecture, too.

Over the time, software engineers have built a wide variety of software systems. It became obvious that some architectural decisions have superior properties, for example regarding efficiency, evolvability, and scalability, than other possible architectural design decisions [86].

A good software architect should have a set of experiences that allows him to choose a solution for a problem that already proved in the past, instead of reinventing the wheel each time. The goal is also that such solutions lead to an architecture that is consistent and easy to understand. Hence, it is important that such a wealth of experience is not only in the mind of several software architects. Instead there is a necessity that this knowledge is uniformly documented, so that other architects can also benefit from this experience [87].

Architectural styles are a primary way of characterizing lessons from experience in software system design [86] and can be interpreted as defining a family of software architectures [88]. According to Taylor et al. [86], architectural styles are a named collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system.

The importance of architectural styles was already assessed in 1992 by Perry and Wolf [89]:

> «The important thing about an architectural style is that it encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. The useful thing about style is that we can use it both to constrain the architecture and to coordinate cooperating architects. Moreover, style embodies those decisions that suffer erosion and drift. An emphasis on style as a constraint on the architecture provides a visibility to certain aspects of the architecture so that violations of those aspects of insensitivity to them will be more obvious.»

Abowd et al. [90] gave a first thorough introduction to architectural styles by presenting a formal definition of the pipe-and-filter style. Later, Shaw and Garlan [85] presented a comprehensive exposition of several architectural styles and their appliance, which led to an increasing popularity and acceptance of architectural styles. Today, architectural styles have become an integral part of current architectural design.

The next chapter gives an introduction to architectural styles. Some of the most common architectural styles that can be used as a basis for the architecture of software systems are presented in Section 9.3. The presented basic styles are intended to provide the foundation for the following Section 9.4, which introduces two more complex styles that are built by using several basic styles.

The presentation of the styles in Section 9.3 and 9.4 is neither intended to be in the full extent regarding details nor is the list of styles complete. Rather, the intention of this work is to give an overview and to provide the general idea of styles. For a presentation in depth [85, 74, 91, 92, 86] should be considered.

9.2 Introduction to Styles

9.2.1 Architectural Style Elements

An architectural style consists — like a software architecture per se — out of a collection of computational *components*. These components can be, for example, clients, servers, databases, filters, and layers. To enable an interaction between these components, they are connected through *connectors*, like (remote) procedure calls, events, protocols, and pipes [85]. A *topology* describes what connections between the components are allowed. The most common topologies include linear, star, or network topologies.

Architectural styles are often strengthened by additional *constraints* which describe, for example, what interactions are allowed or which computations a component may perform. Constraints are often used to induce beneficial properties for the resulting architecture or to keep the architecture “clean”.

9.2.2 Advantages from Styles

The most remarkable advantage is that styles offer solutions for common recurring design problems in software architecture. In this context, styles give sophisticated solutions which were

established over the years and were thoroughly tried and tested by a multitude of software architects. With architectural styles, even less experienced engineers and architects are able to resort to already existing and documented experience [87].

Styles provide also a vocabulary for the communication between the participating architects and engineers in a project [87]. Instead of describing an architecture by circumstantially describing the involved classes, components, connections, and aggregations, it is just necessary to refer to the "publish-subscribe" style, for example.

9.2.3 Describing Styles

There is no unique notation to document architectural styles. Besides informal box-and-lines diagrams, several methods were developed, some especially for the need to describe architectural styles. There are a variety of architectural description languages (ADLs) [55] which focus on various aspects of architectural design and analysis and vary from rather informal to highly formal [93]. Besides ADLs, there are also ontology-based [94], graph-grammar-based [95], or generic formal approaches like Z [96].

There is no single preferred strategy in UML on how to document an architecture [55]. But as UML is the most wide-spread modeling language and understood by the multitude of architects and engineers, UML is often used with an additional textual description in order to describe architectures.

9.2.4 Patterns and Styles

The term "pattern" was originally characterized by the architect Alexander [97, 98]. He described more than 250 patterns that give a solution for a specific design problem in urban and land use planing.

Inspired by Alexander's architectural concept of patterns, Beck and Cunningham [99] began to apply patterns to software. With the following books of Gamma et al. [73] on design patterns and Buschmann et al. [74] on architectural patterns, the idea of patterns in software engineering made its breakthrough.

Design Patterns

Design patterns [73] are generic and proven solutions for common and repeatedly occurring problems in object-oriented development. They focus on the level of program design problems, but are still an abstraction level above the programming language level. Hence, design patterns cannot be directly translated into code. Instead, design patterns give a template for a generic approach that can be applied in many situations.

The original design patterns in [73] are classified into creational patterns, structural patterns, and behavioral patterns. Later on concurrency patterns emerged with [100]. Example patterns are *factory* and *singleton* for creational patterns, *adapter* and *wrapper* for structural patterns, *iterator* and *observer* for behavioral patterns, and *active object* and *double-checked locking* for concurrency patterns.

The main difference between design patterns and architectural styles is the granularity at which they describe a solution [101]. As design patterns are targeted for the level of program design problems, they are, contrary to architectural styles, inapplicable for the enterprise-level system design [86].

Architectural Patterns

Like design patterns, architectural patterns [74] combine experience in software development gathered by engineers and architects. Architectural patterns are also a collection of proven solutions for common occurring problems, like design patterns, but at a much higher level of abstraction than design patterns. An architectural pattern captures also essential elements and knowledge that have proven to provide a system with desired qualities. But contrary to design patterns, architectural patterns are not necessarily bound to object-oriented development.

The difference between architectural patterns and styles is controversially discussed. According to Buschmann et al. [74], architectural patterns and styles are very similar — actually, every architectural style can be described as an architectural pattern. But Buschmann et al. [74] and Taylor et al. [86] emphasize important differences:

1. Styles are independent of each other, whereas patterns depend on the patterns it is built of and on patterns with which it interacts.

2. Styles and patterns have a different scope: patterns apply to a specific design problem, whereas styles apply to a development context.

3. Styles are more abstract. This point arises from the point above. Due to the abstraction in styles, they require human interpretation and do not directly lead to a concrete system design.

On closer examination, it becomes apparent that various architectures, like pipe-and-filter, client-server, blackboard, or implicit invocation can be found in the architectural patterns as well as in the architectural styles. As shown in Section 9.4, it is also possible to build styles out of multiple simpler styles.

According to McGovern et al. [101], architectural styles and architectural patterns are essentially synonymous, whereas an architectural style is a central, organizing concept for a system and an architectural pattern describes a coarse-grained solution at the level of subsystems or modules and their relationships.

In fact, most independent authors [85, 101, 13, 92] claim that architectural styles and architectural patterns are equivalent and the term “pattern” and “style” can just be used synonymously.

9.2.5 Taxonomy of Architectural Style Usage

Architectural styles can obviously be used in the phase of modeling an architecture and to document an architecture. Moreover, Giesecke and Hasselbring [88] identified a total of five different modes of using architectural styles:

1. *Ad-hoc Use of Styles:* This view can also be referred to as the styles-as-patterns view, that is, every style may be expressed as a design pattern. In the ad-hoc usage, the architects have to judge on the basis of their experience regarding the consultation of a pattern collection and the selection of a pattern to apply.

2. *Use as Platform-oriented Styles:* An implementation platform that shall be used may impose an architectural style. At this, the implementation platform can be distinguished into system software, middleware, programming paradigm, hardware, and organizational structure. For example, a middleware may induce a layered architecture as itself is a software layer between the operating system and the application layer.

3. *Use as Customized Styles:* Styles can be customized so that the different variants of an architectural style meet specific needs. For example, the pipe-and-filter style (Section 9.3.2) includes several variants which are intended for different purposes.

4. *Style-based Pre-modeling:* Several architectural models are initially created by using architectural styles, whereas each model addresses a subset of the desired quality characteristics. Each resulting model can then be analyzed. If the models meet the desired quality characteristics, an overall system model is synthesized from the individual models.

5. *Style-based Documentation and Analysis:* Styles are not considered during the architecture phase. They are applied to intermediate or final results of the architecture instead. Doing so may help to document the architecture by interpreting it in terms of a specific architectural style, style-specific analyses may be applied, and it may ease the understanding of the overall system.

9.2.6 Derive Architecture from Styles

In order to use the right architectural style for a specific architectural problem, a process similar to the software engineering process is necessary. First, a requirements analysis on the given problem statement must be done. Depending on the domain, granularity, functional and non-functional requirements, and further constraints, a suitable style has to be chosen. If there are several styles that would be applicable, then the simplest of these should be favored. In the case that no style is suitable, traditional engineering methods must be applied [87].

If a suitable style is found, the style contains instructions on how to implement the given style. However, styles do not prescribe the individual classes and components within an architecture. Instead, a style defines roles that define the architecture. In this respect the architect has to decide what components have to be added or whether any already existing component should adopt such a role.

Styles provide only the blueprint to a specific design problem. For the transformation into a concrete architecture, several decisions can be necessary regarding the right configuration of a style [87].

9.3 Basic Styles

9.3.1 Hierarchical Styles

Layered System

Layered systems are organized into multiple hierarchical layers, with each layer providing a set of services, that is, a group of externally visible operations. Each layer provides this set of services to the layer above it and uses the services from the layer below [85]. A service may also use other services within its own layer [86]. The style is called *opaque* [85, 92] or *strict* [86] if only adjacent layers are allowed to communicate. If a service is allowed to skip one or more layers in order to call another service from a layer which is two or more layers apart, then the style is termed as *transparent* [92] or *nonstrict* [86]. It is also possible to use a layered system which is only partially opaque. This is common in the domain of operating systems where the lowest layer provides the core functionality and can only be accessed by the adjacent layer above. The layer above then provides a common interface to the remaining layers above [85].

The connectors between the layers depend on the intended use of the architecture. If the layers are intended to run on a single machine, then the connectors are typically procedure calls. In a distributed environment the connectors are usually remote procedure calls and network protocols, respectively.

Well known examples for layered systems are layered communication protocols, in particular the OSI model [102]. The lowest layer is usually in charge of the hardware mapping of the connection. Each layer uses the more rudimentary functionality of the layer below and provides in turn a more abstract functionality as basis for the layer above, typically by combining several services from the layer below.

The increasing level of abstraction is also the main advantage of layered systems. It allows to partition a complex problem into a sequence of incremental steps, where each step is built upon the previous step. As long as the specified interface from a lower layer does not change, the layers above are not affected from changes in a lower layer. Lower layers are not affected at all from changes in a layer above. Like in the domain of communication protocols, layered systems are often used to define global standards with defined interfaces. This also allows that different implementors can provide their own implementation of a specific layer.

But a layered system architecture is not always applicable as not every system can be rationally structured in layers. For systems where a layered architecture is applicable, it may be difficult to define the right level of abstraction and for some services it is not obvious to which layer they should belong [85]. Finally, each layer adds overhead and latency to the processing of data, thus an opaque layered system with many levels can become inefficient [86]. In domains where performance is a main criterion, a layered system can thus be unfeasible.

Three-Tier Three-tier architectures are a special case of a layered system with exactly three layers, as depicted in Fig. 9.1. The top-most layer is the *presentation layer*, which represents the front-end. It presents the data to the user and includes all objects that interact with the user, like windows, forms, or Web pages. The middle layer contains the *application* or *business logic*. It contains all control and entity objects and is responsible for the overall processing of data. Such

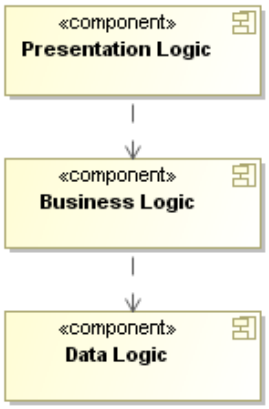


Fig. 9.1: Three-tier architecture

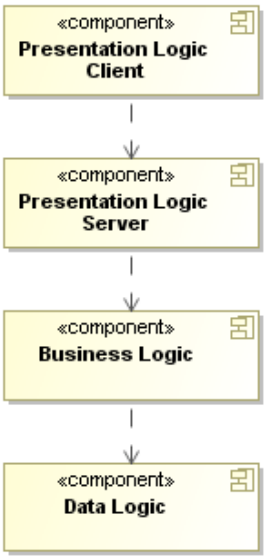


Fig. 9.2: Four-tier architecture

a separation between presentation and application layer allows different user interfaces for the same application logic. Below is the *data layer*, which is responsible for persisting and querying the data. The data layer uses also the repository style (Section 9.3.3) and can be shared by several applications [92, 13].

Four-Tier A four-tier architecture is a specialization of the three-tier architecture with an additional layer. The presentation layer from the three-tier architecture is split in two separate layers, with one layer located on the client machine and the second layer located on the server machine [92]. Fig. 9.2 shows the component diagram for a four-tier architecture. Such a separation can be reasonable if multiple different user interfaces exist and they share common objects. In this case, the common objects can be located on the server-side presentation layer, whereas the specific objects that differ from user interface to user interface are located in the respective client layer.

Client-server

The client-server style shown in Fig. 9.3 is the most frequently encountered one of the architectural styles for network-based applications [103]. This style can be regarded as a layered system with two layers, where the bottom layer represents the server and the top layer represents the clients. Multiple mutually independent clients access the same server via remote procedure calls based on the respective network protocols [86]. The number of servers is not limited to a single server, though. Further servers can provide additional services or serve as redundant backup systems for a primary server. Clients know the available servers but they are usually unaware of the existence of other clients [13].

Fig. 9.3: The client-server style consists of multiple clients which request services from the central server component.

Clients are sometimes distinguished into thin-clients and thick-clients [86]. Thin-clients are those who offer only user interface functions to access the processing functions on a remote server whereas thick-clients include additional processing logic.

A client-server architecture relies on separation of concerns, that is, the (thin-)client contains the presentation logic, whereas the server contains the business and data logic. Thick-clients also contain parts of, respectively the whole business logic. This allows a separate development of the client and the server components, and allows that the components can be evolved independently [103].

A typical example for a client-server architecture is a central database system with multiple clients. Each client provides a user interface and allows the input of data. A client is also responsible to check the input data for sanity, for example range checks on numerical values. When input is complete, the data is sent to the database server, that is, the substantial processing is delegated to the database server. The database must then perform the according transactions and is responsible to ensure the integrity of the data [92].

Architectures based on the client-server style are usually well suited for distributed systems that manage large amounts of data [92] or systems where a central instance provides a set of services to multiple clients. In order to update the components regarding the business or data logic, it is just necessary to update the server. Every client uses the new central components then automatically.

Peer-to-Peer A peer-to-peer architecture (P2P) is derived from the client-server architecture with the difference that every peer is both client and server. This also implies that there is no central instance like in the client-server architecture. In UML this can be depicted as in Fig. 9.4. Like clients and servers, peers are also connected by a network protocol.

Peer-to-peer architectures became popular with the appearance of file-sharing applications like eMule or Gnutella. Besides file-sharing, P2P is also used in P2P-based distributed file systems [104] or for communication systems like Skype [105]. Such applications often use a specialized P2P protocol for communication, especially to locate files in the distributed environment.

In a peer-to-peer system, the failure of single systems has usually no crucial impact on the overall system. The workload can be equally distributed on all participating peers, whereas in the traditional client-server architecture, the main work occurs at the central server system.

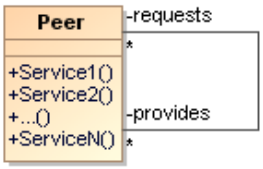


Fig. 9.4: Peer-to-peer architecture: each peer is both client and server at the same time.

The absence of central servers can be both advantage and disadvantage, depending on the application domain. With no central instance there is no single point of failure. But on the other hand, there is no possibility to centrally maintain and administer the crucial components. Thus, sometimes a hybrid approach is chosen like in Skype, where the login procedure is handled by central servers, whereas all the following communication occurs via P2P.

9.3.2 Dataflow

Batch-sequential

The batch-sequential style is a remainder from the early days of computing. It consists of separate programs which are executed in a specified order. Each program uses the data from the previous step as its input and operates on the data. After one program has finished, the output can be used as input for the following program. This separation can be useful for big amounts of data or complex computations, where the overall problem must be subdivided into several single steps.

Batch-sequential processing is, for example, used to process data overnight, when the load of the main system is low and extensive processing does not affect the user's work. In the example of Fig. 9.5, a bank updates all its accounts overnight, based on the transactions that occurred over the day [86].

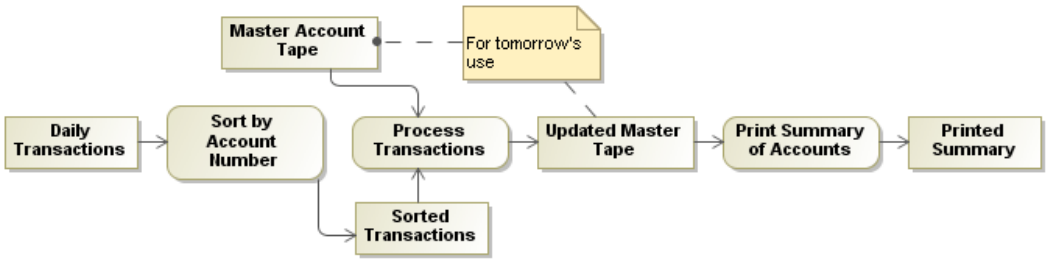


Fig. 9.5: Using the batch sequential style to process financial records. (based on [86])

This style is hardly suitable for applications where interaction between the components is required. Moreover, it does not support concurrency between the components. If additional concurrency is a matter, then the pipe-and-filter style described in Section 9.3.2 can be suitable.

Pipe-and-filter

In the pipe-and-filter style, each component is a filter and the filters are connected through pipes. Each filter receives a stream of data as input and delivers a stream of data as its output. There is no requirement that a filter must process the whole input before its output can begin. Thus, the individual filters can work concurrently by processing the data incrementally so that the output of data can begin before the whole input data is consumed. Because of this behavior the components are called filters. The term pipes originates from conduits which deliver the data [85].

The filters adopt the idea of encapsulation, so each filter must be completely independent from any other filter and they must not share any information. Each filter knows only the type of data it receives as input and of course the type of data it provides as output. In particular, they do not know their adjacent neighbors.

There are several derivations of the pipe-and-filter style. The common pipe-and-filter style does not restrict to any special topology, thus T-fittings are allowed as well as circular connections between the filters. One of the derivations are *pipelines* where filters can only be composed in a linear sequence. Another derivation are *bounded pipes* where the maximum amount of data that can reside on a pipe is limited [85]. The *uniform pipe-and-filter* style constrains that all filters must have the same interface [103].

An example where the pipe-and-filter style is used is the Unix shell. Unix provides a notation to concatenate multiple processes (filters) via pipes [85, 103, 92], for example "`cat readme.txt | grep help | sort`" connects "cat" with "grep" and "grep" with "sort" via the pipe symbol. The process `cat` reads from the file readme.txt and streams all containing lines to the `grep` process. Then `grep` filters every line out which does not contain the word "help" and forwards the remaining lines to the `sort` process where the lines get sorted.

Fig. 9.6: An example for a pipe-and-filter architecture: the Unix shell.

According to Shaw and Garlan [85], a pipe-and-filter style has several advantages. An architecture based on pipe-and-filter is merely a composition of individual filters. Filters support reuse, as existing filters can easily be added to the filter-pipeline as long as any two connected filters exchange the same type of data. Pipe-and-filter systems can easily be re-factored as new filters can be added and old filters can be replaced by improved ones. The separation in encapsulated filters allows an analysis regarding throughput and deadlocks. And as stated in the introduction, the filters can operate concurrently as each filter can be implemented as an individual task. The synchronization occurs implicit through the exchange of data through the pipes.

On the other hand, pipe-and-filter systems lead to a batch-organization of processing [85] and are oriented towards data transformation [55]. Because of the transformational character of such systems, they are hardly suited for interactive applications. Additionally, each filter does not know more than its input and output stream. Thus, a filter can not communicate with its environment. Finally, in the case where the filters use the lowest common data type for the data

exchange, it is necessary that each filter parses the input and reconstructs the original data type before it can operate on it. When the filter forwards the result it must in turn be converted to the common data format. This leads to a higher complexity for the development of each filter as well as a lower performance due to the higher workload.

9.3.3 Shared Memory

Blackboard

By using this style, the components are individual programs known as *knowledge sources* [85, 55], which communicate exclusively through a global memory, called the blackboard. The blackboard can be accessed through direct memory access, procedure call, or database access.

Taylor et al. [86] describes the origin of the style's name as follows:

> «The intuitive sense of the style is one of many diverse experts sitting around a blackboard, all attempting to cooperate in the solution of a large, complex problem. As any given expert recognizes some part of the problem on the blackboard for which he feels competent to solve, he grabs that subproblem, goes away and works on it, and when finished, returns and posts the solution on the blackboard. Posting that solution may enable another expert to identify a problem which he can solve, and so the process continues until the whole problem is solved.»

A blackboard architecture can be implemented in two ways. Either a blackboard manager notifies the interested components of an update on the blackboard [106, 86] via publish-subscribe (Section 9.3.5), which is very similar to the observer design pattern [73]. Or otherwise the programs poll the blackboard in order to determine if a value of interest changed.

Blackboard architecture is used for heuristic problem solving in the domain of artificial intelligence where solution strategies are not a priori known [86] and also in signal processing, like speech and pattern recognition [107, 108, 85]. If a solution strategy for a problem is already known at the time of development, then a blackboard architecture is most likely inappropriate.

Repository

Contrary to the blackboard architecture, the data in a pure repository architecture is always passive [106]. Similar is that there is also a central component, the repository, which can be accessed at the same way. Subsystems can connect to the repository to query, insert, update, or delete data in it. Like in the client-server architecture, there can be several subsystems which are independent from each other, though they may interact through the repository. Fig. 9.7 illustrates such a scenario.

Repositories are typically used for database management systems [92]. The repository is responsible for the integrity of the data and must deal with concurrent access from different subsystems. Centralization to a single instance also allows easier maintenance of the repository.

Besides database management systems, repositories are generally suited for all application domains where data processing occurs on data which is stored in a central location. The drawback is often that as soon as the repository structure is fixed and subsystems are built to operate

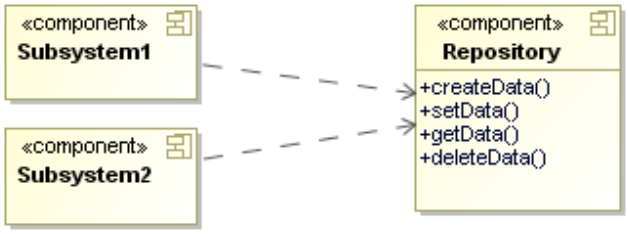


Fig. 9.7: In the repository style, there can be several subsystems that interact with a shared repository. Subsystems are not necessarily aware of each other.

on the data, it becomes difficult to change the structures afterwards as all subsystems rely on the defined structures. Also, the central repository can be a bottleneck with a growing number of connected systems [92].

According to Clements et al. [55], in modern systems the distinctions between blackboard and repository architecture have been blurred, as many database management systems that were originally repositories now provide a triggering mechanism that turns them into blackboards.

9.3.4 Interpreter

Basic Interpreter

The basic interpreter style is also called *virtual machine* because it executes program code in an artificial, software-based, controlled environment [85, 103].

This architecture consists basically of four components, as shown in Fig. 9.8. A memory contains the program code that is to be interpreted. Another component represents the control state of the interpretation engine, based on the program code and the program data. The interpretation engine fetches instructions and data from the interpreter state that are to be interpreted and uses the data memory to fetch and store the current state of the program being interpreted [85]. These four components are typically closely bound by using direct procedure calls as well as shared state [86].

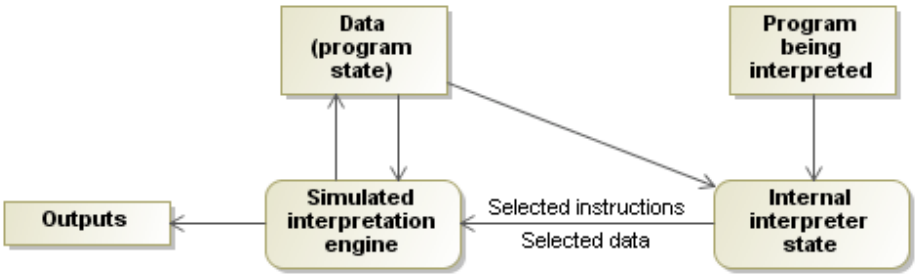


Fig. 9.8: A basic interpreter architecture with its four components. (based on [85])

Microsoft Excel is probably the best known application which implements an interpreter architecture. The formulas entered in an Excel sheet are interpreted by the Excel interpreter engine. Furthermore it is possible to define macros which are in turn executed by the Visual Basic interpreter [86].

A basic interpreter enables a highly dynamic behavior for an application and allows the end-user to define own operations respectively to enhance the capabilities of a program to the end-user's needs. The separation between instruction and implementation on a particular platform also allows a higher portability [103].

Adding a basic interpreter to an application makes the development more complex as it is necessary to manage the evaluation environment and it is hard to predict all cases that an executable will execute when additional code is added by an end-user [103]. As interpreted code takes longer to execute than the similar compiled code, an interpreter may also be unsuited when fast execution of the code is necessary [86].

Mobile code

In distributed environments it can be more efficient to move the code to another location for its execution, for example to the location where a required dataset resides. A mobile code architecture consists of an execution environment similar to the one in the basic interpreter style and also a component which handles receipt and deployment of code and state. The transmission between the latter ones occurs via network protocols [86].

Depending on the location of the components before and after the execution, the component which requested the execution, and the location of the actual computation, the mobile code style can be classified into *code-on-demand*, *remote evaluation* and *mobile agent* [109]. These will be introduced in the following:

Code-on-demand In the code-on-demand style, the client has a set of resources available and downloads code from a remote location that can then be locally executed on the client [109].

By using code-on-demand it is possible to add features to a deployed client. It is also more efficient in terms of performance when code interacting with the user and using local resources is executed locally [103]. Typical examples for code-on-demand are scripting languages like JavaScript or VBScript [86].

Remote evaluation The remote evaluation style is derived from the client-server and basic interpreter style [86]. The client has the code to be executed but not the required resources for the execution. Thus, the client transfers the code to the remote host that has the required resources. The remote host executes the received code and returns the results back to the client [103, 86].

Remote evaluation is basically the inverted approach of code-on-demand and benefits also of the locality of both code and resources. But this style adds also security issues in the case that arbitrary clients can use the server and the server cannot trust the clients [103]. A typical example for remote evaluation would be the domain of grid computing [86].

Mobile agent By using the mobile agent style, the client has code and state, but parts of the resources are located at a remote site. The client transfers the computational component,

consisting of code, state, and required resources to the remote site. The resources may also include possible intermediate results [109].

Contrary to code-on-demand and remote evaluation, the transfer works in both ways [103], but it is not required to return to the initiating client [86]. The main advantage is the dynamism, as an application can stop the execution at one location and move to another location where for example the next dataset for the processing resides [103].

9.3.5 Implicit Invocation

Publish-subscribe

Publish-subscribe is an asynchronous messaging paradigm to exchange messages between clients and servers [110]. Senders of messages do not send their messages to a specific receiver. Rather, subscribers register their interest with publishers by providing a call-back interface to be used [86]. Each publisher maintains a list of subscribers with the according call-backs to the subscribers. When new information is available, then the publisher can notify all subscribers in its list. If a subscriber is not longer interested in receiving notifications, it can also deregister from the publishers list.

The publish-subscribe style is originally known as *news service* [111], but also the now more common name "publish-subscribe" originates from the newspaper domain and describes the relationship of the involved components. In the newspaper domain, a magazine or newspaper publisher periodically creates information and its subscribers obtain a copy of the information or are at least is informed of its availability [86]. In the domain of design patterns, this is similar to the observer pattern [73].

While in smaller applications call-back procedures are sufficient, those are not longer feasible in the case of a larger network-based application. Network protocols are used for this task instead.

The publish-subscribe style is typically used for graphical user interfaces (GUIs), but also in larger message-oriented middleware systems, for example in Java Message Service.

An advantage of this style is that publishers and subscribers are very loosely coupled and subscribers can register or deregister at runtime to services of interest.

With a large number of subscribers a point-to-point protocol relationship between publisher and its subscribers becomes inefficient. Thus, it can be necessary to use intermediary proxies or caches as well as specialized protocols for broadcasts [86].

Event-based

Akin to the publish-subscribe style, the event-based style is characterized by loosely coupled components. The main difference is that there is no discrimination between publishers and subscribers, but instead every component is allowed to send and receive events. Also, the registration of interest to a particular event is only handled by the connectors, not by the components, like in the publish-subscribe style.

Components are connected to a common event-bus, as shown in Fig. 9.9, but do not necessarily know each other. In order to communicate, a component dispatches an event on the event-bus and every connected component receives this event. Thus, a component that sends an

event sends it only to the event-bus where the interested components are listening, while in the publish-subscribe style a publisher would send it to every single subscriber. Components that receive an event can react on it, but may also ignore it [86].

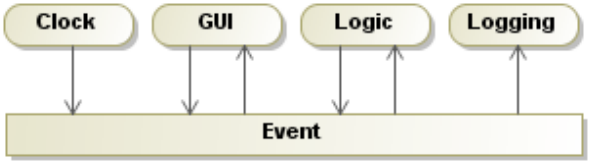


Fig. 9.9: A simple event-based architecture: the clock only sends events, the logger only receives events, GUI and logic can do both.

The event-based style makes it easy to add new components, as a new component can be introduced by simply registering it to the common event-bus. This also supports a high degree of reuse. Components do only need a general event interface and can then be easily attached to other event-buses [85]. This allows also that heterogeneous applications communicate through the event-bus as long as they share the same event interface [86]. Components that are connected to the event-bus can be refactored or replaced by other components without affecting the other components on the bus [112].

The main disadvantage of this style is that, contrary to procedure calls, there is no guarantee that another component reacts on an event nor is guaranteed in which order events will arrive [85]. With many participants on a common event-bus, the event-bus may become a bottleneck and is by all means the single point of failure [103, 86]. If it is necessary to exchange big chunks of data, then the event-bus may also be unsuitable for the data-exchange. Alternatively some kind of side-channel like a shared repository should be used then [85].

9.4 Compound Styles

9.4.1 C2

The C2 style is built based on several styles mentioned in the previous section. Its predecessor is Chiron-1 [113], a user interface development system. This style is referred to as Chiron-2, or C2 style for short [114].

The idea behind the development of C2 is the need for a component-based development, especially for user interfaces. C2 is designed for a higher reuse of UI components and a highly independent component structure [114]:

> «Components may be written in different programming languages, components may be running concurrently in a distributed, heterogeneous environment without shared address spaces, architectures may be changed at runtime, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active and described in different formalisms, and multiple media types may be involved.»

Though the style was initially targeted for graphical user interface applications, it was found to be suitable for many applications outside the domain of GUIs [86].

To enable a high degree of reuse and flexible composition of components, it combines the event-based style with a layered client-server architecture. Multiple concurrent components can be connected through message-routing connectors. Regarding the connection of components and connectors, only three options are allowed. Every component and every connector has a defined “top” and “bottom” interface, whereas only a “top interface” can be connected with a “bottom interface” and vice-versa. Components can not be connected directly, but through an intermediate connector. Connections between different connectors are allowed. With these rules a layering of components is induced [86]. Fig. 9.10 shows a simple C2 architecture with two of such layers.

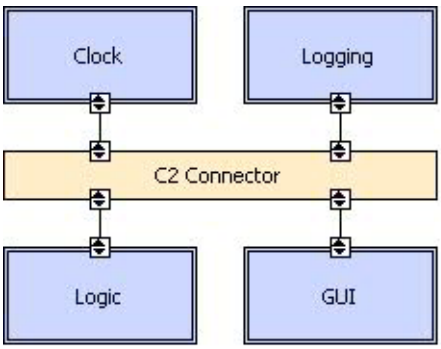


Fig. 9.10: A simple C2 architecture (xADL diagram [115])

The components in such an architecture communicate asynchronously via messages. Messages are either notifications, for example due to the change of an object, or requests. Notifications are sent only “downwards” in the architecture, whereas requests can only be “upwards”. With this constraint, a component within this architectural hierarchy is only aware of components above, that is, those it may request services from. A component can send notifications to the components below, though it does not know what, if any, components there are [114, 86].

If a component would directly request services from upper components, the components would be closely dependent and the reusability would suffer. In order to avoid this, C2 induces the notion of domain translation. Domain translation transforms both requests and notifications into the specific form understood by the respective recipient [114].

To enable concurrency of the components and due to the asynchronous nature of this style, each component can have its own thread or even multiple threads. Thereby, the architecture can also be spread in a distributed environment.

The benefits of this style is an aggregation of the benefits of the used styles described in Section 9.3. Taylor et al. [86] summarize them as follows:

- Substrate independence: ease in modifying the application to work with new platforms.
- Applications are composable from heterogeneous components running on diverse platforms.

- Support for product lines by substituting one component for another
- Ability to design in the model-view-controller style, but with very strong separation between the model and the user interface elements.
- Natural support for concurrent components.
- Support form network-distributed applications.

The C2 style is mainly suited for larger applications with several independent components. The routing of events through the layers can be complex, hence multiple layers with lots of events can be ineffective. For simple applications, the overhead induced by the component interaction may be to high.

9.4.2 REST

The World Wide Web was originally targeted for groups of researchers, but in late 1993 it became clear that more than just researchers would be interested in the Web [103]. The public awareness led to an exponential growth, so that 15 years later in July 2008, Google reported [116] that its search engine has indexed 1 trillion unique URLs.

The early architecture assumed direct connections between clients and server, in particular, the Web was not aware of techniques like caching or proxies. With the rapid growth of the Web on the one side and some poor characteristics of the network on the other side, a collapse of the Internet infrastructure had to be anticipated [103].

According to Fielding [103], the Representational State Transfer (REST) style "has been developed to represent the model for how the modern Web should work". Like C2, REST is derived from several simpler architectural styles and additional constraints.

Client-server REST is based on the client-server style, introduced in Section 9.3.1. This separation of concerns into a client and a server part allows that both components can evolve independently. That is, the client development can focus on the user interface concerns, while the server development can focus on data storage concerns, for example. This simplifies the development of client and server components and improves the portability of the user interface across multiple platforms.

Stateless A stateless client-server architecture is derived from the usual client-server architecture with the additional constraint that the server must not hold any session state. Instead, the session state must be kept on the client. On a request, the client has to include all necessary information to perform the request.

With no session state on the server, in particular no records of previous requests, each request must be self-contained. With multiple identical servers, for example to perform a load-balancing, each request can be handled by a different server. Allocated resources can immediately be freed after a request is completed.

But this induces a higher network usage, as each request must contain all necessary informations. It also transfers the control over the current state from the server to the client, allowing a malicious client to alter the state.

Cache A stateless client-server architecture can be further extended by adding caches. If a response to a request is cacheable, then responses can be stored in a cache for a later reuse. If an identical request arrives, then the request does not need to be executed again, but instead the previous result from the cache can be used.

Caches can be located either near server, as gateways, or near the clients as proxies or local caches [86]. This replication of information can improve the system performance as some interactions may be partially or completely eliminated.

It must be assured that the data in the cache does not differ from the actual result that would have been obtained after an execution of the request. Otherwise the reliability of the systems suffers.

Uniform Interface The REST style relies on a uniform interface between components in order to simplify and decouple the architecture. To obtain a uniform interface, REST defines four constraints.

1. *Identification of resources:* In REST, a resource is the key abstraction. On a request, a resource identifier is used to identify a particular resource. In a Web-based environment, this would typically be a URI (Uniform Resource Identifier).

2. *Manipulation of resources through representations:* A representation consists of data and metadata to describe that data. Representations are used to transfer the captured current or intended state of a resource between components.

3. *Self-descriptive messages:* Each request contains all informations that are needed to execute the request.

4. *Hypermedia as the engine of application state:* On a successful response, related resources should be included in a representation via their identification, for example the URI.

Layered System A REST architecture can comprise several opaque layers (Section 9.3.1), that is, components can only use the adjacent layers but cannot bypass intermediate layers nor are they aware of other components behind the adjacent layers.

This allows to encapsulate legacy services but also to improve the systems scalability. Intermediate components, such as proxies, can provide an interface to a service, hiding the back-end system in a layer below. As each request must be self-contained, the proxy is able to select one of several equivalent servers that should execute the request. It allows also that a request can be partially processed by an intermediary or filtered, due to security concerns [86].

The combination of the layered system style and the uniform interface constraint creates a data-flow network with filter components, similar to the uniform pipe-and-filter style.

Code-on-demand REST allows optionally to add the mobile code variant code-on-demand (Section 9.3.4) to extend the clients functionality by transferring code to the client. This includes, for example, pre-compiled Java applets, but also scripts like JavaScript. This allows to add features after the deployment, but some code, like Java applets, may be blocked by firewalls.

Although REST is a generic architectural style and not limited to the use within the Web, the Hypertext Transfer Protocol (HTTP) is the primary application-level protocol for communication between Web components [103]. Nevertheless, it is also possible to design an architecture that conforms with REST, without using HTTP.

REST unites multiple benefits within one hybrid style by combining several simpler styles and its constraints to one compound style. These constraints can be concentrated within six REST principles [117, 118]:

1. The key abstraction of information is a resource, identified by a URL.
2. The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes.
3. All interactions are context-free.
4. Only a few primitive operations are available.
5. Idempotent operations and representation metadata are encouraged in support of caching.
6. The presence of intermediaries is promoted.

9.5 Conclusion

Less experienced software architects and engineers often face the problem that they agonize about design problems in architecture, like several other architects have done before, thereby reinventing the wheel over and over again. In this respect, architectural styles are a reasonable methodology on how expertise can be documented. Hence, architectural styles provide knowledge on architectural decisions that have proven several times before and are known to elicit desired properties from the resulting architecture. Moreover, architectural styles help as a common vocabulary to describe the characteristics of an architecture, provided that everyone involved knows this vocabulary.

The broad acceptance of styles and patterns in the domain of software engineering led to a multitude of books and publications that try to catalog and analyze them. Any good architect should not only be aware of this, but also be acquainted with the most common styles. Architectural styles are not only helpful during the construction of an architecture, but also for the understanding of existing architectures.

However, the use of architectural styles (and patterns) is sometimes refused because they might look complex at a first glance and might also entail additional constraints. Initially this seems counterproductive and limiting for the development. Though, styles allow to establish a clean and simple architecture, which eases the development ultimately.

But architectural styles provide no silver bullet solution for all kinds of architectural problems. Instead, architectural styles can be seen as “tools” a software architect can use for the construction of a software architecture. Hence, it is up to the architect to choose an architectural style that meets the requirements for a specific architecture. Applying a style that does not fit to the problem domain may entail drawbacks and limitations without adding any advantages.

Bibliography

[1] David Garlan. Software architecture: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM Press.

[2] Manfred Broy. *Lecture: Software Architectures*. 2008.

[3] Robert C. Martin. Design principles and design patterns. Technical report, www.objectmentor.com, 2000.

[4] Wikipedia. Design pattern (computer science). `http://en.wikipedia.org/wiki/Design_pattern`, 2009. [Online; accessed 20-January-2010].

[5] Wikipedia. Architectural pattern (computer science). `http://en.wikipedia.org/wiki/Architectural_pattern`, 2009. [Online; accessed 20-January-2010].

[6] Wikipedia. Functional requirement. `http://en.wikipedia.org/wiki/Functional_requirement`, 2009. [Online; accessed 20-January-2010].

[7] Wikipedia. Non-functional requirement. `http://en.wikipedia.org/wiki/Non-functional_requirement`, 2009. [Online; accessed 20-January-2010].

[8] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, Boston ; Munich [u.a.], 2003.

[9] IEEE. Recommended practice for architectural description of software-intensive systems, 2007.

[10] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, Boston ; Munich [u.a.], 2005.

[11] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Trans. Software Eng.*, 21(4):269–274, 1995.

[12] Hans van Vliet. *Software Engineering Principle and Practice 3rd. edition*. John Wiley & Sons, Ltd, third edition edition, 2006.

[13] Ian Sommerville. *Software Engineering*. Pearson Education, 8th edition, 2007.

[14] Dr. Alistair Cockburn. Using both incremental and iterative development. *Defense Software Engineering*, 2008.

[15] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, third edition edition, 2004.

[16] James Rumbaugh Ivar Jacobson, Grady Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[17] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, second edition edition, 2004.

[18] Stefan Van Baelen Andrew Wils. Software architecture and extreme programming. *Information Technology of European Advancement*, 2007.

[19] David L. Parnas. *Software Fundamentals*, chapter 7, pages 143–155. Addison-Wesley, 2001.

[20] Wikipedia. Decomposition (computer science). http://en.wikipedia.org/wiki/Decomposition_(computer_science), 2008. 2009-12-10.

[21] Gernot Starke. *Effektive Software-Achitekturen*. Hanser Verlag, 2008.

[22] ISO. Software engineering – product quality. Technical report, ISO, 2001.

[23] Johannes Bergsmann. Nicht-funktionale anforderungen. *Quality Knowledge Letter*, (5), 12 2004.

[24] DIN/EN/ISO. Medical device risk management. Technical report, ISO, 2007.

[25] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[26] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[27] Gerd Beneken, Ulrike Hammerschall, Manfred Broy, Maria Victoria Cengarle, Jan Jürjens, Bernhard Rumpe, and Maurce Schoenmakers. Componentware - State of the Art 2003. In *Proceedings of the CUE Workshop Venedig*, 2003.

[28] D. Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 138–155, 1968.

[29] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Putting the parts together concepts, description techniques, and development process for componentware. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8050, Washington, DC, USA, 2000. IEEE Computer Society.

[30] Frederick P. Brooks. mitp, first edition, 2003.

[31] Clemens Szyperski. *Component Software -Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[32] Johannes Siedersleben. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt Verlag, 2004.

[33] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware methodology based on process patterns. 1998.

[34] *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[35] John Cheesman and John Daniels. *UML Components, A Simple Process for Specifying Component-Based Systems.* Addison-Wesley, 2001.

[36] Philippe Kruchten. *The Rational Unified Process, An Introduction*. Addison-Wesley, second edition, 2000.

[37] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1058, 1972.

[38] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pages 293–324, 2002.

[39] I.M. Holland. Specifying reusable components using contracts. In *ECOOP*, volume 92, pages 287–308. Springer, 1993.

[40] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *Object-technologies for advanced software: Second JSSST International Symposium, ISOTAS 96, Kanazawa, Japan, March 1996: proceedings*, page 22. Springer, 1996.

[41] Y. Smaragdakis and D. Batory. Implementing reusable object-oriented components. In *Int. Conference on Software Reuse*, 1998.

[42] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. *Lecture Notes in Computer Science*, 1445:550–570, 1998.

[43] Craig Walls. *Modular Java: Creating Flexible Applications with OSGi and Spring*. Pragmatic Programmers, 2009.

[44] et al. Kiczales, G. Aspect-oriented programming. In *Proceeding of the European Conference on Object-Oriented Programming(ECOOP)*. Springer Verlag, 1997.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, pages 327–353, 2001.

[46] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE)*, pages 107–119, 1999.

[47] Dirk Staehler, Ingo Meier, Rolf Scheuch, Christian Schmoelling, and Daniel Somssich. Carl Hanser Verlag, 2009.

[48] Wikipedia. Enterprise application integration. `http://en.wikipedia.org/wiki/Enterprise_Application_Integration`, 2009. [Online; accessed 20-January-2010].

[49] Gregor Engels, Andreas Hess, Bernhard Humm, Oliver Juwig, Marc Lohmann, and Jan-Peter Richter. *Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten*. dpunkt Verlag, 2008.

[50] Wikipedia. Service-oriented architecture. `http://en.wikipedia.org/wiki/Service-oriented_architecture`, 2009. [Online; accessed 20-January-2010].

[51] Ali Arsanjani. Service-oriented modeling and architecture. http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/, 2004.

[52] Micheal Bell. *Service Oriented Modeling*. John Wiley & Sons, 2008.

[53] Wikipedia. Service-oriented modeling. `http://en.wikipedia.org/wiki/Service-oriented_modeling`, 2009. [Online; accessed 20-January-2010].

[54] Stefan Malich. Betriebswirtschaftlicher Verlag Dr. Th. Gabler / GWV Fachverlage GmbH, Wiesbaden, 2008.

[55] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting software architectures*. 2003.

[56] Mark Shereshevsky, Habib Ammari, Nicholay Gradetsky, Ali Mili, and Hany H. Ammar. Information theoretic metrics for software architectures. In *COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, page 151, Washington, DC, USA, 2001. IEEE Computer Society.

[57] M. Ionita, D. Hammer, and H. Obbink. Scenario-based Software Architecture Evaluation Methods: An Overview. In *Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering*, 2002.

[58] Muhammad Ali Babar and Ian Gorton. Comparison of Scenario-Based Software Architecture Evaluation Methods. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 600–607, Washington, DC, USA, 2004. IEEE Computer Society.

[59] Stefan Eicker, Christian Hegmanns, and Stefan Malich. Technical Report 14, March 2007.

[60] Ralf Reussner and Wilhelm Hasselbring. *Handbuch der Software-Architektur: Werkzeuge für Controlling und Management*. dpunkt Verlag, 2nd edition, 2008.

[61] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.

[62] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[63] Rick Kazman Mark Klein. Attribute-based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, october 1999. available at http://www.sei.cmu.edu/reports/99tr022.pdf.

[64] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[65] Mikael Lindvall, Roseanne Tesoriero Tvedt, and Patricia Costa. An Empirically-Based Process for Software Architecture Evaluation. *Empirical Softw. Engg.*, 8(1):83–108, 2003.

[66] David Bettencourt da Cruz and Birgit Penzenstadler. Designing, Documenting, and Evaluating Software Architecture. Technical Report TUM-INFO-06-I0818-0/1.-FI, Technische Universität München, Institut für Informatik, jun 2008. available at http://www.in.tum.de/forschung/publikationen/index.html.de.

[67] W.F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[68] Marc van Kempen, Michel Chaudron, Derrick Kourie, and Andrew Boake. Towards proving preservation of behaviour of refactoring of UML models. In *SAICSIT '05: Proceedings of the 2005 annual research conference ofthe South African institute of computer scientists and information technologistson IT research in developing countries*, 2005.

[69] I. Ivkovic and K. Kontogiannis. A framework for software architecture refactoring using model transformations and semantic annotations. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, 2006.

[70] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[71] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[72] D. Tamzalit and T. Mens. Using graph transformation to evolve software architectures. *Preproceedings of BENEVOL 2008, Eindhoven University of Technology, The Netherlands.*, pages 31–41.

[73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[74] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. 1996.

[75] William Brown, Raphael Malveau, and Hays McCormick. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[76] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Transactions on software Engineering*, pages 638–653, 2002.

[77] B. Tekinerdogan. Asaam: aspectual software architecture analysis method. In *WICSA Proceedings*, 2004.

[78] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[79] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Computer Science*, 72(4):59–70, 2003.

[80] J. Philipps and B. Rumpe. Refinement of information flow architectures. In *ICFEM*, pages 203–212, 1997.

[81] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[82] A.G. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[83] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. *Lecture Notes in Computer Science*, pages 144–158, 2003.

[84] C. Riva, P. Selonen, T. Systa, and J. Xu. UML-based reverse engineering and model analysis approaches for software architecture maintenance. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Washington, DC, USA, 2004.

[85] David Garlan Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[86] Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. 2008.

[87] Michael Stal. Software-muster. In Ralf Reussner and Wilhelm Hasselbring, editors, *Handbuch der Software-Architektur*, chapter 17, pages 331–355. Dpunkt Verlag, 2006.

[88] Simon Giesecke. Taxonomy of architectural style usage. In *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*, pages 1–10, New York, NY, USA, 2006. ACM.

[89] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[90] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 9–20, New York, NY, USA, 1993. ACM.

[91] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[92] Bernd Bruegge and Allen H. Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 3rd edition, 2009.

[93] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14:43–52, 1997.

[94] Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. Ontology-based modelling of architectural styles. *Information and Software Technology*, 51(12):1739 – 1749, 2009. Quality of UML Models.

[95] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Softw. Eng.*, 24(7):521–533, 1998.

[96] J. Michael Spivey. *The Z Notation: a reference manual*. Prentice Hall, 1989.

[97] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[98] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[99] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. In *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming.*, 1987.

[100] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[101] James McGovern, Scott W. Ambler, Michael E. Stevens, James Linn, Elias K. Jo, and Vikas Sharan. *The Practical Guide to Enterprise Architecture*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[102] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[103] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[104] Bernhard Amann, Benedikt Elser, Yaser Houri, and Thomas Fuhrmann. Igorfs: A distributed p2p file system. In *Proceedings of the Eigth IEEE International Conference on Peer-to-Peer Computing (P2P'08)*, Aachen, Germany, September 8 – 11, 2008. IEEE Computer Society.

[105] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. In *BlackHat Europe*. EADS Corporate Research Center, 2006.

[106] Gustav Pomberger and Wolfgang Pree. *Software Engineering: Architektur-Design und Prozessorientierung*. Hanser, 3rd edition, 2004.

[107] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and Raj Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Comput. Surv.*, 12(2):213–253, 1980.

[108] H. Penny Nii. Blackboard systems. *AI Magazine*, 7(2):38–53,82–106, 1986.

[109] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[110] Zafar U. Singhera. Modeling load on a publish/subscribe system. In *DEXA '08: Proceedings of the 2008 19th International Conference on Database and Expert Systems Application*, pages 706–710, Washington, DC, USA, 2008. IEEE Computer Society.

[111] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[112] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.*, 1(3):229–268, 1992.

[113] Richard N. Taylor, Kari A. Nies, Gregory Alan Bolcer, Craig A. MacFarlane, Kenneth M. Anderson, and Gregory F. Johnson. Chiron-1: a software architecture for user interface development, maintenance, and run-time support. *ACM Trans. Comput.-Hum. Interact.*, 2(2):105–144, 1995.

[114] R.N. Taylor, N. Medvidovic, K.M. Anderson, Jr. Whitehead, E.J., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, Jun 1996.

[115] Eric M. Dashofy, André Van der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 103, Washington, DC, USA, 2001. IEEE Computer Society.

[116] Jesse Alpert and Nissan Hajaj. The official google blog: We knew the web was big... http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html, Jul 2008. [Online; accessed 26-January-2010].

[117] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[118] Justin R. Erenkrantz, Michael Gorlick, and Richard N. Taylor. Rethinking web services from first principles. In *2nd International Conference on Design Science Research in Information Systems and Technology*, Pasadena, California, May 2007.